RUGRAT: Runtime Test Case Generation using Dynamic Compilers

Ben Breech, Lori Pollock and John Cavazos
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19711
{breech,pollock,cavazos}@cis.udel.edu

Abstract

The testing of error handling and dynamic security mechanisms often depends on reproducing specific conditions outside the realm of an application's normal program state. We present RUGRAT*, a novel technique to automatically generate tests for these challenging test situations.* RUGRAT *uses a dynamic compiler to add instructions to the program during execution, and thus dynamically generates tests to exercise code designed to handle uncommon situations during program execution. The* RUGRAT *testing approach is independent of the source language, requires no modification to the source or binary program under test and generates runtime tests automatically based on a simple test specification. We demonstrate* RUGRAT*'s capabilities by targeting two particular uncommon situations: handling errors from system and application calls, and testing security mechanisms that protect a program against attacks on function pointers. Both code coverage and failure detection results indicate that* RUGRAT *is a cost effective approach that reduces the number of required test inputs and need for vulnerable programs.*

1 Introduction

Due to limited time and resources, testers often target "common" situations, i.e., those situations likely to be encountered by users of the application. Unfortunately, the infrequently executed code that remains untested often involves important security mechanisms or error handling code [8, 25]. Without testing, inadequate handling of errors such as memory allocation or disk access errors can lead to severe program failures in the field.

Generally, programs contain error handling code designed to accommodate abnormal inputs or environmental conditions. Sometimes, testing error handling code can be incorporated as part of the testing process, particularly when the code can be exercised by easily generated test cases such as code to handle bad user input [20], or through the use of fuzz testers [17], which generate random data for the program to handle. However, testing code that handles conditions beyond invalid user input, such as resource depletion or other environmental conditions, can become practically impossible because these conditions are extremely difficult to reproduce at the application level. For example, executing code that handles memory allocation errors, file I/O errors, or other resource errors requires notification from the operating system.

To cover error handling code beyond handling bad user input, researchers have developed fault injection techniques [12, 26] that attempt to force the program to execute error handling code. Probabilistic approaches, such as those that randomly change values in memory, are limited because they may not trigger particular error handling code [26], while compiler-guided exception raising [12] requires compiler-inserted extra code changing the code under test.

Other approaches require the tester to provide additional code to exercise error handling code. The tester may replace particular library functions with stub functions [6, 23] that return values indicating failure. Another approach is to use Aspect Oriented Programming [21] to provide wrapper code that causes the program to exercise error handling code. Both approaches require the tester to provide additional source code that can become complicated when more fine grained control (e.g., exercising the error handling code only at a particular callsite) is needed.

Dynamic security mechanisms [9, 28] are also difficult to test because they require carefully crafted inputs [10] that require particular domain knowledge [1] to exploit vulnerabilities. Dynamic security mechanisms, which execute in conjunction with the program, react to suspicious behavior detected at runtime to prevent exploitations of program vulnerabilities. Current testing approaches require vulnerable programs on which to test, but there is a lack of suitable inputs [10].

In this paper, we present and evaluate an automated approach to generating tests for the uncommon cases of error handling and dynamic security mechanisms which de-

pend upon difficult-to-simulate environmental conditions. We have implemented our technique as a testing framework called RUGRAT (RUntime GeneRAtion of Tests), capable of simulating uncommon situations to exercise rarely executed code. RUGRAT takes as input a test specification and a compiled binary program. During execution, RUGRAT generates tests by adding instructions into the executing instruction stream to simulate the desired test conditions. The key insight behind RUGRAT is that automatic runtime test generation for uncommon situations can be performed by running the program through an automatic test generator working with a dynamic compiler.

The RUGRAT approach to testing has several benefits. Runtime tests are generated automatically based on a simple test specification. The runtime tests simulate dynamic conditions that are difficult for a tester to reproduce. The testing system is independent of the source programming language of the program under test because RUGRAT takes binary code as input. Unlike binary rewriters, RUGRAT requires no modification of the source or binary of the program under test. That is, while RUGRAT adds instructions into the executing instruction stream, the original binary code remains unchanged when testing is complete. The dynamic compiler is only used during testing, and thus the program under test need not execute with a dynamic compiler outside the testing environment. The RUGRAT testing framework is layered on top of the dynamic compiler and could be implementable with any dynamic compiler. RUGRAT requires little knowledge of the inner workings of the dynamic compiler and does not need to generate stubs for system calls. RUGRAT could be used during maintenance or during initial development.

We demonstrate RUGRAT's capabilities by targeting two important uncommon situations: handling errors from system and application calls, and testing security mechanisms that protect a program against attacks on function pointers. Our previous system [5] was limited to testing mechanisms protecting against stack smashing. This paper demonstrates that the RUGRAT framework can now be applied to other testing tasks, and thus is more generally applicable than the initial system that targeted stack smashing. Our specific contributions are

- an overview of the RUGRAT testing architecture and process for testing uncommon situations;
- instantiation of the runtime test generator for (1) testing a program's handling of errors triggered by the failure of certain system and application calls, including revealing omission errors, and (2) testing certain security mechanisms, such as those designed to protect function pointers;
- a quantitative evaluation of the effectiveness of RUGRAT with respect to code coverage and a case study of failure detection capability.

```
if ((sptr = malloc(size+1)) == NULL) {
    findmem();
    if ((sptr = malloc(size+1)) == NULL)
      xlfail("insufficient string space");
}
```

Figure 1: Error handling code in the 130.li lisp interpreter

Our experimental results indicate that runtime test generation can significantly increase the coverage and fault exposure within error handling code and function pointer protection mechanisms with much fewer test inputs and without the limitations of other approaches.

After describing the testing challenges and state of the art in Section 2, we present the RUGRAT testing framework and testing process in Section 3. Section 4 describes the specific runtime test generation strategies for error handling and function pointer protection mechanisms. We present our experimental evaluation study in Sections 5 and 6 followed by conclusions and future directions.

2 Testing Challenges and State of the Art

This section illustrates the challenges of testing error handling code and dynamic security mechanisms that protect function pointers, and overviews the current approaches.

2.1 Error Handling

Consider the code snippet shown in Figure 1, which comes from the `130.li` application in the SPEC application suite [22]. The code attempts to allocate memory through `malloc`. If the allocation fails, the function `findmem` tries to find some buffers that could be freed and then the allocation is retried. `findmem` is only executed at this callsite when the program is out of memory. Covering the `findmem` callsite shown in Figure 1 requires the test case to somehow allocate all available memory. The difficulty in causing `malloc` to fail means that the `findmem` callsite will likely go uncovered during testing. More generally, error handling code may be omitted entirely, which may not be detected during testing.

Besides the work described in the introduction, other approaches include modifying the memory image of library code to return different values for system calls [24], providing stubs for common system library functions to return different values [6, 23], and generating data to trigger certain exceptions [25]. Techniques closely related to RUGRAT include Holodeck [24], FIG [6] and related approaches [23]. These approaches replace dynamic library functions with stub functions. The stub functions, which are called by the

```
work = unzip;
// stmts that may change work
for (;;) {
   if ((*work)(ifd, ofd) != OK){
// other non-pointer code removed
```

(a) C code with indirect function call

```
movl   $unzip, work
// ...
movl    work, %ecx
// setup arguments
call    *%ecx
```

(b) Unprotected assembly code

```
  movl    $unzip, %edx
  xorl    __mech_key, %edx
  movl    %edx, work
// ...
  movl    work, %ecx
  xorl    __mech_key, %ecx
  // setup arguments
  call    *%ecx
```

(c) Assembly code with security mechanism

Figure 2: Example of a security mechanism for function pointer usage

application, decide whether to fail the call by returning an error value or allow the call to proceed normally by calling the actual library call. The tester is responsible for providing, and modifying, the stub functions. These approaches tend to require extensive effort from the tester and/or require modification of the program source or binary, which changes the code being tested. Neither the FIG nor the Holodeck approach target application function calls. RUGRAT removes this limitation and extensive testing effort by using a dynamic compiler.

2.2 Pointer Protection Mechanisms

Function pointers are useful components of (usually C-like) programming languages that provide the capability to indirectly call functions, enable callbacks, simplify some portions of code, and allow for basic reuse of code. Unfortunately, function pointers can be subverted for arbitrary execution of malicious code [9]. An attack can be performed by overflowing a buffer [1], allowing the attacker to put an arbitrary value in the function pointer. When invoked as a call, the function pointer jumps to arbitrary code specified by the attacker.

Determining that a program is vulnerable to such attacks can be difficult. Static analysis tools can examine a program for potential vulnerabilities, but can miss some vulnerabilities and report false positives [15]. Array bounds checking is a more complete solution, but can be expensive [3]. Various dynamic security mechanisms that execute in conjunction with the program have been developed to combat attacks against function pointers [9, 28]. The key insight of these approaches is that an attack can be thwarted by encrypting the function pointer while in memory and decrypting the function pointer just before it is dereferenced.

Figure 2a shows an example in the application `164.gzip` [22], where a different function, such as different compression/decompression, executes depending on the value of the `work` function pointer. Figure 2b shows the equivalent assembly code generated by a compiler without any security mechanism. The function pointer, `work`, is first initialized to the address of the function `unzip`. Next, the function referenced by `work` is invoked by copying the value of `work` into the `ecx` register and issuing the `call *%ecx` instruction. Figure 2c shows the same assembly code with the security mechanism added to protect the function pointer usage. After initialization, `work` is encrypted by xoring the address with the key. Invoking the function referenced by `work` now requires an extra instruction to decrypt the value. When an attack occurs, the attacker overwrites the value of the encrypted pointer with the address of malicious code. This value is decrypted, resulting in a bad address being passed to the function call. The program jumps to the bad address and will likely incur a segmentation fault instead of executing the attacker's code.

To test such a security mechanism, the mechanism is applied to a program known to be vulnerable to attacks against function pointers. The program is then run twice with the mechanism; once with normal inputs to ensure the mechanism has not broken the program and once using the exploit as input to ensure the mechanism protects the program's function pointers.

This testing process is usually inadequate since few exploitable programs are available for testing [10]. Without systematic testing of each of the program points where the mechanism is applied, some errors may not be exposed until the mechanism is widely deployed. In previous work [5], we described methods to dynamically generate test cases to test security mechanisms that protect against stack smashing attacks [1]. In this paper, we demonstrate how the approach can be generalized to other security mechanisms, by generating runtime tests for security mechanisms to protect function pointer usage.

3 The RUGRAT Testing Process

RUGRAT creates runtime tests, or *dynatests*, during program execution. The dynatests are executable instructions added to the program to modify the execution state of the program to simulate an uncommon situation. Thus, a RUGRAT test case consists of both a normal input to the program and a dynatest created at run time.

Figure 3 shows a high level view of the RUGRAT framework. RUGRAT requires two inputs from the tester. The first input is the test specification, which details the type of

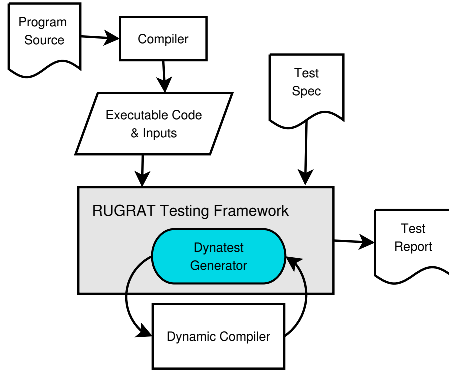

Figure 3: RUGRAT Testing Framework

dynatests that are to be created and the points to be targeted for testing. The second input to RUGRAT is the compiled program and normal program inputs to be executed. The output of RUGRAT is a test report describing the pass/fail result of the tests.

The testing specification provides the information that RUGRAT needs to create and insert dynatests. The most important information in the specification is the set of points in the program that should be tested. The tester can specify these points generally (e.g., all call sites) or specifically (e.g., call to X in function Y). In the current implementation, the testing points are specified through a set of environment variables that detail the particular location (e.g., call site address) where the Dynatest generator should insert dynatests. Program addresses do not need to be computed by the tester; RUGRAT determines the addresses by executing the program once and provides the list to the tester to select from. No undue burden is imposed on the tester for specifying testing locations. Other information in the testing specification is specific to the particular testing task, such as the kind of error conditions to be tested (see section 4), and an oracle for the task (e.g., how should the program behave when performing this particular testing task).

RUGRAT interacts with a dynamic compiler to perform testing tasks. Dynamic compilers are commonly used to speed up interpreted code (e.g., all JVMs are dynamic compilers) and used for on-the-fly program optimizations [2]. Dynamic compilers also have been used to ensure a program does not allow unauthorized control transfers [14], to locate faults by dynamically switching predicates [27], and to perform coverage analysis [18] and impact analysis [4].

The dynamic compiler enables RUGRAT to analyze and modify the state of the executing program. RUGRAT's dynatest generator module is responsible for communicating with the dynamic compiler. During program execution, the dynamic compiler provides information about the program state to the generator, which then creates dynatests based on the test specification. The generator gives the dynatests to the dynamic compiler for insertion into the executing instruction stream.

Figure 4 shows a more detailed view of how RUGRAT interacts with the dynamic compiler. For our prototype, we have used the DynamoRIO [7] dynamic compiler, which is capable of running a wide variety of statically compiled programs. During execution, DynamoRIO partitions the program's instructions into sequences called basic blocks[1]. After a basic block is created by the dynamic compiler, the block is input to the dynatest generator, which analyzes the instructions in the block. If the generator determines that a dynatest should be added to the block, the generator creates instructions to simulate uncommon situations, such as an out of memory error. The generator then gives the basic block, which now includes any dynatests, back to the dynamic compiler. The basic block is passed by the dynamic compiler to the CPU for native execution. A new program state is created after the execution of the basic block. The dynamic compiler creates the next basic block and the process continues until the program terminates. The test oracle examines each new program state to determine if a dynatest passed or failed. The definition of success for each dynatest depends on the test specification.

4 Dynamic Test Generation Strategies

The specifics of dynatest generation vary depending on the testing task, which is described by the test specification. This section describes the strategies for generating dynatests for error handling code and function pointer protection mechanisms.

4.1 Testing for Error Handling

RUGRAT's dynamic test generation is especially useful for testing error handling that requires hard to simulate external environmental conditions (e.g., out of memory, disk write failure, process creation failure). The key insight behind RUGRAT is that the program's error code can be tested by selectively failing the related functions. RUGRAT's test generation strategy for error handling can be applied to both application and system library calls. Testing code that handles application call errors, however, may introduce complexities due to side effects, such as modifications to global data structures, or changes to static variables. RUGRAT can handle some of these side effects when the testing specification details the side effect (e.g., which global variable should be modified).

The tester selects a program and test input to run. The tester specifies, through the test specification, the type of error conditions that should be simulated (e.g., memory allocation should fail) and target test points. In this instan-

[1] A basic block in DynamoRIO is a sequence of instructions that execute contiguously on the CPU.

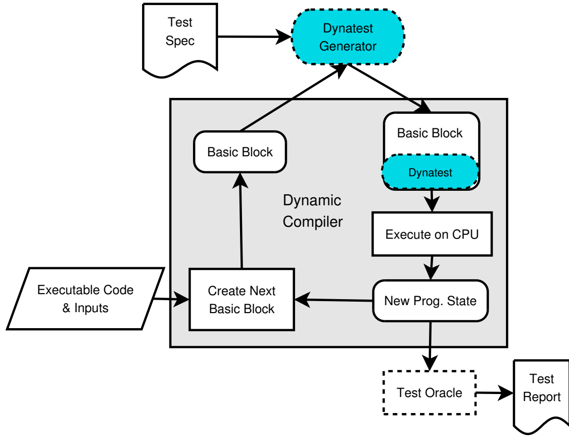

Figure 4: System Architecture. The dashed boxes are components of RUGRAT.

tiation, a testing point is the location of a function call whose return value should be modified. A tester can specify through the testing specification that all, or a subset of, calls to a particular function should be tested. The program is then executed under RUGRAT.

When a basic block is sent to the dynatest generator at execution time, the dynatest generator examines the basic block's instructions looking for particular function calls, as described in the test specification. When the specified call occurs in a basic block, the call is allowed to proceed normally. After the call returns, the dynatest generator adds a dynatest to the basic block. The dynatest is a set of instructions that change the result of the call. For a system call, the result is determined by the return value of the call and the value of the `errno` global variable. The basic block, along with the dynatest, are given back to the dynamic compiler and executed by the CPU. Because the return value of the call and the `errno` variable are modified, the program now executes as though an error has occurred and will execute any associated error handling code. The oracle checks that the error handling code has been executed. The test passes if the error handling code correctly handles the condition. The correctness of the error handling code can be determined by using the same methods a tester would use to verify the correctness of other program code.

Consider testing for an out of memory situation. The test specification indicates particular `malloc` callsites in the application code to target for testing. On success, `malloc` will return a pointer to the newly allocated memory. On failure, a `NULL` pointer will be returned and `errno` set to `ENOMEM`. During program execution under RUGRAT, a targeted `malloc` call is allowed to proceed normally. After `malloc` returns, the dynatest generator adds instructions to the basic block to change the return value of the `malloc` call to `NULL` and set `errno` to `ENOMEM`.

Table 1 lists some of the system calls that RUGRAT can target. The table shows the system call, the value returned on success, value returned on error, different values of `errno` that can be set and the error that is simulated by a dynatest. The table is not exhaustive; other calls can also be targeted. To apply RUGRAT to test application calls that could trigger error handling, the test specification would also include side effect information to be modified.

Call	Success	Error	`errno` vals	Environmental condition
`malloc`	non-NULL pointer	NULL	`ENOMEM`	out of memory
`socket`	int > 0	-1	`EACCESS`, and others	no socket available
`fork`	int ≥ 0	-1	`EAGAIN` `ENOMEM`	process copy failed
`read` `write`	int ≥ 0	-1	`EIO` and others	IO error
`open`	int > 0	-1	`EACCESS` and others	file opening failed
`fopen`	file pointer	NULL	`EACCESS` and others	file opening failed

Table 1: Example targeted system calls

Note that the RUGRAT approach sacrifices some simulation accuracy to reduce testing complexity. A more accurate simulation of the error condition could be obtained by modifying the particular system library call, such as `malloc`, to behave precisely as if an error had occurred. Modifying the library calls, however, would make the testing strategy much more complex to automate.

Note that RUGRAT offers more accuracy and benefits than simply changing branch predicates [27] to execute the error handling code. RUGRAT selectively fails a particular call by changing the return value and, possibly, modifying other variables as side effects of the function. Statements that are either control or data dependent on these values can thus execute correctly. Changing only branch predicates without modifying the associated return value or side-effect values may result in strange execution behavior, which would not aid testers. Additionally, the RUGRAT approach allows a tester to automatically locate callsites where error handling has been omitted.

4.2 Testing Function Pointer Protection Mechanisms

RUGRAT tests function pointer security mechanisms by generating a dynatest that simulates an attack. The tester begins by first selecting a program to be tested. Next, the tester creates the test specification that indicates the protected function pointers on which to test the protection mechanism. For this instantiation, testing points are callsites that use function pointers. Again, the tester can easily specify

that all such callsites should be tested or only a subset. The tester executes the program under RUGRAT. RUGRAT's runtime test generation strategy enables many test cases to be automatically and systematically generated for the protection mechanism without the need for a vulnerable program. The testing can occur either during initial development, or during maintenance phases to ensure any changes do not compromise the security mechanism.

When given a basic block from the dynamic compiler, the dynatest generator scans the instructions of the basic block looking for indirect function calls, which occur as instructions to copy the function address (value of the function pointer) into a register, *call_reg*, and then a call instruction with *call_reg* as the argument.[2] To create a dynatest that simulates an attack, the dynatest generator adds instructions to the basic block to place a new value into *call_reg* prior to any decryption. Thus, the dynatest simulates an attack that overwrote the function pointer while the pointer was still in memory. The program continues its execution under RUGRAT. If the security mechanism is functioning properly, the decryption should modify *call_reg* to have a bad address, causing the program to terminate when the function pointer is invoked. The oracle checks that the program terminates after a dynatest has been inserted. If the program terminates, the security mechanism passes the test since the mechanism prevented the attack from succeeding. If the security mechanism behaves incorrectly, then the program will jump to the address assigned by the dynatest generator. The attack has succeeded so RUGRAT reports that the protection mechanism has failed the test.

4.3 General Instantiations of RUGRAT

Other instantiations of RUGRAT are possible beyond testing function pointer security mechanisms and error handling code. For example, in previous work [5], we detailed an early version of RUGRAT that only tested security mechanisms that guard against stack smashing attacks.

New instantiations of RUGRAT require a new Dynatest generator and oracle. A new generator needs to know the possible testing points. For example, in the case of function pointer protection mechanisms, a testing point is any indirect function call. Next, the new generator needs to know how to create a dynatest, such as changing the value of a function at the testing point. Finally, the oracle needs to know the expected program behavior to determine pass/fail.

5 Evaluation Methodology

Questions and Measures. The specific research questions and measures for effectiveness vary slightly for the two testing tasks. In both tasks, the cost associated with RUGRAT is measured as the time needed to run the application under RUGRAT. The space requirements for RUGRAT are negligible as the memory needed for the dynatest generator is small. DynamoRIO's memory footprint is also small [7].

[2] An indirect call may be made with the function pointer value on the stack and not in a register. The security mechanisms, however, will usually modify the compiler to force all indirect calls to occur from registers only.

Program	Description	Source LOC	# funcs
008.espresso	Boolean function minimizer	9,844	363
130.li	Lisp interpreter	4,888	366
132.ijpeg	JPEG compressor	15,925	476
147.vortex	Object Oriented database	40,242	925
164.gzip	Gzip compression	5,604	106
space	ESA ADL Interpreter	6,230	136
boxed-sim	Box Simulator	8,994	199
lout	Document Typesetter	19,768	457

Table 2: Subjects of Analysis

The specific questions on effectiveness of RUGRAT for **testing error handling code** are:

RQ1 *How much error handling code actually requires specific external environment conditions to trigger execution?*

RQ2 *How effective is* RUGRAT *in generating tests to cover error handling code?*

RQ3 *How much of the error handling code covered by* RUGRAT *is covered without* RUGRAT*?*

RQ4 *How effective is* RUGRAT *in detecting failures in error handling code execution?*

The amount of error handling code (RQ 1) is measured as the static number of lines of code (LOC) that handles a particular error condition, if the condition is handled at all. Ideally, we would like to know how difficult it would be to test the targeted error handling without RUGRAT. However, it is difficult to quantitatively measure the difficulty in testing error handling code without RUGRAT as it varies with the error. Some errors, such as file open errors, could be straightforward to test by temporarily removing, or renaming, key files. Other error conditions, such as `fork` or `malloc` errors, are more difficult.

To obtain some sense of how well RUGRAT does, we measured the error handling code coverage by running the same program with the same inputs with and without RUGRAT. The coverage obtained with RUGRAT answers RQ 2, while comparing this coverage against the coverage without RUGRAT answers RQ 3. To answer RQ 4, we conducted a case study of the failure detection capability of RUGRAT with seeded faults in error handling code.

The specific questions on effectiveness of RUGRAT for **testing function pointer protection mechanisms** are:

RQ1 *How effective is* RUGRAT *at systematically generating test cases to cover the targeted function pointer protection mechanism?*

RQ2 *How effective is* RUGRAT *in exposing faults in the function pointer protection mechanism?*

The function pointer protection mechanisms operate by encrypting function pointers while in memory and decrypting the pointers just prior to their use. As such, each callsite where the program uses a function pointer is a possible test point for a generated dynatest. To answer RQ 1, we measured how well RUGRAT identifies indirect callsites as test points in the program at run time and how many distinct test cases can be automatically provided for the protection mechanism by using RUGRAT. We implemented our own function pointer protection mechanism similar to the PointGuard [9] mechanism. We also performed a case study of RUGRAT's capability to expose a fault in the function pointer protection mechanism (RQ 2).

Subjects. After examining many codes from well known benchmark suites, we chose a set of C programs that provide the most opportunities to evaluate our approach. We chose programs that use the most function pointers and/or contain the most error handling code. The set includes programs from the SPEC [22] and MiBench [13] suites, as well as the `space` application [11]. Table 2 lists the applications and their characteristics. Program inputs were either provided with the suite, or created by the authors to provide adequate coverage for our evaluation.

Methodology. All experiments were performed on a Red Hat Fedora Core 5 Linux machine with 1.0 GB of memory. Code coverage was measured by using `gcov`. Test pass/fail was determined as the basic blocks execute, and currently based on implementation of a check for the expected behavior based on the test specification.

Threats to Validity. Threats to validity derive primarily from the decision to use DynamoRIO as the underlying dynamic compiler which may introduce bias as we are reliant upon DynamoRIO's analysis. To guard against this, we used PIN [16], when appropriate, to verify some of DynamoRIO's analysis. We had to limit our subjects to programs that would execute correctly on DynamoRIO. Similarly, DynamoRIO limited us in some of the testing, which should be doable with a more robust dynamic compiler.

Another threat to validity concerns the function pointer protection mechanism we used in our study. We implemented a mechanism patterned after PointGuard [9]. Our mechanism works with the applications we used in our study, but is not intended as a general function pointer protection mechanism. However, the general applicability of our security mechanism is not important to the study.

Our study was limited to small-to-medium sized applications. Of more relevance is the function pointer usage and error handling code. Additional calls and more error handling code triggered by environmental conditions may be necessary to generalize the results.

Program/function	EH call sites	Total EH stmts	# rugrat covrd	# w/o rugrat covrd	% incr covrd
space/malloc	13/14	45	44	13	69
space/fopen	1	3	3	1	67
130.li/malloc	3	81	68	52	20
164.gzip/write	1	25	17	1	64
boxed-sim/malloc	71	0	–	–	–
lout/malloc†	27	87	56	19	43
lout/OpenFile	4	42	22	6	38

† lout has 8 `malloc` callsites with no EH code associated.

Table 3: RUGRAT effectiveness for error handling.

6 Results and Analysis

6.1 Program Coverage

Error Handling Code. Table 3 presents the results to answer questions RQ 1, RQ 2, and RQ 3 for error handling code. The first column shows the program and "trigger" function that, under certain uncommon environmental conditions, could trigger the error handling code under test. The set includes both system and application functions. Because the studied applications use few files, testing the error handling code for `fopen` could indeed be done by simply renaming files. However, this is not the case for an application where a large number of files are being accessed in different places in the code. Testing the error handling code for `malloc` without RUGRAT would require some way of allocating most of the available memory. This can be obtrusive as it may crash not only the program under test, but other applications. Furthermore, testing the error handling code at a particular callsite would require precise timing for when to allocate all the memory.

The second column shows the number of unique callsites that were covered by the program inputs and would result in calling the trigger function. The number of unique callsites was obtained independently of RUGRAT and DynamoRIO by writing a separate program that utilized PIN [16]. When RUGRAT was used for testing, RUGRAT identified all but one of these callsites and inserted dynatests accordingly. When testing `space`, RUGRAT tested 13 of the 14 `malloc` callsites covered by the test cases. RUGRAT failed to detect one callsite because DynamoRIO incurred a segmentation fault for unknown reasons when a dynatest was inserted at that callsite. In general, the number of unique callsites to target for testing error handling was relatively small because these applications abstracted away most of the relevant details into a small set of functions. To demonstrate that RUGRAT is able to target application calls as well as systems calls, RUGRAT was successfully used to exercise error handling code in `OpenFile` from `lout`.

The 'Total # EH stmts' column is a count of the number of error handling statements to be targeted for testing,

and also answers RQ 1. The next two columns provide the raw counts for statement coverage using RUGRAT and then without using RUGRAT on the same inputs, followed by the percent increase in error handling statement coverage achieved by using RUGRAT. In general, RUGRAT's coverage of error handling code was high. In some cases, the applications' error handling consisted of detecting the error, printing a message, and quitting. In others, such as `130.li` and `164.gzip`, the error handling was more complex. In these instances, the application either tried to recover from the error (see Figure 1 for how `130.li` tries to recover from an out of memory error) or attempted to exit more gracefully by freeing allocated buffers and closing relevant files. Upon closer examination of the error handling coverage missed by RUGRAT, the error handling code includes various options that were not covered by the test inputs.

Table 3 also shows RUGRAT's ability to locate some omission errors. In particular, the `lout` typesetting program has 27 callsites where `malloc` is invoked. 8 of those callsites have no associated error handling code. Similarly, the 71 callsites in `boxed-sim` do not have any error handling code. RUGRAT was able to automatically find these omission errors during testing.

The distribution of `space` [11] that we used includes over 13,000 test cases that were designed to maximize statement coverage. Without RUGRAT, the system calls in `space` always succeeded. The only error handling code that was exercised by any of the 13,000+ test cases was the IF statement that checked for a possible error. However, RUGRAT is able to force `space` down paths to handle the errors, and provides significantly increased coverage of the error handling code. The lack of error handling coverage by the large test suite of `space` demonstrates the difficulty in creating test cases to exercise error handling code.

`space` also includes extensive code to handle application errors during processing. Our examination of the coverage for `space` for all 13,000+ test cases without RUGRAT exposed 7 program points where code designed to handle an application error was not exercised by any test case. We inspected the code at each of the 7 missed points and determined that 5 points were impossible to reach. The remaining 2 may be possible but were not covered by any of the 13,000+ test cases. RUGRAT was able to cover the remaining 2 testing points by generating dynatests to fail the associated application calls.

Function Pointer Protection Mechanisms. Table 4 shows the data to address RQ 1 for testing function pointer security mechanisms. Each unique callsite that calls a function indirectly through a function pointer is a candidate for testing the function pointer protection mechanism. For each application, we used PIN [16] to count the number of unique callsites that utilize function pointers and are covered by the program inputs.

Program	# unique fptr callsites = # callsites tested
132.ijpeg	126
008.espresso	10
130.li	3
147.vortex	7
164.gzip	2
Total	148

Table 4: RUGRAT effectiveness for protection mechanism

RUGRAT was very successful, as it identified all possible function pointer callsites and generated dynatests to simulate attacks against each of the callsites. In the testing of our protection mechanism, each test passed as each attack was detected by the protection mechanism. Most importantly, RUGRAT was able to test all possible program points where the protection mechanism should be applied without requiring a vulnerable program.

In our study, we observed that some of the applications abstract away the details involved in invoking function pointers. In `130.li`, there are only 3 function pointer callsites. However, `130.li` invokes over 150 different functions through function pointers. These are primarily called at one callsite. This exposes a limitation of using DynamoRIO as the underlying dynamic compiler in RUGRAT. DynamoRIO's API allows the client module (dynatest generator) to analyze and modify particular basic blocks, but does not provide any contextual information while doing so. Thus, RUGRAT currently can target a particular *callsite*, but not a particular invocation through that callsite. Future versions of DynamoRIO could remove this limitation.

6.2 Failure Detection Case Studies

Error Handling Code. We performed a failure detection case study using the `space` [11] program to examine RUGRAT's effectiveness at revealing faults and to assess how much tester effort RUGRAT can save. The space program includes extensive error handling code to handle application errors and over 13,000 test cases that are designed to maximize statement coverage. To meet the goals of our failure detection study, we randomly selected 100 test cases.

We analyzed coverage information to identify "interesting" error handling points. For our case study, an error handling point consists of an IF statement that checks for an error and the associated IF body that handles the error. "Interesting" error points were selected: (1) the IF statement of the error handling point was covered but the associated IF body was not (i.e., the test inputs did not trigger the error handling code), and (2) the error handling code attempts to handle the error rather than printing a message and quitting. We identified 20 "interesting" points that met this criteria for the 100 test cases. `space` was run under RUGRAT us-

ing the 100 test cases as inputs. We verified that RUGRAT forced `space` to exercise the IF body for each error handling point. The outputs of `space` under these conditions were saved as the expected results.

Graduate students, but none of the authors, familiar with C manually seeded faults in the IF bodies of any of the 20 error handling points. This process created 34 faulty versions with one seeded fault per version. The seeded faults were distributed over 18 of the 20 interesting points. Each faulty version of `space` was run under RUGRAT, which forced `space` to exercise all of the error handling code.

We performed a study to estimate how many tests a tester may need to generate to cover all 20 error handling points used in our failure detection study if they did not use RUGRAT. We randomly chose an additional 100 test cases for `space` and computed their error handling coverage. We continued to randomly choose another 100 test cases until we had a pool of cases that covered all 20 of the error handling points. We found that we had to choose 1,600 additional test cases to provide the desired coverage. While this is just one case study, this provides an indication of how much testing effort could be saved by RUGRAT. In this case, a tester may have to create anywhere from an additional 20 to 1,600 test cases to cover the error handling points.

Next, the results of the faulty versions were compared against the expected results to detect failures. Out of the 34 seeded faults, RUGRAT detected 15 failures. RUGRAT did not expose failures for 19 of the seeded faults. Six of the non-exposed faults affected the return values of functions. The callers of these faulty functions only checked for a non-zero return (e.g., a return value of 1 or -1 caused the caller to perform the same action). Two of the undetected faults involved allocating too little memory for a data structure. `space` did not seem to notice the problem, possibly because the underlying implementation of `malloc` allocated slightly more memory than was requested. One fault caused callers of the faulty function to quit the program. Two faults were not detected for unknown reasons. Both of these faults initialized certain variables to different values, but the execution of `space` was unaffected on the test cases that covered these faults.

The remaining 8 undetected faults show an interesting feature of the `space` program. The faults went undetected by RUGRAT because the callers of the faulty functions simply repeated the error handling code. For example, if a function handled an error by the assignment `*x = 0` and then returned an error value, the caller would immediately set `*x = 0` as well. The 8 faults went undetected because `space` corrected for the faults by repeating the proper assignments. This makes `space` more robust with regard to errors, but possibly more difficult to maintain.

Function Pointer Protection Mechanisms. We also performed a case study on RUGRAT's effectiveness at exposing failures in function pointer mechanisms. Key generation is important for the correct functioning of these mechanisms. The encryption keys are randomly generated each time the application starts so the key generator must be fast. Due to the limited amount of vulnerable programs available for testing [10], problems in key generation may go unnoticed until the mechanism has been deployed. RUGRAT greatly increases the amount of testing that can be performed by dynamically creating dynatests.

We seeded a fault in our function pointer protection mechanism by breaking the key generator so that 5% of the generated keys were the identity key; far more often than would normally be expected. The broken mechanism was applied to `132.ijpeg` (which has the most opportunities for testing), which was then run under RUGRAT for all 126 indirect callsites. The identity key was generated 3 times and all 3 attacks succeeded, indicating that the protection mechanism was not functioning according to specification. Note that RUGRAT was able to detect the failure without using a vulnerable program.

6.3 Testing Costs

Timing data was collected by executing each application with no testing (and no RUGRAT), and executing again with RUGRAT performing testing. Due to space limitations, the timing data is not shown. In general, testing error handling code is fast because most programs terminate early when a system error occurred. Some tests for `130.li` and `space` took longer than the normal execution time. This is due to a combination of overhead from DynamoRIO and/or recovering from errors. When the callsite shown in Figure 1 was tested, `130.li` attempted to recover from the error. The recovery was successful and `130.li` continued execution and finished. The combination of the recovery and overhead of DynamoRIO (and RUGRAT) led to a longer execution time for this test case of 130.li. The overhead resulted in a slowdown factor of 1.2. The testing times for `space` took longer than normal because `space` is a short-lived program so the startup time for DynamoRIO becomes significant.

In all cases, the time needed to perform the testing of the function pointer protection mechanism using RUGRAT was far less than the time needed to run the application without any testing. Once an attack was made against a function pointer, the mechanism decrypted the function pointer value into an invalid address. When the function pointer was invoked, the application jumped to a bad address and was terminated. This happened quickly, so the time needed for testing was considerably shortened.

7 Conclusions and Future Directions

We presented and evaluated RUGRAT, a testing framework for generating runtime tests for the uncommon cases

of error handling and dynamic security mechanisms which are dependent on difficult-to-simulate environmental conditions. Our experimental results indicate that RUGRAT is cost effective in identifying testing points and creating test cases at those points that can significantly increase the coverage of rarely executed code. We also demonstrated that RUGRAT can expose failures that may otherwise go unnoticed due to lack of test cases. Thus, the tester does not need to create additioanl test cases, which may be difficult to craft, to cover the error handling code.

We believe that RUGRAT is useful in addressing other testing challenges. For example, RUGRAT could be used to provide additional test cases for certain security policy enforcement techniques that restrict resources that an application may access (see, for instance, [19]). The restrictions that these techniques enforce are usually dictated by a per-application security policy. RUGRAT could generate runtime tests that simulate that an application has been taken over by an attacker. Lastly, RUGRAT could be implemented for Java with an underlying dynamic compiler, with specifics for language differences and exception raising semantics carefully handled.

References

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

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. *Programming Language Design and Implementation*, 2000.

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

[4] B. Breech, M. Tegtmeyer, and L. Pollock. A comparison of online and dynamic impact analysis algorithms. *European Conf. on Software Maintenance and Reengineering*, 2005.

[5] B. Breech, M. Tegtmeyer, and L. Pollock. An attack simulator for systematically testing program-based security mechanisms. *International Symposium on Software Reliability Engineering*, 2006.

[6] P. Broadwell, N. Sastry, and J. Traupman. FIG: A prototype tool for online verification of recovery mechanisms. *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, 2002.

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

[8] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. *International Conference on Software Engineering*, 2006.

[9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. *USENIX Security Symposium*, 2003.

[10] 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.

[11] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[12] C. Fu, A. Milanova, B. Ryder, and D. G. Wonnacott. Robustness testing of java server applications. *IEEE Transactions on Software Engineering*, 31:292 – 310, 2005.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE Workshop on Workload Characterization*, 2001.

[14] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

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

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *Programming Language Design and Implementation*, 2005.

[17] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communication of the ACM*, 1990.

[18] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. *International Conference on Software Engineering*, 2005.

[19] N. Provos. Improving host security with system call policies. *USENIX Security Symposium*, 2003.

[20] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, 2000.

[21] O. Spinczyk, D. Lohmann, and M. Urban. Advances in AOP with AspectC++. *Software Methodologies Tools and Techniques*, 2005.

[22] Standard Performance Evaluation Corporation. SPEC benchmarks. http://www.spec.org.

[23] M. Süßkraut and C. Fetzer. Automatically finding and patching bad error handling. *European Dependable Computing Conference*, 2006.

[24] H. H. Thompson, J. A. Whittaker, and F. E. Mottay. Software security vulnerability testing in hostile environments. *Symposium on Applied Computing (SAC)*, 2002.

[25] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 2000.

[26] T. K. Tsai, M. Chen Hsueh, H. Zhao, Z. Kalbarczyk, and R. K. Iyer. Stress-based and path-based fault injection. *IEEE Transactions on Computers*, 48:1183, 1999.

[27] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. *International Conference on Software Engineering*, 2006.

[28] G. Zhu and A. Tyagi. Protection against indirect overflow attacks on pointers. *International Information Assurance Workshop*, 2004.