A General Framework for Dynamic Stub Injection

Maria Christakis
Microsoft Research
Redmond, WA, USA
mchri@microsoft.com

Patrick Emmisberger
Dept. of Computer Science
ETH Zurich, Switzerland
empatric@student.ethz.ch

Patrice Godefroid
Microsoft Research
Redmond, WA, USA
pg@microsoft.com

Peter Müller
Dept. of Computer Science
ETH Zurich, Switzerland
peter.mueller@inf.ethz.ch

***Abstract*—Stub testing is a standard technique to simulate the behavior of dependencies of an application under test such as the file system. Even though existing frameworks automate the actual stub injection, testers typically have to implement manually where and when to inject stubs, in addition to the stub behavior. This paper presents a novel framework that reduces this effort. The framework provides a domain specific language to describe stub injection strategies and stub behaviors via declarative rules, as well as a tool that automatically injects stubs dynamically into binary code according to these rules. Both the domain specific language and the injection are language independent, which enables the reuse of stubs and injection strategies across applications. We implemented this framework for both unmanaged (assembly) and managed (.NET) code and used it to perform fault injection for twelve large applications, which revealed numerous crashes and bugs in error handling code. We also show how to prioritize the analysis of test failures based on a comparison of the effectiveness of stub injection rules across applications.**

I. Introduction

A notorious difficulty in testing is dealing with operations whose behavior is not determined by the test inputs. Such operations include certain system calls (such as obtaining the current date), accessing external components (such as file systems and databases), and non-deterministic operations (such as generating random numbers). Controlling the behavior of such operations is necessary to ensure that tests behave predictably and to increase test coverage.

The standard solution to this problem is to replace these operations by stubs (or, similarly, fakes or mocks), whose behavior can be controlled by the tester [1]. Using stubs involves four main steps: (1) Deciding which functions to replace by stubs. For example, for a file API, testers might replace functions to read from a file, but not functions to manipulate file names. (2) For each function to be replaced, implementing one or more stubs. These stubs could for instance provide deterministic behavior or simulate erroneous behavior. (3) For each *call* to a replaced function, deciding whether to call the original function or a stub. For instance, it is useful to inject stubs for *all* calls to a non-deterministic function, but to simulate erroneous behavior only for *some* calls to prevent the application under test from terminating early and, thus, exploring only shallow paths. (4) For each chosen call, performing the actual stub injection.

Existing tools and frameworks automate some aspects of these steps, but are often too restrictive. For instance, aspect-oriented programming [2] provides an elegant way of expressing the behavior of stubs and injecting them via aspect weaving. However, this approach prescribes the use of an aspect-oriented programming language, which limits its applicability. Mocking frameworks [1], [3], [4], [5] provide similar functionality, but are typically language specific, which hampers reuse of stub code across programming languages. Fault injection frameworks like LFI [6] and Jaca [7] inject stubs that return error codes and set flags, but they do not allow testers to inject arbitrary behaviors, which is for instance necessary to simulate the behavior of external components. Several frameworks automate the actual stub injection [8], [9], [10], but do not provide support for the first three steps above. As a result, testing with stubs remains a mostly manual and time-consuming effort.

In this paper, we present a framework for stub injection that supports all four steps outlined above. At the core of the framework is a domain specific language (DSL) that allows testers to express the behavior of stubs as well as flexible strategies for injecting them. Our framework automatically implements the specified injection rules by dynamically altering the executable program.

A key virtue of this framework is that it is language independent: stubs are expressed in a very large subset of C that gets interpreted rather than compiled; they are then dynamically injected into binary code. Consequently, our framework does not require source code, which makes it widely applicable. Moreover, the injection rules can be reused across languages, for instance, to test different clients of the same external component.

Being able to reuse stub injection rules across applications does not only reduce effort; comparing their effectiveness across applications also allows one to optimize both testing and debugging. For instance, a rule that leads to failures in most applications is likely to produce behavior that cannot exist without the injection and, thus, should be reviewed to avoid spurious errors. In contrast, a rule that hardly ever leads to a failure is very likely to have found a true bug when it does produce a failure. Hence, failures produced by such rules should be given priority during the analysis of test results. We capture the effectiveness of rules in finding

true bugs by an indicator that adapts Engler et al.'s "bugs-as-deviant-behaviors" strategy [11] from the context of static analysis and specification mining to automatic testing.

We illustrate the power of our framework by testing error handling code in mature applications, such as Microsoft Notepad and Excel. We use our framework to inject faults in system calls and measure its effectiveness in finding bugs. For twelve mature applications, we were able to provoke over 170 crashes; we argue that they are due to real bugs.

This paper makes the following technical contributions:

1) A domain specific language that allows testers to specify flexible stub injection rules. The rules express the behavior of stubs as well as injection strategies.
2) A novel technique and implementation for dynamic stub injection into binaries via instrumentation of both unmanaged (assembly) and managed (.NET) code.
3) An extensive evaluation, which uses our framework to perform fault injection for twelve mature applications including Microsoft Excel, Word, and Notepad, and discovered over 170 crashes and numerous bugs.
4) A methodology for prioritizing the analysis of program failures based on a comparison of the effectiveness of stub injection rules across applications.

Outline. The next section gives an overview of our framework. Sect. III presents our DSL, while Sect. IV explains the code instrumentation. We present our evaluation and indicator for the effectiveness of stub injection rules in Sect. V. We discuss related work in Sect. VI and conclude in Sect. VII.

II. Overview

In this section, we present an example for a stub injection rule and explain how it is applied by our framework.

Various system functions take as argument a buffer that is being populated by the function, and flag an error if the buffer is too small. If the data to be copied into the buffer is not determined by the application under test (but for instance by the file system) then stubs can be used to provide the data or to simulate the case that the buffer is too small.

An example of such a function is `GetModuleFileNameW` from the Windows `kernel32.dll`. It retrieves the fully-qualified path for the file that contains a specified module, copies it into a specified buffer, and returns the number of copied characters (excluding the terminating null value). If the buffer is too small, the function truncates the string and sets the last error to `ERROR_INSUFFICIENT_BUFFER`. In this case, the returned value is equal to the buffer size, whereas it is smaller in the successful case.

Fig. 1 shows a stub injection rule that simulates the error case. The rule first specifies the function whose behavior is altered (line 1) and the names of its arguments (line 2) such that the stub code can refer to them. Line 3 specifies the injection strategy, which in this case directs our framework

```
1  rule KERNEL32.dll!GetModuleFileNameW(
2            hModule, lpFileName, nSize)
3    frequency every(2);
4    after {
5      if (nSize < 32767) {
6        SetLastError(
7            ERROR_INSUFFICIENT_BUFFER);
8        result = nSize;
9      }
10   }
```

Figure 1: An example stub injection rule.

to dynamically inject the stub for every other call to the function. Lines 4–10 specify the stub behavior. The **`after`** keyword (line 4) indicates that the stub will call the original function and perform additional operations *after* it has terminated. Here, these operations (lines 5–8) set the error code and adjust the return value. Both adjustments are performed only if the provided buffer is smaller than the maximum path length in Windows.

This rule illustrates several important aspects of our framework. First, stubs may have non-trivial behavior. For instance, they may refer to function arguments, which is crucial to specify accurate stubs and, thus, avoid false alarms. They may also change the state in arbitrary ways: Here, we set a global error flag, but it is also possible to modify complex data structures. Moreover, stubs may maintain their own data structures to implement complex behavior that spans multiple stub invocations. We discuss such an example in Sect. V. Second, since the stub is implemented in an interpreted subset of C, the rule is language independent. It can be used to test clients of this function written in various languages. Third, our DSL allows one to either replace or augment the behavior of the original function. Executing the original function is useful when only some aspects of its behavior need to be altered; it is then not necessary to re-implement all other aspects in the stub. In our example, the original function will still populate the buffer, and the stub only sets the error code and result value. Another rule for the same function could replace calls to the original function by code that populates the buffer itself.

To apply stub injection rules, we use a custom loader for Dynamic Link Libraries (DLLs) that instruments the loaded DLL to inject calls to our runtime component in the functions prescribed by the rules. Fig. 2 illustrates the workflow. To use our framework, the application under test is started through a dedicated launcher, which instructs the Windows loader to load our runtime component as the very first statically imported DLL. Upon loading, our runtime wraps the Windows loader with our own dedicated loader, which reads the stub injection rules from disk and instruments each subsequently loaded DLL (in memory). Once all DLLs are loaded and instrumented, the executor calls the `main` function of the application. Executing the application may

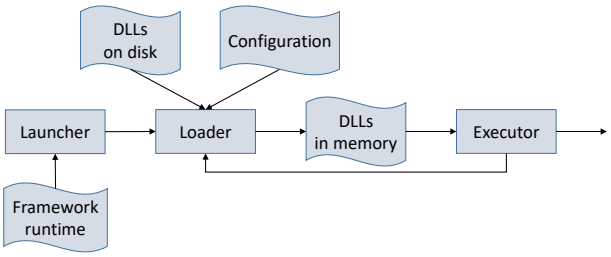

Figure 2: The workflow of stub injection.

trigger the loading of additional DLLs, which are handled analogously. For each call to an instrumented function, the executor dynamically evaluates the injection strategy and, if required, interprets the stub code.

Performing the instrumentation dynamically on the level of binaries has several advantages: First, our framework does not need source code, which makes it widely applicable. Second, we instrument the code that is actually deployed, which avoids potential issues with compilation and optimization. Third, it is not necessary to know in advance which DLLs will be loaded by an application. Fourth, our approach handles complex invocation schemes including recursion, reentrant calls, and calls through function pointers; these are difficult for approaches based on static rewriting.

III. A DSL for Dynamic Stub Injection

Our DSL for writing stub injection rules allows testers to control the functionality of our framework. The most interesting aspects of the grammar are presented in Fig. 3 and explained below. We omit some features (such as the ability to import headers) for brevity.

The *configuration* c of our framework consists of declarations of global and thread-local variables, and of a list of rule definitions. The variables allow stubs to maintain data structures that persist across several stub invocations, for instance, to compare the results of two subsequent calls (see Sect. V-C for an example). A rule r specifies a set of target functions, an injection strategy, and a stub.

The set of *target functions* includes only DLL-exported functions and is determined by two patterns for module and function names, m and f. A pattern is either an identifier (the name of the module or function), a wildcard ($\star$), which matches anything, or a regular expression. Regular expressions are useful to specify injection rules for a whole class of functions. For example, system functions whose name contains `create` typically return a handle to a newly created resource. Our DSL allows one to conveniently write one rule to inject stubs in all of them. After these patterns, the user may optionally specify the function parameters $\overline{p}$, which are used only to refer to them in the stub code.

The sets of target functions of different rules may overlap. For any given call, our framework applies the last rule in the configuration that matches the module and function name.

Configuration	c	::= **`global`**`(`$decls$`)` **`thread`**`(`$decls$`)` $\overline{r}$
Rule	r	::= **`rule`** m`!`f `[(`$\overline{p}$`)]` is sb
Strategy	is	::= $[dp]$ $[fr]$ $[rp]$ | **`none`**
Stub	sb	::= $[bf]$ $[af]$ $[vs]$
Depth	dp	::= **`depth`** (**`all`** | **`top`**)
Frequency	fr	::= **`frequency`** (**`every`**`(`int`)` | **`probability`**`(`$float$`)` | **`every_probability`**`(`int`,`$float$`)`)
Repeat	rp	::= **`repeat`** (**`infinity`** | int)
Before	bf	::= **`before`** {$code$}
After	af	::= **`after`** {$code$}
Variables	vs	::= **`call`**`(`$decls$`)`

Figure 3: The most interesting aspects of the grammar of our DSL. $\overline{x}$ denotes a possibly empty list of symbols x. We omit the specification of (optional) type information.

As we will discuss in Sect. IV, our framework uses the type information available in the debugging symbols to locate function entry points. If the debugging symbols are not available, testers may include both parameter and result types in the rules (not shown in the grammar).

Our framework instruments target functions with functionality that determines dynamically whether or not to inject a stub for any given call to the function. This decision is prescribed via the *injection strategy* is, which is defined as a combination of three filters; stubs are injected only for calls that pass all three. The depth filter dp filters calls based on the call stack. The option **`all`** retains all calls; **`top`** drops calls that are called (directly or transitively) by an instrumented function, for instance, recursive calls. The **`top`** option is useful to retain only calls at API boundaries, but drop calls from within the API or an injected stub.

The frequency filter fr and repeat filter rp select calls based on the call history. For the frequency filter, the option **`every`**`(n)` retains every n-th call to the same function and drops the others. For convenience, we support the option **`always`** as a shorthand for **`every`**`(1)`. The option **`probability`**`(`p`)` retains the call with probability p. We use **`never`** as a shorthand for **`probability`**`(0)`. Finally, **`every_probability`**`(`n`,`p`)` combines the **`every`** and **`probability`** options. It drops the call with probability $1 - p$. Out of the remaining calls to the same function, it retains every n-th. Finally, the repeat filter rp retains all calls with option **`infinity`** or only the first n calls to a function.

For convenience, any of the filters may be omitted. In this case, we choose the most permissive filter as default (that is, **`depth all`**, **`frequency always`**, and **`repeat infinity`**).

As an alternative to the three filters, the strategy is can be set to **`none`**. In this case, the framework ignores the call

completely. This option is useful to override earlier, general rules in the configuration for specific functions.

Our framework injects stubs for those instrumented calls that pass all three filters of the injection strategy. The behavior of a stub *sb* is described via three components. The **before** *action* specifies code that is executed upon entry to the stub. If this code executes a **return** statement, the stub terminates. Otherwise, our framework automatically executes the original function for which the stub was injected, followed by the code specified in the **after** *action*. This structure allows stubs to either *replace* or *augment* the original function. Replacing the original function is for instance useful when the stub simulates the behavior of an external component such as a database; augmenting lets the stub reuse some of the original function's behavior, as we illustrated in Sect. II.

The **before** and **after** actions are written in a large subset of C including **while**, **break**, **continue**, **return**, and **if-else** statements, but no **for**, **goto**, or **switch** statements, as well as no comma and ternary operators; an extension is straightforward. **after** actions may inspect the result of the original function via a predefined **result** variable. The value of this variable is returned when the action terminates; so **after** actions indicate their return value by assigning to **result**, as illustrated in Fig. 1.

When stub code is split over two actions, local variables declared in one are not in scope for the other. To work around this issue, the component *vs* allows the declaration of variables that are in scope of both actions of a stub.

IV. Instrumentation

This section explains how our framework instruments the application under test to implement the injection strategy and inject the stubs described by our DSL. We present solutions for both unmanaged (assembly) and managed (.NET) code and discuss some of the technical challenges involved.

A. Unmanaged code

Recall from Sect. II that our framework uses a custom loader to instrument imported DLLs according to the stub injection rules expressed in our DSL. To perform this instrumentation for unmanaged code, the custom loader creates a copy of each target function, that is, each function that matches a rule in the configuration for which the strategy is different from **none**. We use an extension of the Detours library [8] for this purpose. The loader then instruments the original version of the target function, while the copy is still available and can be called from the stub.

Fig. 4 shows the instrumentation of a target function `foo`. Its body is modified to start with a jump to a piece of assembly code that evaluates the injection strategy and, if required, executes the stub code. Modifying the code of the target function allows us to handle all calls, including calls

```
foo:
  jmp foo_stub
  ; more instructions

foo_stub:
  ; push information about foo
  call before_event
  cmp eax, 1
  je return

  call foo_original

  ; push information about foo
  call after_event
return:
  ret
```

Figure 4: Instrumentation of unmanaged function `foo`.

that involve function aliases, function pointers, or pointer arithmetic.

The assembly code (`foo_stub` in Fig. 4) is generated for each target function. It stores information about the target function `foo` and then calls the `before_event` function of our framework. This function determines, based on the previously stored information about `foo` and the injection strategy of the rule, whether a stub should be injected. If so, it interprets the code in the corresponding **before** action. The return value of `before_event` determines whether the **before** action replaces the original function or augments it. In the former case, `foo_stub` terminates and the application under test continues after the call to `foo`. In the latter case, `foo_stub` calls the copy of the target function we created earlier and then invokes the framework function `after_event` to execute the **after** action, if required.

The actual code is more complicated than shown in Fig. 4. For instance, it needs to pass along the result value of the **before** action, handle the declaration of variables that persist across the entire stub (component *vs* in Fig. 3), and manage registers, the stack pointer, and any data on the stack. We omit these technical details here and instead discuss our solutions to three more interesting technical challenges that arise in the context of unmanaged code.

Type information. When a rule targets multiple functions (through a regular expression or wildcard in the patterns), our framework needs to identify all DLL-exported functions to be instrumented. DLL export tables contain all exported symbols, but do not specify which symbols refer to functions and which to variables. Hence, they are not sufficient to determine where to apply the instrumentation.

Our framework provides two solutions to this problem. If the debugging symbols are available, our framework can precisely determine the exported functions of a DLL: an address in a DLL export table is a function entry point if and only if there exists a debugging symbol for a function starting at the same address.

If debugging symbols are not available, we determine the DLL-exported functions by consulting the DLL section table. This table contains information about which address ranges (or sections) contain executable code, read-write data, and read-only data. This approach works well in practice, although we have encountered a few DLLs in which the executable sections also contain read-only data, such that the instrumentation may corrupt data. In the absence of debugging symbols, our framework requires the stub injection rules to include parameter and result types of target functions in order to determine the memory layout of the stack frame.

Heap isolation. Our framework allocates memory to maintain internal data structures such as counters, and also permits stubs to allocate memory. This can lead to two problems: First, when the memory allocated by the framework and the memory allocated by the application are interspersed, a buffer overflow in the application under test is likely to corrupt the memory allocated by our framework. As a result, testers observe behavior that they would not see without instrumentation, which makes it difficult to reproduce and debug the error.

Second, the heap is a shared resource and a lock must be acquired for memory allocation. This lock is exposed to applications and may, in combination with locks that are introduced by our framework, lead to deadlocks. Both memory corruptions and deadlocks occurred several times during our experiments before we implemented the following solution.

We avoid both problems by providing internal rules that redirect all memory allocation operations performed by our framework to a separate, private heap. This private heap has its own lock, which avoids deadlocks between our framework and the application under test. We attempt to keep a gap between the two heaps. This gap shrinks as the application allocates more memory, and eventually there may be a need for several such memory blocks, each for the application and the framework. Nevertheless, there are considerably fewer places where the heaps of the application and of the framework are adjacent, which significantly reduces the risk of corrupting the memory of the framework.

Code modification. As shown in Fig. 4, our framework modifies the implementation of function `foo` by replacing the first n bytes with an unconditional jump instruction (for instance, $n = 6$ on 32bit Intel architectures). This modification will handle calls to `foo` correctly, but may lead to problems if the program tries to jump to an original instruction at address `foo`+*offset*, where $1 \leq \mathit{offset} < n$. Since these addresses no longer contain a valid instruction, the processor triggers an "invalid instruction" exception.

To avoid this problem, our framework provides an internal rule for the exception dispatcher that undoes the instrumentation of function `foo`, sets the injection strategy for `foo` to **`none`** to avoid the same problem in the rest of the execution, and resumes execution of the process at address `foo`+*offset*.

```
void foo(...) {
  bool inject = Executor.mayInject("foo");
  bool augment = false;

  if (inject) {
    augment = before_action();
  }
  if (augment) {
    try {
      // original function body of foo
    } finally {
      if (inject) {
        after_action();
      }
    }
  }
}
```

Figure 5: Instrumentation of managed function `foo`.

These steps effectively disable rules targeting `foo`.

Before implementing this solution, we encountered this problem multiple times on 64bit architectures, especially for functions that contain loops early in the function body such as some string functions.

B. Managed code

In the case of managed code, the loader of our framework is notified of two kinds of events in the .NET virtual machine. First, upon module loading, the loader retrieves all target functions in the module. Since .NET bytecode is typed, this step is simpler than for unmanaged code. Second, before each of these functions is JIT-compiled, the executor changes their in-memory bytecode representation through the .NET Profiling API as shown in Fig. 5.

The call to `Executor.mayInject` evaluates the injection strategy and yields whether a stub needs to be injected. If so, we execute the **`before`** action. Its result indicates whether the original function should be augmented or replaced. In the former case, we execute the original function body and the **`after`** action. The **`try-finally`** block ensures that the **`after`** action is considered for all normal and abrupt terminations of `foo`'s body.

As for unmanaged code, we simplified the presentation by omitting several technicalities such as the handling of result values. Note, however, that the technical challenges discussed for unmanaged code do not apply to managed code: the bytecode language is typed, which makes it easy to identify functions, the automatic memory management avoids problems with buffer overruns, and the structure of the bytecode makes it easy to modify the function body.

V. Experimental Evaluation

We evaluated our framework on several large applications. To have a simple and unambiguous definition of success and failure of a test, we focus on a specific kind of stub here, namely stubs that inject faults. In the following, we

describe the design of our experiments, present their results, and suggest a methodology for prioritizing the analysis of crashes based on an indicator that predicts the effectiveness of a stub injection rule in finding real bugs.

A. Design of experiments

Our experiments are defined by the applications under test, the test scenarios to be executed, and the stub injection. We describe these three aspects in the following.

Applications under test. We selected twelve widely used, commercial applications (see first column in Tab. I). We tested the latest version of all applications (at the time of writing) on a machine running Windows 10.

Test scenarios. Our fault injection will focus on system calls that request resources and access external resources such as the Windows registry. Many such calls are made during the start-up of an application; hence, we use the following simple steps as the test scenario for most applications: start the application, wait a fixed amount of time, and close it. Only the test scenario for Microsoft Notepad was slightly more involved and included selecting 'File'→'Open' and then 'Cancel' in the file browsing window.

Although our test scenarios are seemingly simple, they were sufficient for injecting faults in thousands of calls and detecting over 170 crashes (see Tab. I for more details).

Our framework allows one to express test scenarios in XML, combine them with different configurations for stub injection to obtain the concrete test cases, and then automatically execute these test cases.

Stub injection rules. As we discussed in Sect. III, defining a rule consists of determining the target functions, the injection strategy, and the stub behavior.

Target functions. For our experiments, we focus on two typical applications of stub testing: functions that request new resources and functions that access resources external to the application under test. To identify which functions to instrument, we executed the test scenario for two of the target applications (Microsoft Notepad and Notepad++, selected at random) and used our framework to record all function calls including their arguments and return values (up to a certain number of pointer dereferences).

To identify functions that request resources, we considered those functions among the recorded ones whose name contains the string `create`. We then consulted the documentation for a random subset of these functions and selected those that indeed request resources. We ended up selecting eleven functions with this approach. An example is `CreateFile` from `kernelbase.dll`, which creates or opens a file or I/O device. This function might fail, for instance, when access to the file or device is denied.

To identify functions that access external resources, we considered those functions among the recorded ones that returned different values for the same arguments across at least two calls, that is, functions whose result depends on some state. (We over-approximate the comparison of arguments, for instance, by dereferencing pointers a fixed number of times.) Again, we consulted the documentation for a random subset of these functions and selected those that access some external resource. In the end, we selected 29 functions with this approach. An example is function `GetEnvironmentVariableW` from `kernelbase.dll`, which retrieves the content of an environment variable. This content may change or the variables may get undefined between two calls to this function.

Injection strategy. We defined five different injection strategies, called NEVER, ALWAYS, EVERYOTHERCALL, ONCE, and FIFTYFIFTY. The NEVER strategy never injects a fault to a function call, and is used to check that our instrumentation alone does not cause crashes in any of the applications we consider. This was not the case in our experiments and, hence, the baseline strategy NEVER is not shown in Tab. I. The ALWAYS strategy injects a fault every time there is a call to a target function, the EVERYOTHERCALL strategy injects a fault every other time there is a call to the function, ONCE injects it only the first time there is a call, and FIFTYFIFTY injects a fault with a 50% probability.

Stub behavior. Fault injection requires rather simple stubs, which typically return a value and set an error code to specify what error occurred. We express these behaviors through **`after`** actions because for our target functions, the stub behavior is more faithful to the actual behavior of the target function if we execute the original function first (for instance, to fill a buffer) and then simulate that an error occurred. In total, we provided 39 **`after`** and two **`before`** actions for the 40 target functions we had selected.

When a system function fails in Windows, it typically returns an error code between 1 and 15,999. Depending on the error code, callers may handle each failure differently. Therefore, to avoid spurious errors, it is crucial that our stubs provide error codes which may actually occur. For each of the 30 functions that sets an error code, we consulted the Microsoft Developer Network (MSDN) and Stack Overflow for error codes that have occurred in real situations.

For (the remaining ten) functions that access external resources, we consulted the documentation to identify possible faults and corner cases. E.g., the stub for `GetSystemTime` calls the original function and then sets the date to Feb. 29 in a leap year, a corner case that has caused failures in Zune and Azure in the past. In many other cases, the stub returns `NULL` to indicate that the resource cannot be accessed.

In summary, we defined 200 rules (40 target functions x 5 injection strategies). For stubs that set an error code, this means that only one error code was tested per target function. We then applied each of these rules (in isolation) to every target application, for a total of 2,400 runs (200 rules x 12 applications).

B. Results: Detecting crashes

Tab. I reports the results of our experiments. The first column of the table shows the application, the second column the injection strategy, the third column the number of crashes found in each application per injection strategy, the fourth column shows the number of applied rules (that is, the number of target functions that are called at least once), the fifth column shows the number of instrumented calls, and the last column the number of these calls in which a stub was injected. Note that each table row shows the sum for the 40 test runs per injection strategy.

The data shows that for ten out of the twelve target applications (all but Microsoft OneNote Launcher and Microsoft Paint), our simple test scenario executes thousands of instrumented calls, indicating that we are not testing shallow execution paths. Note, however, that none of the target functions were called in Microsoft Paint.

Our evaluation focuses on Windows applications. However, our target functions are system functions that exist in similar forms on all platforms (e.g., to access the file and window system). Moreover, we selected a diverse set of classical desktop applications. Therefore, our findings demonstrate the effectiveness of the framework in a large software domain.

Comparing injection strategies. All injection strategies detected crashes, although the effectiveness differs between strategies. Across all target applications, the ALWAYS strategy detects the most crashes, a total of 62, followed by strategies FIFTYFIFTY and EVERYOTHERCALL, which detect 52 and 51 crashes, respectively. The ONCE strategy detects the fewest crashes, a total of 11, which is to be expected as it injects faults in the smallest number of calls.

Even though ALWAYS is overall the most effective strategy, it is useful to be able to define and apply different strategies. In the following we give two examples.

First, for the same program execution (that is, for the same set of instrumented calls), the ALWAYS strategy will inject more stubs than FIFTYFIFTY and EVERYOTHERCALL. Nevertheless, there are four applications in our experiments (PowerPoint, Publisher, Visio, and Notepad++), for which ALWAYS injects significantly *fewer* stubs. A likely explanation is that for these applications, FIFTYFIFTY and EVERYOTHERCALL explore deeper paths and, therefore, encounter more calls to the target function. Hence, these strategies have the potential to reveal deeper errors.

Second, the ONCE strategy produces by far the most crashes per 1,000 injected stubs (over 50, compared to at most 1.5 for the other three strategies). This suggests that the first call to a target function is particularly effective in detecting problems in error handling code. This strategy is therefore useful to reveal such problems in executions that are shorter and, thus, easier to debug than executions that crash at a later call to the target function.

Uniqueness of crashes. To determine whether the crashes found with different injection strategies overlap, we manually inspected all detected crashes in Notepad and Visio.

We inspected the eight crash sites (across injection strategies) for Notepad with a debugger and found that six of them are unique. There were two duplicate crash sites between strategies EVERYOTHERCALL and FIFTYFIFTY. One of them was caused by injecting faults in the exact same calls.

By inspecting the 23 crash sites for Visio, we found that there are two duplicates, one between strategies ALWAYS and FIFTYFIFTY, and one between EVERYOTHERCALL and FIFTYFIFTY. Both of these were due to the fact that faults were injected in the exact same calls by the two strategies.

This analysis provides another indication that it is indeed useful to apply different stub injection strategies because they are likely to produce different crashes. Different crash sites of course do not imply that the different crashes are caused by different bugs. Nevertheless, having few duplicate crash sites reduces the debugging effort considerably.

Rules triggering crashes. As we described in the experimental setup, we selected 40 target functions based on the test executions of two applications and then applied the resulting rules to all twelve applications. Across all applications, crashes were detected by applying 19 rules with the ALWAYS strategy, 16 with the EVERYOTHERCALL and FIFTYFIFTY strategies, and 6 with the ONCE strategy. Note that the functions targeted by the 19 rules of the ALWAYS strategy do not subsume those of the other strategies. For instance, a rule does not lead to any crashes with the ALWAYS strategy if the first call of the target function terminates the program normally; FIFTYFIFTY might not inject a stub for that call and then crash on a later call. Overall, injecting faults in 15 target functions (out of 40) did not lead to any crashes with any strategy, while injecting faults in the other 25 led to some crash with some strategy.

Performance overhead. The runtime and memory overhead of our framework depends on three factors: (1) the number of loaded DLLs that we instrument, (2) the injection strategy, which determines how often stubs are executed, and (3) how computationally expensive the stubs are. In our experiments, we instrumented only one function per test run, and all of our stubs were computationally cheap, so the overall performance overhead was negligible.

C. Results: From crashes to bugs

The crashes found with our framework are real bugs only if the stubs simulate behaviors that are realistic in practice. The fact that our stub injection rules are reusable across applications enables one to derive a measure for the effectiveness of a rule and use it both to identify unrealistic stub behaviors and to prioritize crashes, during the analysis of test results, that are more likely to occur in practice.

We propose the following heuristic for predicting the effectiveness in detecting real bugs of stub injection rules.

Application	Injection strategy	Crashes	Applied rules	Instrumented calls	Stubbed calls
Internet Explorer	ALWAYS	2	30	1,364	1,364
	EVERYOTHERCALL	3	30	1,537	775
	ONCE	0	31	3,363	31
	FIFTYFIFTY	2	29	1,434	708
Microsoft Excel	ALWAYS	4	7	1,643	1,643
	EVERYOTHERCALL	5	6	769	387
	ONCE	1	6	9,316	6
	FIFTYFIFTY	3	7	1,335	650
Microsoft Notepad	ALWAYS	1	25	38,499	38,499
	EVERYOTHERCALL	4	31	13,143	6,580
	ONCE	0	31	34,207	31
	FIFTYFIFTY	3	31	10,256	4,997
Microsoft OneNote Launcher	ALWAYS	1	13	106	106
	EVERYOTHERCALL	1	13	101	55
	ONCE	1	13	156	13
	FIFTYFIFTY	1	13	105	55
Microsoft Outlook	ALWAYS	8	19	2,686	2,686
	EVERYOTHERCALL	6	17	1,369	690
	ONCE	2	16	21,203	16
	FIFTYFIFTY	8	14	1,309	648
Microsoft Paint	ALWAYS	0	0	0	0
	EVERYOTHERCALL	0	0	0	0
	ONCE	0	0	0	0
	FIFTYFIFTY	0	0	0	0
Microsoft PowerPoint	ALWAYS	9	28	5,029	5,029
	EVERYOTHERCALL	7	27	2,950	1,481
	ONCE	2	28	31,677	28
	FIFTYFIFTY	7	28	16,074	7,827
Microsoft Publisher	ALWAYS	7	18	1,883	1,883
	EVERYOTHERCALL	4	20	13,758	6,885
	ONCE	1	20	21,522	20
	FIFTYFIFTY	5	23	5,177	2,532
Microsoft Visio	ALWAYS	10	26	1,646	1,646
	EVERYOTHERCALL	6	27	3,176	1,595
	ONCE	0	28	23,255	28
	FIFTYFIFTY	7	30	8,440	4,126
Microsoft Word	ALWAYS	14	29	17,906	17,906
	EVERYOTHERCALL	9	28	25,550	12,784
	ONCE	2	29	32,630	29
	FIFTYFIFTY	8	28	18,122	8,904
Notepad++	ALWAYS	0	8	149	149
	EVERYOTHERCALL	0	13	1,883	945
	ONCE	0	6	73	6
	FIFTYFIFTY	0	14	3,965	1,942
Skype for Business	ALWAYS	6	12	2,413	2,413
	EVERYOTHERCALL	6	12	2,126	1,066
	ONCE	2	11	4,290	11
	FIFTYFIFTY	8	19	2,976	1,446
Total		176	132	390,571	140,621

Table I: The results of our experiments.

For each target function f_i and each stub defined for f_i, we compute the following *real-bug indicator* across all injection strategies and applications:

$$rbi_i = 1 - \frac{\text{number of crashes by stubbing } f_i}{\text{number of stubbed calls to } f_i}$$

The indicator rbi_i predicts the likelihood of detecting real bugs when injecting the given stub in function f_i. A low value occurs if most stubbed calls lead to a crash. That is, most applications do not handle the stub behavior correctly, thus making it more likely that this behavior cannot occur in practice; otherwise, we would expect more callers to handle it correctly. Conversely, a high value occurs if relatively few stubbed calls lead to a crash. In this case, we conclude that it is possible to handle the injected stub behavior correctly, such that failing to do so is most likely a real bug.

We propose to calibrate the real-bug indicators for a set of rules by computing them for each combination of target function and stub across a set of applications and injection strategies. The results can then be used in three ways: First, rules with a low indicator should be reviewed and, if necessary, revised. Second, when the tests for an application produce more crashes than the developers can analyze, priority should be given to those that were produced by rules with a high real-bug indicator because they are more

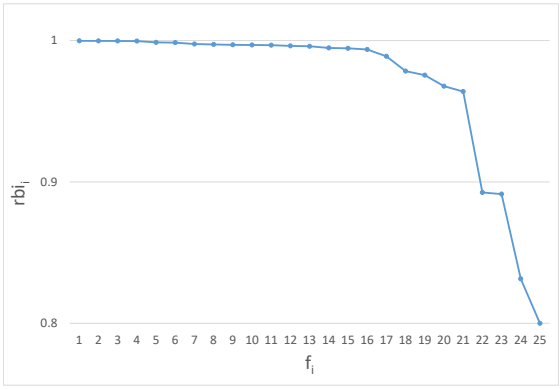

Figure 6: The real-bug indicators rbi_i for the 25 target functions f_i that lead to crashes.

likely to reveal real bugs. Third, when the available testing time for a new application (one that was not used during calibration) does not allow one to apply all available rules, testers should choose those that have a high indicator *and* have produced at least one crash during calibration (that is, the indicator is not trivially 100%, which could also be achieved by a stub that just calls the original function).

Indicators in our experiments. In our experiments, 25 out of 40 rules led to a crash for at least one application and injection strategy. Fig. 6 shows the real-bug indicator rbi_i on the y-axis for each of these 25 target functions f_i (in decreasing order). The indicators are high: between 99.98% and 80%. The indicator for the remaining 15 target functions is 100% since they did not lead to any crashes even though stubs were injected; we omitted those in Fig. 6.

The high indicators for our target functions are not surprising, given how diligently we defined the stubs. Consequently, stubbed calls to these functions are handled correctly at most call sites in our experiments. As an example, we present below a bug found in Excel, where fault injection in one of the target functions leads to a crash only 17 times out of 6,940. These "statistical outlier crashes" point to client code where a realistic fault is not handled correctly.

Overall, it is likely that all crashes found in our evaluation are caused by real bugs because: (1) the stubs comply with the documentation of the target functions, (2) we inspected tens of crashes and found no false alarms, and (3) the stubs that led to crashes have high real-bug indicators.

Computing the real-bug indicators *per application* can give an indication of how robust an application is against a set of rules. For example, Notepad is more robust against our set of rules than Excel, as it crashed fewer times even though there are many more stubbed calls across all strategies. In our experiments, Notepad++ was the most robust of all, since it never crashed for any strategy, even though we do inject faults. Therefore, there exists at least one target application that is robust to this subset of applied rules.

```
1  for (int c = 0; c < 2; c++) {
2    error = RegQueryValueExW(key, /* ... */,
3                             buffer, &buffer_size);
4    if (ERROR_SUCCESS == error) {
5      value_ptr = buffer;
6      break;
7    }
8    else if (ERROR_MORE_DATA == error) {
9      buffer.resize(buffer_size)
10   }
11 }
12 Assert(error == ERROR_SUCCESS);
```

Figure 7: Snippet of code from Microsoft Excel.

Example of a real bug found. We manually inspected many crashes found during our experiments, starting with those due to rules with high real-bug indicators rbi_i. Here, we present a sample bug found during this analysis.

Function `RegQueryValueExW` from `advapi32.dll` retrieves and stores the data associated with a specified registry key in a given buffer. If the buffer is not large enough to hold the data, which can be of arbitrary length, the function returns `ERROR_MORE_DATA`. In this case, the required buffer size is stored in a variable and the contents of the buffer are undefined. For our experiments, we wrote a rule that simulates this behavior (the rule with the sixth highest real-bug indicator in Fig. 6). When successful, `RegQueryValueExW` returns `ERROR_SUCCESS`.

Fig. 7 shows a snippet of code from Microsoft Excel containing a call to `RegQueryValueExW` (lines 2–3). When the function returns successfully, `value_ptr` is initialized to point to the contents of the buffer (lines 4–5). If, however, the function returns `ERROR_MORE_DATA`, the buffer is resized to fit the data (lines 8–9). This logic is retried twice in a loop (line 1), although it might take more than two tries to succeed. In other words, this code incorrectly assumes that the data associated with the registry key `key` will not be modified between the two calls to `RegQueryValueExW`.

The assertion on line 12 (which is a part of the actual implementation) makes this assumption even more explicit by aborting the application when the data has not been successfully retrieved after the loop. However, this assertion is not present in release code, which may thus have unpredictable behavior when more than two tries are required. Indeed, when testing Excel with the rule for `RegQueryValueExW` and the ALWAYS strategy, an access violation occurs when `value_ptr` is dereferenced much later in the code.

The rule is shown in Fig. 8. We use two thread-local variables to simulate that the buffer was not large enough to hold the data associated with a specific key for the first and second call. Note that our framework would also allow one to define a rule that tracks the number of calls per key (by maintaining a global map from keys to counter values).

```
thread last_key -> void*;
thread counter -> int;

rule ADVAPI32.dll!RegQueryValueExW(hKey,
           /* ... */, lpData, lpcbData)
  before {
    if (last_key != hKey) {
      last_key = hKey;
      counter = 0;
    } else {
      counter++;
    }
  }
  after {
    if (counter < 2) {
      (*lpcbData)++;
      SetLastError(ERROR_MORE_DATA);
      result = ERROR_MORE_DATA;
    }
  }
```

Figure 8: Stub injection rule for `RegQueryValueExW`.

VI. Related Work

As we discussed in the introduction, frameworks for writing stubs, fakes, or mocks [1], [3], [4], [5] typically target a particular programming language, which prohibits reuse of stub code across multiple languages. In contrast, our framework injects stubs by dynamically altering an executable program, independently of its source language. Aspect-oriented programming (AOP) [2] provides features like "advice" and "pointcut" that are similar to our **`before`** and **`after`** actions and to the use of regular expressions that match target functions in our rules, respectively. An AOP framework for assembly would have simplified the implementation of our approach, but to our knowledge, such a framework does not exist.

Frameworks that automate stub injection are usually tied to a particular runtime environment, like Detours [8] for unmanaged code, Moles [9] for .NET, and Javassist [10] for Java. We build upon and extend Detours and the .NET Profiling API to support automatic stub injection for both managed and unmanaged code. Moreover, our framework provides support for the process of writing stubs and injecting them in an application under test from start to finish.

Error handling code has been targeted by numerous fault injection techniques. In general, software-implemented fault injection [12] comprises three categories [13]. (1) Data error injection [14], [15], [16] performs low-level data corruption. (2) Techniques that inject code changes [17] either simulate faulty instruction decoding or common error patterns. (3) Interface error injection corrupts values passed between modules (e.g., a library and its client). Many prominent approaches in this category [18], [19], [7] focus on testing the robustness of the callee, e.g., a library. In contrast to these, the LFI framework [6] aims at finding bugs in clients due to imperfect documentation of libraries. Although our framework can also be used to perform fault injection, its purpose is much more general. Any code may be injected to augment or replace the functionality of any call.

Error handling code has also been targeted by various static analyses [20], [21], [22]. In contrast to that work, our purely dynamic approach does not report any false positives as long as the defined stub behaviors are realistic. Moreover, the detected crashes are reproducible and can be examined using standard debuggers.

A core originality of our approach is a domain specific language for expressing the behavior of stubs as well as injection strategies. In comparison to related work [2], [6], our DSL is more human-readable, flexible, and expressive. Although we do not infer the stubs, there is related work that infers what faults to inject as well as error specifications [6], [23], [24]. We compensate by supporting reusable stubs and avoid any spurious errors due to inference by pushing the responsibility to write realistic stubs to the user.

Our statistical methodology to identify crashes that are more likely to point to real bugs is inspired by Engler et al. [11] and follow-up work on specification mining and program repair [25]. Their work applies to classes of bugs broader than those considered in Sect. V. However, our more limited scope and dynamic approach allowed us to define a simple formula for determining the real-bug indicator rbi_i. This formula is also easy to compute automatically (unlike for instance LFI's estimate [6], which requires manual code inspection to determine false positives and negatives).

VII. Concluding Remarks

In this paper, we introduced a general framework for dynamic stub injection. Our framework provides an expressive DSL for writing stub injection rules and implements a novel technique for dynamically injecting the specified stubs into both managed and unmanaged binaries. In an extensive evaluation of several mature applications, we used our framework to perform fault injection, and discovered over 170 crashes. To prioritize the analysis of these crashes, we devised an indicator that predicts the effectiveness of stub injection rules in finding real bugs. We developed our approach for Windows and .NET, but similar instrumentation techniques can be implemented for other platforms.

A promising direction for future work is to develop debugging support on top of our framework. For instance, debugging a crash caused by a stub injection might benefit from inspecting other calls that handle the same stub correctly. Given that our framework can automatically produce large numbers of correct and faulty executions, one could also try to learn a (likely) program repair automatically. Moreover, since an application might fail long after a stubbed call, it would be useful to provide automatic support for determining the root cause of a crash, for instance, by comparing faulty and correct traces [26].

REFERENCES

[1] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: Unit testing with mock objects," 2000.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, ser. LNCS, vol. 1241. Springer, 1997, pp. 220–242.

[3] "JMock," http://www.jmock.org/.

[4] "EasyMock," http://easymock.org/.

[5] "Moq," https://github.com/moq/moq4.

[6] P. D. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *DSN*. IEEE Computer Society, 2009, pp. 379–388.

[7] E. Martins, C. M. F. Rubira, and N. G. M. Leme, "Jaca: A reflective fault injection tool based on patterns," in *DSN*. IEEE Computer Society, 2002, pp. 483–482.

[8] G. Hunt and D. Brubacher, "Detours: Binary interception of Win32 functions," in *Windows NT Symposium*. USENIX, 1999.

[9] N. Tillmann and J. de Halleux, "Moles: Tool-assisted environment isolation with closures," in *TOOLS*, ser. LNCS, vol. 6141. Springer, 2010, pp. 253–270.

[10] "Javassist," http://www.javassist.org.

[11] D. R. Engler, D. Y. Chen, and A. Chou, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *SOSP*. ACM, 2001, pp. 57–72.

[12] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *DSN*. IEEE Computer Society, 2000, pp. 417–426.

[13] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *EDCC*. IEEE Computer Society, 2012, pp. 162–172.

[14] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A flexible software-based fault and error injection system," *TC*, vol. 44, pp. 248–260, 1995.

[15] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *TSE*, vol. 24, pp. 125–136, 1998.

[16] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic object-oriented fault injection tool," in *DSN*. IEEE Computer Society, 2001, pp. 83–88.

[17] J. Durães and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *TSE*, vol. 32, pp. 849–867, 2006.

[18] M. Rodríguez, F. Salles, J. Fabre, and J. Arlat, "MAFALDA: Microkernel assessment by fault injection and design aid," in *EDCC*. IEEE Computer Society, 1999, pp. 143–160.

[19] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *TSE*, vol. 26, pp. 837–848, 2000.

[20] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *ESOP*, ser. LNCS, vol. 4421. Springer, 2007, pp. 520–535.

[21] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *PLDI*. ACM, 2009, pp. 270–280.

[22] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *OSDI*. USENIX, 2014, pp. 249–265.

[23] M. Acharya and T. Xie, "Mining API error-handling specifications from source code," in *FASE*, ser. LNCS, vol. 5503. Springer, 2009, pp. 370–384.

[24] S. Jana, Y. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *Security*. USENIX, 2016, pp. 345–362.

[25] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. T. Devanbu, "On the "naturalness" of buggy code," in *ICSE*. ACM, 2016, pp. 428–439.

[26] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *POPL*. ACM, 2003, pp. 97–105.