Testing program robustness against deviant behavior

Patrick Emmisberger

Master's Thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

12/16/2016

Supervised by:
Dr. Maria Christakis
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

With the increasing demands on software quality, testing has become an important part of the software development process. In particular, *unit testing*[21][3] is a widely applied technique for detecting bugs in a modular and repeatable way. However, dealing with external resources, whose behavior is not determined by test inputs, is a problem well-known for its difficulty. While different existing approaches (see Chapter 4) solve some aspects of this problem, they are often too restrictive. Initially motivated by interface-level fault injection, we present a general architecture for injecting stubs at the binary level. Built around a DSL (Domain Specific Language), our framework allows to alter the behavior of native and managed (i.e. .NET-based) functions in a flexible and language-independent way. We illustrate the potential of this framework in an evaluation that shows, how we can search for bugs in error handling code of large applications such as *Microsoft Excel or Word*.

Outline. Chapter 1 motivates this thesis, provides a summary of the necessary background and ends with a guided tour of our tool. Chapter 2 presents the general architecture and – in places – takes a deep dive into the technical details. In Chapter 3 we show the evaluation of our tool on a set of frequently used applications. We discuss related work in Chapter 4 and conclude the thesis in Chapter 5.

Acknowledgements

First and foremost, I would like to thank Dr. Maria Christakis for giving me the opportunity to work on this project, for the ideas and the valuable feedback. I continually value our collaborations and have enjoyed working on our projects immensely. Furthermore, I would also like to thank Prof. Dr. Peter Müller for allowing me to work on this thesis at the Chair of Programming Methodology. Additionally, I would like to thank Dr. Patrice Godefroid at Microsoft Research for all the insights and help with this project. Thanks go out to my friends and cofounders Aaron Richiger, Benjamin von Deschwanden and Martin Keller for their understanding during the development of this thesis and picking up the slack when I was unable to. Finally, I would like to thank my parents for their continued support, especially during my studies at ETH.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Approach . . . 2
- 1.3 Background . . . 2
  - 1.3.1 Executables, Dynamic Link Libraries and Modules . . . 2
  - 1.3.2 Component Object Model . . . 3
  - 1.3.3 The .NET Framework and the CLR . . . 3
- 1.4 A Guided Tour of Brute . . . 4

2 Architecture **9**
- 2.1 Overview . . . 9
- 2.2 Modifying Functions with Detours . . . 10
- 2.3 Running a Process with Brute . . . 11
- 2.4 Generic Hooks . . . 12
- 2.5 Dynamic Module Tracking . . . 15
- 2.6 Reentrancy and Safe Functions . . . 15
- 2.7 Metadata and Type Information . . . 16
- 2.8 Stack Frame Serialization . . . 17
- 2.9 Rule Specification Language . . . 18
  - 2.9.1 Rule Definitions . . . 19
  - 2.9.2 Fault Modeling Language . . . 21
- 2.10 Support for Managed Code . . . 24
  - 2.10.1 Approach . . . 24
  - 2.10.2 Implementation . . . 25
  - 2.10.3 Conclusion . . . 25
- 2.11 Tracing . . . 26
- 2.12 Challenges . . . 27
  - 2.12.1 Finding Function Entrypoints . . . 27
  - 2.12.2 Function Aliasing . . . 28
  - 2.12.3 Automatic Recovery . . . 29
  - 2.12.4 Shared Resources . . . 31

3 Evaluation **35**
- 3.1 Infrastructure . . . 35
- 3.2 Setup . . . 36
  - 3.2.1 Selection of Applications . . . 36
  - 3.2.2 Selection of Functions . . . 37

3.2.3 Selection of Fault Models . . . 38
3.2.4 Selection of Injection Strategies . . . 38
3.2.5 Function Grouping . . . 39
3.3 Results . . . 39
3.3.1 Fault Models . . . 39
3.3.2 Injection Strategies . . . 41
3.3.3 Crashes . . . 42
3.3.4 Example . . . 43

4 Related Work **47**

5 Conclusion **49**
5.1 Future Work . . . 49

Chapter 1

Introduction

1.1 Motivation

Robustness and reliability have always been important traits of a good software product. In today's software industry, where continuous deployment and automatic updates are widespread practices, the time that can be spent on testing a next release decreases steadily. An established practice used to cope with the increasing release cadence is automated unit testing. Only by automating the testing procedure itself, the release deadlines can be met and certain functional and qualitative standards maintained while the codebase is evolving.

At the same time, the breadth of software-driven devices and the variety of environmental configurations (e.g. cellular network vs. wireless vs. LAN, AC vs. battery) increases. This is in bold contrast to the typical development environment, which is very uniform, meaning test runs are always executed on the same machine and under the same network conditions etc. Automated test suites can, at least to a certain extent, mitigate this problem by running their tests in parallel on a number of different hardware configurations. However, two issues arise: First, the number of testable configurations will almost always be smaller than the number of configurations in production (this holds true especially for shrink-wrap software and mobile apps). Second, the test suite should, in addition to the *happy path* (i.e. given correct input, the program provides the correct output), also exercise failure and exceptional paths (e.g. network connection is disrupted). However, this is often not the case, as testing for such deviant behavior is difficult.

A pure software approach to emulating these exceptional states is a technique called *mocking*. Instead of directly invoking an API, it is accessed through an abstract interface. This allows replacing the implementation of the API during testing, thus simulating the exceptional states in a controlled way. The downside of this approach is the considerable amount of additional development effort. This is aggravated by the fact that the mocking infrastructure often cannot be shared between different projects, as the mocks are very project-specific.

In this thesis, we try to address some of these problems by providing a tool that enables the injection of deviant behavior in a controlled way, while keeping the effort below the demands of traditional mocking. The requirements for this framework are therefore set as follows:

1. Deviant behavior should be captured in a succinct and modular way.
2. The framework should be applicable to existing codebases without adaptation or the need to recompile the code.
3. When a model for a specific deviant behavior has been developed targeting one program, it should be transferable to a different program with no or very low effort.

1.2 Approach

During this thesis, we developed a framework called *Brute*, which allows injecting deviant behavior in a controlled and reproducible way. We do not rely on source code, but instrument an application directly on the assembly level. Therefore our solution is applicable to a wide range of applications. For our current implementation we chose to target Microsoft Windows applications only, however the ideas are applicable to other operating systems as well.

Many functions indicate for every call if the operation was successful or not. This happens by either returning an error code, setting an error/status flag or throwing an exception. The core idea is to provide the developer with a simple way of injecting these error states directly at the system API border. This allows us to test applications that were not specifically designed for testing and removes the need for defining an abstract interface of an API only for testing purposes.

Concretely, the developer can specify a *fault model* and an *injection strategy* on a per-function basis. The *fault model* describes the type of error that is injected, based on the arguments, the original return value and the original error code. *Fault models* can become arbitrarily complex, invoke other API functions or user-defined code and also keep state over multiple calls. They have the power to modify the arguments to a function before the call occurs and its result value after the call completed. The *injection strategy* specifies how often and under which circumstances a fault is injected. Together, the fault model and injection strategy define a *rule*, which can be applied to one or multiple functions that are being identified by their name (either explicitly or using regular expressions). Finally, there is a *launcher* that executes an unmodified program and alters the behavior of functions dynamically at runtime based on a set of *rules*. Currently, we support the instrumentation of DLL-exported functions (see Section 1.3.1) and managed functions (see Section 1.3.3).

Additionally, there is extensive tracing support, which allows listing function calls and the capturing of arguments, return values, error codes and parts of the heap. This is not only useful for developing the *fault models* in the first place, but also for debugging in general.

1.3 Background

The following sections provide information on the concepts and terminology required for the understanding of the remaining parts of this report. The esteemed reader already familiar with the topics at hand may skip the respective section(s).

1.3.1 Executables, Dynamic Link Libraries and Modules

On the Windows platform, executable code is stored in PE (Portable Executable) files. While PE files come in various subtypes, the two most common types are executables and DLLs (dynamic link libraries). Executables are applications with a single main entry point, while DLLs cannot be run on their own, but are referenced from executables and other libraries. The contents of a PE file (usually when loaded into memory) is referred to as a *module*.

Internally, PE files contain a header with multiple so-called directories. These are organized as tables and provide metadata for the loader. The important directories for our purposes are the *export*, *import* and *.NET metadata directory*. The *export directory* contains a list of functions and variables that this module exports and is normally only present in DLLs (although, because of their similar structure, executables can also export functions and be loaded like DLLs). The *import directory* contains a list of all external functions that this module needs. Modules that

are referenced in the *import directory* are loaded at the same time as the referencing module. Therefore, the modules are referred to as *statically referenced* modules, not to be mistaken for *statically linked* modules, which is a compile-time only concept. Should a module not be found, a message box is shown to the user and the loading process is aborted. Finally, The *.NET metadata directory* contains the type information for managed code. See Section 1.3.3 and 2.10 for more information.

To execute the code, the respective file is loaded into memory (usually using memory mapped files, shared between processes and with copy-on-write semantics), relocated and some entry code is executed. For executables, this is the familiar `main` function. DLLs also have an entry point named *DllMain* which is called after the DLL was loaded into a process, when the process creates or destroys a thread and before the process terminates. A module is separated into different *sections*, which are also described as part of the file header. Every section may have different alignment restrictions and protection flags (e.g. executable, read-only, read-write).

DLLs can be dynamically loaded using an API called `LoadLibrary`. In contrast to statically loaded DLLs, no error message is displayed when loading such a library fails and the caller has to deal with the missing dependency. Another API function provides access to the addresses of the functions and variables listed in the *export directory*.

Every executable depends at least on `ntdll.dll` and `kernel32.dll`. `ntdll.dll` contains the low-level interactions with the kernel and provides runtime services like the loader and memory management. `kernel32.dll` exports the common system APIs for memory management, thread management, I/O etc. User-code normally only interfaces with `kernel32.dll` directly, while `ntdll.dll` remains hidden in the background.

1.3.2 Component Object Model

The COM (Component Object Model) defines an ABI (application binary interface) as well as some common interfaces to build components that can be shared between different languages. While primarily used by Windows and its supporting frameworks to provide the more high-level APIs, there exist other implementations like Mozilla's XPCOM. COM is not bound to a specific programming language, but predominately used in C++ applications. COM exposes components in an object oriented manner, uses reference counting for managing object lifetimes and provides other runtime services like concurrency management for multi-threaded applications, remote procedure calls (DCOM) and component versioning.

For this project, we need to interface primarily with two APIs that are exposed through COM: the DIA (Debug Interface Access) API for reading debugging symbols and the CLR profiling API. Because COM not only manages the creation of objects, but also concurrency (e.g. dispatching a call from one thread to a different thread transparently) it needs to be initialized on a per-thread basis. This can be problematic because of incompatibilities on how the threading model is initialized (i.e. initialized differently by the application than *Brute* expects) or because COM is not yet initialized when we require to use it (see Section 2.7).

1.3.3 The .NET Framework and the CLR

.NET Framework is an umbrella term for multiple ECMA specifications (in particular the CLI (Common Language Infrastructure) and the programming language C#) as well as an implementation of these specifications referred to as the CLR (Common Language Runtime). The CLI defines a virtual machine (similar to the Java VM) and a portable low-level language called IL (Intermediate Language) (similar to Java Bytecode).

Multiple languages target the CLI, the prevalent one being C#, but many other languages, such as VB.NET and F#, exist. Code that runs on a CLI virtual machine is called *managed code*. In contrast to native (i.e. *unmanaged*) code, programs written for the CLI are compiled into IL instead of assembly and stored together with type information (the .NET metadata) as a PE file. While native PE files and managed PE contain the same header information, managed functions have to be just-in-time compiled into native assembly code before execution. This usually happens on demand, however precompilation for faster start-up times is possible. Aside from just-in-time compilation, the CLR provides other facilities like automatic memory management (garbage collection) and *P/Invoke*, a way to interact with native code.

IL is a low-level, strongly and statically typed, object oriented, stack-based language. Because of its simple structure (compared to the source languages), IL can be analyzed and rewritten with relatively little effort. We use the facility to rewrite IL code on-the-fly to provide the same type of instrumentation for managed functions as we do for unmanaged code. The CLR exposes multiple COM interfaces that allow unmanaged code to interact with the virtual machine itself. The most important interface for our purposes is the *profiling API*, which provides the callbacks for instrumenting and rewriting IL code.

1.4 A Guided Tour of Brute

In this section, we go through a short but complete example for defining a rule and applying it to an application. This provides an intuition of how *Brute* works, which becomes valuable when we later discuss the architecture and some of the implementation details.

For this guided tour, we will develop a fault model concerning LFN (long file name) support for the Windows API function `GetModuleFileNameW`. This function expects the handle of a module and a buffer and will, given the buffer is sufficiently large, copy the name of the source file into the buffer. It returns the number of characters copied excluding the zero-termination char (or zero if the call failed). If the return value is equal to the buffer size, the file name was truncated. In Windows XP, the error code is set to `ERROR_SUCCESS`, regardless of any truncation. For later versions of Windows, `ERROR_INSUFFICIENT_BUFFER` is returned as an error code. According to the MSDN[1] this function supports long file names and thus, buffer sizes up to 32KB could be required. While the Windows kernel supports long file names, many applications still assume the maximum length of a path to be `MAX_PATH` chars, which is a constant defined by the Windows headers as 260. The *fault model* we are developing will simulate file name lengths well beyond `MAX_PATH` to find callers that are not ready for long file names.

```
DWORD WINAPI GetModuleFileNameW(
  _In_opt_ HMODULE hModule,
  _Out_    LPWSTR  lpFilename,
  _In_     DWORD   nSize
);
```

Listing 1.1: Signature of `GetModuleFileNameW`

Based on the aforementioned information, we can create a *fault model* that simulates the truncation of the filename, should the buffer not be large enough. Listing 1.2 shows, how such a model can be encoded using *Brute*. First, we specify that our rule targets only the function `GetModuleFileNameW` in module `KERNEL32.dll` (line 1). Next, we provide names for the parameters, such that we can access them in the *after action*. The types of the arguments can be omitted, because the type information can be deduced from the debugging symbols automatically. We inject code after the function returns (lines 5-10) and check if the buffer size is larger than 32000, which is close to the real maximum length of 32768. If the buffer is smaller,

[1] `https://msdn.microsoft.com/en-us/library/windows/desktop/ms683197(v=vs.85).aspx`

we change the return value to the buffer size to indicate a truncation and set the error code to `ERROR_SUCCESS` as mandated by the MSDN documentation. The other rule attributes (lines 2-4) disable tracing, instruct *Brute* to apply the fault model to every second call and to do fault injection on recursive calls to `GetModuleFileNameW` as well as on surface level calls.

```
1  rule KERNEL32.dll!GetModuleFileNameW(hModule, lpFileName, nSize)
2    include recursive;
3    trace none;
4    frequency every_nth(2);
5    after {
6      if (nSize < 32000) {
7        last_win32_error = ERROR_SUCCESS;
8        result = nSize;
9      }
10   }
```

Listing 1.2: Rule for forcing LFN support with `GetModuleFileNameW`

We can save this rule in a file `tour.br` and then launch any application to which we want to apply this rule with *Brute*. In case we see any unexpected behavior or crashes while the application runs, it is most likely caused by insufficient support for long file names. To make sure that the *Brute* instrumentation is not responsible for the crash, we can change line 4 of Listing 1.2 to `frequency none`. This will enable all instrumentation except for injecting the fault. If the crash becomes irreproducible now, we can be sure that it was the result of improper usage of the `GetModuleFileNameW` function and was not related to the instrumentation.

We have found occurrences where this rule will lead to crashes in multiple applications. Depending on the version of Windows and the installed components, injecting this fault will prevent the common file dialog from opening and crash the process trying to show the dialog.

Changing the current time. A second example shows the capabilities of the framework that are not directly related to fault injection. We will create a rule that fixes the date to Feburary, 29^{th} 2016, which is a leap day. This special date has been responsible for problems across all different kind of programs. For example, it caused a service disruption at *Microsoft Azure* in 2012[14]. With this rule, a program or test suite can be executed under the assumption that it is running on a leap day to find any problems related specifically to that date.

To determine the time, the Windows API provides a pair of functions located in `kernel32.dll`. `GetLocalTime` returns the date and time in the current time zone, `GetSystemTime` in the UTC timezone. Both functions have the same signature and expect a pointer to a `SYSTEMTIME` struct, which is filled by the function. Contrary to many other functions, `GetLocalTime` and `GetSystemTime` do not return an error code, injecting a fault is therefore not possible for these functions. Passing a null pointer is prohibited as stated by the documentation and will result in an access violation. Listings 1.3 and 1.4 show the signature for both functions and the definition of the related `SYSTEMTIME` struct.

We can change the returned time and date using an *after action*. Listing 1.5 shows the rule which applies this change by simply assigning the respective members of the `SYSTEMTIME` struct. Since both functions share the same signature, we can target `GetLocalTime` and `GetSystemTime` using a single rule with a regular expression that matches both function names. Because the modification of the struct runs after the function call, we do not need to check if the given pointer is valid. When we inject this rule into Microsoft Notepad and print the date and time (using the

'Edit' → 'Time/Date' menu or the F5 function key) a timestamp with the current time but the date set to '02/29/2016' is inserted.

```
// Get time in local timezone.
void WINAPI GetLocalTime(
  _Out_ LPSYSTEMTIME lpSystemTime
);


// Get UTC time.
void WINAPI GetSystemTime(
  _Out_ LPSYSTEMTIME lpSystemTime
);
```

Listing 1.3: Signature of functions to get date/time.

```
struct SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} *LPSYSTEMTIME;
```

Listing 1.4: Definition of `SYSTEMTIME`.

```
1 rule *!'Get(System|Local)Time'(lpSystemTime)
2   include recursive;
3   frequency always;
4   after {
5     lpSystemTime->wYear = (WORD)2016;
6     lpSystemTime->wMonth = (WORD)2;
7     lpSystemTime->wDay = (WORD)29;
8   }
```

Listing 1.5: Change current date to a leap day.

External Functions. Next, we introduce how external functions can be used as part of *fault models*. To this end, we extend the previous fault model to display a dialog the first time it is called, which allows to select if the leap day should be injected. For this, we import the external function `MessageBoxA` from `user32.dll`, which can display a dialog with various different sets of buttons.

Listing 1.6 shows the full configuration file for the extended version of the fault model. We first declare a global variable `on_leapday`, which stores the answer that the user provides during the first call to one of the functions. The variable is initialized to zero by *Brute*, which indicates that the user has not yet made a choice. The lines 6-8 define different constants related to the `MessageBox` function, whose definitions can be found on the respective MSDN page[2]. On line 10 we import the function `MessageBoxA` (the ANSI version of the function) under the name `MessageBox` with the indicated parameter list.

Inside the *after action*, we first check if the user has already made a choice (line 16) and show the dialog if not (line 17). We use the previously defined constants to configure the dialog to show a question mark icon and a pair of buttons titled 'Yes' and 'No'. If the user selects 'Yes', we store 1 (line 18) in `on_leapday`, otherwise -1 (line 20). On line 24 we check the value of `on_leapday` to see if the user chose to change the date or not. Figure 1.1 shows a screenshot of Microsoft Notepad displaying the dialog.

We conclude the guided tour with this rule, which shows a bit more of the capabilities that *Brute* provides, in particular how our DSL can be used to define powerful and flexible rules.

[2] `https://msdn.microsoft.com/en-us/library/windows/desktop/ms645505(v=vs.85).aspx`

```
#include <win_error.br>
#include <win_types.br>

global on_leapday -> int;

#define MB_YESNO 0x00000004L
#define MB_ICONQUESTION 0x00000020L
#define IDYES 6

import user32.dll!MessageBoxA as MessageBox(PVOID hwnd, LPCSTR text, LPCSTR
    title, DWORD flags) -> int;

rule *!'Get(System|Local)Time'(lpSystemTime)
  include recursive;
  frequency always;
  after {
    if (on_leapday == 0) {
      if (MessageBox(nullptr, "Change date to February 29th?", "Leap Day",
          MB_YESNO | MB_ICONQUESTION) == IDYES) {
        on_leapday = 1;
      } else {
        on_leapday = -1;
      }
    }

    if (on_leapday > 0) {
      lpSystemTime->wYear = (WORD)2016;
      lpSystemTime->wMonth = (WORD)2;
      lpSystemTime->wDay = (WORD)29;
    }
  }
```

Listing 1.6: Extended leap day example with dialog.

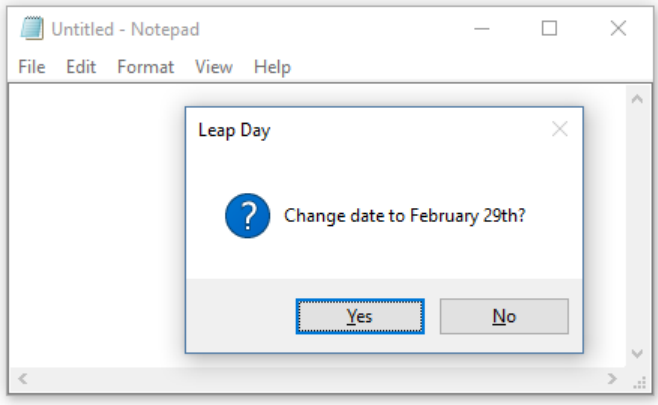

Figure 1.1: Microsoft Notepad showing dialog from *after action*.

Chapter 2

Architecture

This chapter contains a description of *Brute's* architecture. We start with a general overview of the components and their interactions. Continued by a discussion of the different components, we conclude with a section highlighting the more interesting challenges that we faced during development and the solutions that we chose. While many of the ideas are explained on a conceptual level, this chapter is the most technical and will at certain points also dive deep into the implementation and explain some of the low-level issues that we had to solve.

2.1 Overview

Figure 2.1 shows the global architecture of the tool. The three main components are the *Launcher*, the *Brute Runtime* and the *Rule Definitions*, that are stored in a configuration file (indicated by the *rules.br* file in the diagram). The gray parts of the diagram are only relevant for managed applications.

The *Launcher* is a command line application responsible for spawning a new process and injecting the *Brute Runtime* together with the *launch configuration* into the process. The *launch configuration* is a preprocessed version of the command line arguments and other configuration data required by the *Brute Runtime*. The *Brute Runtime* is a DLL that is loaded into the same address space as the target application and therefore has unrestricted access to the process' private memory. When loaded, the *Brute Runtime* finds the *launch configuration* and extracts

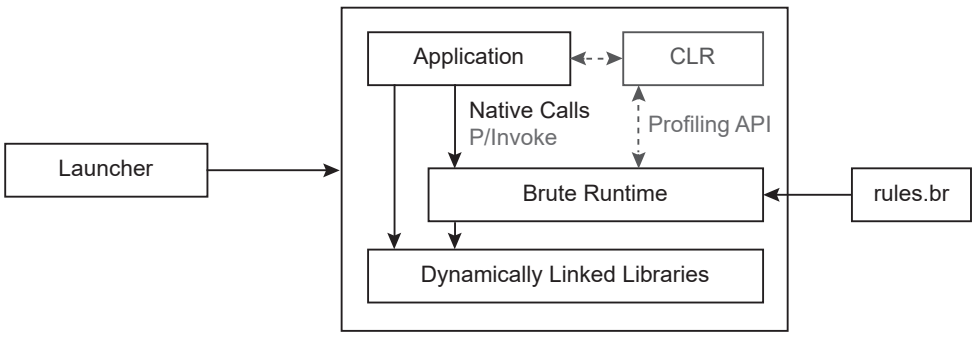

Figure 2.1: Architectural Overview

information about the location of the *Rule Definitions* on disk.

The *Rule Definitions* are a set of files that determine which functions to modify and how to alter their behavior. A detailed discussion of the rule specification language can be found in Section 2.9. The *Brute Runtime* modifies the functions that match one or multiple rules. For unmanaged functions this is done using *Detours*[9]. In the managed case, we use the *Profiling API* that is provided by the CLR.

2.2 Modifying Functions with Detours

Because we operate on the assembly level and do not have access to the source code of the client application, we require a way to modify or replace a function at runtime. This technique is commonly referred to as *hooking* and there exist multiple libraries that simplify this task.

The library we selected for this project is *Detours*[9]. Detours is developed by Microsoft and is being used a multitude of Microsoft's internal tools, as well as by many other companies. A restricted, free version can be obtained directly from Microsoft. A licensing model is available for commercial applications.

The hooking process involves three functions. First, the *target function*, which is the function that we want to modify, then a *hook function*, which contains the replacement for the target function and finally a *trampoline function*, which is generated dynamically and allows the hook function to call the original implementation.

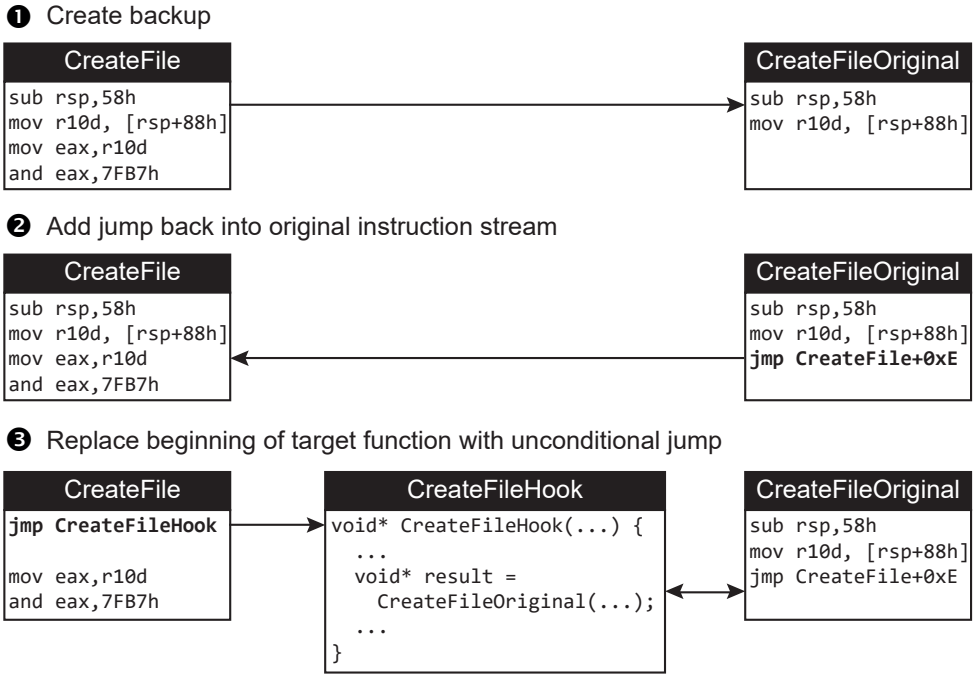

Figure 2.2: Hooking a function with Detours.

Figure 2.2 shows a visualization of the hooking process for the target function `CreateFile`. Detours first creates a backup copy of the instructions at the beginning of the target function. While doing this, it adjusts the instructions in case they use an addressing model that is relative

to the current instruction pointer (RIP addressing). The location where Detours created the backup becomes the trampoline function by appending an unconditional jump back to where the instruction stream continues in the original function.

Subsequently, Detours replaces the beginning of the target function with an unconditional jump to the replacement function. On architectures that use variable-length instruction encoding, in particular x86 and x64, the jump instruction might be longer than the first original instruction. Therefore, Detours needs to copy at least as many instructions, such that the total size of the copied instructions is larger than the size of the jump instruction.

Now the hooking operation is complete. When a caller jumps to the entry point of the `CreateFile` function, the execution will immediately be redirected to our replacement function. By keeping a backup of the overwritten instructions, the original implementation is still available at a different address. The replacement function is aware of this address and can invoke the original implementation at will.

Since the jump instruction transfers the execution directly from the target function to the hook functions, the signatures (i.e. the calling convention, number and types of parameters and return type) of these functions must match exactly. Otherwise, the call will lead to a stack imbalance and in most cases, shortly after, to a crash.

2.3 Running a Process with Brute

In this section we describe the main steps that are performed when a new process is started with *Brute* and how the *Brute Runtime* is initialized. Figure 2.3 shows a timeline of the process lifecycle and the most important events.

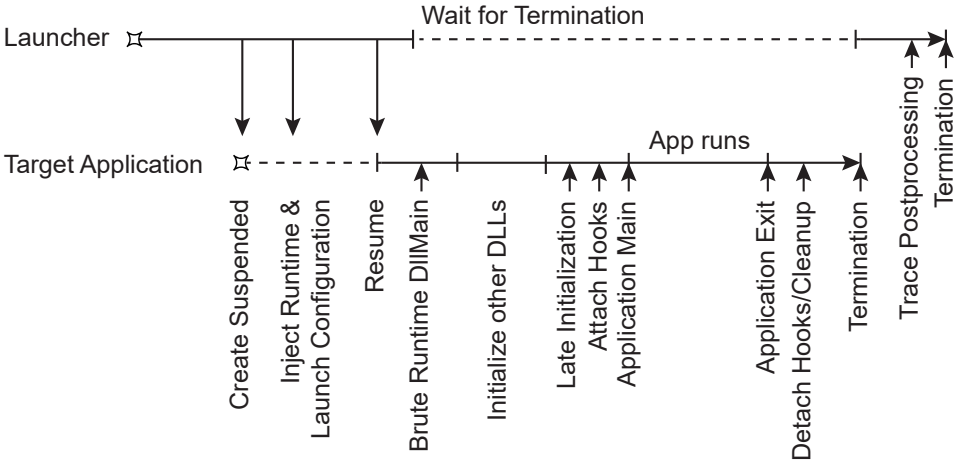

Figure 2.3: Timeline of the process lifecycle when running with Brute.

The whole procedure is started by running the *Launcher* process. It parses the command line arguments and determines which client application to start. Subsequently, it creates a new process in a suspended state. Using Detours[9], we manipulate the DLL import table to reference the *Brute Runtime* as the first statically imported DLL. When the newly started process is resumed, the Windows loader will therefore first load the *Brute Runtime* and call its `DllMain` function before any user code runs. However, we cannot fully initialize *Brute* here, because code that runs

inside of `DllMain` is subject to multiple restrictions. The most important one being, that we are not allowed to dynamically load other libraries using the `LoadLibrary` API. To circumvent this restriction, we hook the entry point of the application (i.e. the client's `main` function) and defer the remaining initialization until all other statically referenced DLLs are loaded.

When the Windows loader has finished initializing the other DLLs and jumps to the entry point, we regain control over the process and perform most of the initialization work. This includes parsing the rule definitions, generating the internal function repository (i.e. the table of all DLL-exported functions), loading the metadata for the currently loaded DLLs, such as symbol information and registering ourselves as a managed profiler for this process. This second initialization phase is called *Late Initialization*. Now that all necessary information for applying the rules is ready, we attach the hooks and finally call the original entry point.

At this point, the application runs normally and we get callbacks for all hooked functions. If the application is managed (or hosts a CLR in any other way, like using a managed COM server) we will also get callbacks from the CLR. When DLLs are loaded or unloaded dynamically, we update the internal function repository and attach hooks accordingly as described in Section 2.5.

Just after the application exits (either by returning from the `main` function or calling `ExitProcess`) the `DllMain` function of the *Brute Runtime* DLLs is called again, this time indicating that the process is terminating. During this call, we unhook all functions, flush all remaining tracing data to disk and finally terminate the process.

After the launcher detects that the target application has exited, it performs some post-processing of the tracing data and then exits as well. More information about the tracing feature can be found in Section 2.11.

2.4 Generic Hooks

For reasons described in Section 2.2, the signature of any target function must be known in order to hook it. This is problematic for our purposes, because we want to hook functions on a large scale, without needing the full signature or any signature at all. To this end, we developed an extension to Detours that allows hooking without any knowledge about a function, except for its start address. This section is fairly technical and while the most important concepts are explained to the extent necessary to understand the ideas, basic knowledge about processor architectures is assumed.

In order to understand how generic hooks work, we first need to discuss how function arguments are passed from caller to callee (and vice-versa for the return value) on the assembly level. We will also take a look at the role of calling conventions. When compiling from a higher-level language to assembly, the compiler needs to map function arguments to registers or areas of memory, since the assembly instruction for calling a function does not handle function parameters. In order to pass the arguments correctly, the caller and callee must agree on where arguments are stored, even if the two functions are written in different high-level languages or compiled using different compilers. To accomplish this, there are different calling conventions, which define how the arguments are mapped to processor resources (like registers or memory). A discussion of the different calling conventions can be found in Section 2.8. To allow for recursion and avoid concurrency issues, argument values are usually placed on the stack or in registers, if the calling convention allows this and the registers are large enough. Since we need a way of intercepting a function call independent of the calling convention, we can only rely on properties that all calling conventions have in common, which are the following:

1. The stack grows from top to bottom (i.e. from higher addresses to lower addresses).
2. The stack pointer points to the last value on the stack.

3. The stack below (i.e. lower addresses) the stack pointer can be used freely by the callee.
4. Upon entering the callee, the return address (i.e. the address of the next instruction to execute when the function returns) is the last value on the stack.

As a side note, these statements only hold true for x86 and x64. Other processor architectures like ARM use completely different calling conventions. While an implementation of generic hooks would still be possible, the solution presented here only works for these two Intel-based architectures.

Figure 2.4 visualizes the stack directly after a function is entered. For the remainder of this section, the example will always target x86 but is still directly applicable to x64 by extending addresses to 64-bit and switching register names with their 64-bit counterpart (e.g. ESP becomes RSP). ESP is the processor's stack pointer register (i.e. a pointer to the last value on the stack). EAX, EBX, ECX and EDX are general purpose registers.

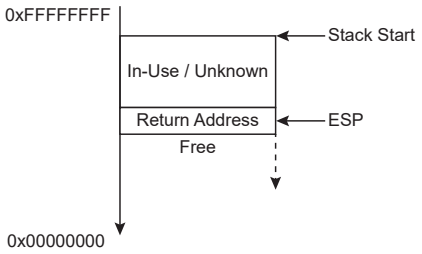

Figure 2.4: Stack layout upon entering a function.

We can execute arbitrary code before and after the call to the original function, under the condition that we preserve all the relevant state. This includes at least all registers, as well as the stack pointer and the data that is above the current stack top (i.e. addresses higher than the stack pointer). While all these conditions must hold when we call the original implementation (i.e. the trampoline function that Detours generated), we can temporarily violate these restrictions as long as we reestablish them again before calling the original function. This allows us to perform a callback before and after the original function runs. Note that without the signature, it is not possible to skip the call to the original function, since some calling conventions require that the callee removes the arguments from the stack and we do not have the necessary information to do this.

From this, we can derive the pseudo code for a generic hook as seen in Figure 2.5. We first push the (volatile) registers onto the stack, call `beforeCallback` and restore the register values from the stack after we return from the callback (lines 1-3). Then, we call the original implementation (line 4), followed by the same procedure for `afterCallback` (lines 5-7). Since we need precise control over the registers and stack, the implementation of this stub must be written in assembly. We already provide implementations for x86 and x64. As mentioned earlier, an implementation for ARM is possible, but was not in the scope of this project.

```
1: push registers
2: call beforeCallback
3: pop registers

4: call originalImplementation

5: push registers
6: call afterCallback
7: pop registers

8: ret
```

Figure 2.5: Pseudo-code for generic hook.

Figure 2.6 shows the layout of the stack during the call to `beforeCallback` (line 2). *Ret. Addr. Callback* represents the address, at which the execution resumes after the callback completes and is the address of line 3 and 7, respectively. Inside the callback, we can calculate the original ESP and therefore, access the return address and preceding data on the stack. Should the user provide a signature for the function (either by providing debugging symbols or specifying the

signature as part of the rule), we are still able to read and modify the arguments and the return value.

While the implementation follows the structure outlined in Algorithm 2.5, there are a few additional caveats that need to be addressed. First, we cannot call the original implementation directly (line 4). Doing this would push an additional return address onto the stack, with the effect that all accesses to the stack would be off by the size of an address (i.e. 4 bytes on x86 or 8 bytes on x64). However, we can replace the return address on the stack with the address of line 5 in the algorithm and restore the original return address as part of the `afterCallback`. While this change is visible to the callee, it is not problematic, because the callee should not make any assumptions about the address of a caller[1]. An issue that arises as a direct consequence of this is that we need to store the original return address for the duration of the call. Since storing the address on the stack is not an option, we use a shadow stack in thread-local storage.

Second, preserving the stack and registers is not sufficient in order to make the *before* and *after callbacks* completely transparent to the user code. In general, we need to preserve the whole state that is shared between the *Brute Runtime* and the user code. While we try to keep the amount of overlap as minimal as possible (see Section 2.12.4 for more information) and stack/registers cover most of this overlap, there are other instances of shared data. An important example is the error code of the last API call (accessed using `GetLastError()`). Windows maintains a thread-local variable, that is used by most of the Windows APIs to communicate the type of error that occurred during the last call. Should we use these APIs during the callbacks (the after callback being the more interesting one here), we inadvertently overwrite the error code that was returned by the original implementation with the error code that happens to be returned by the last API call in the callback.

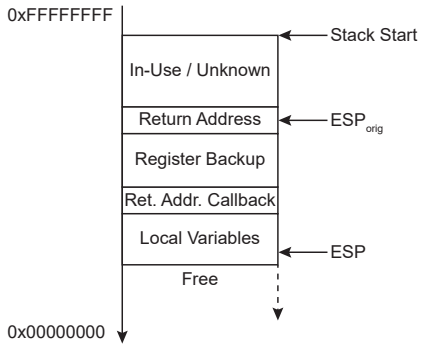

Figure 2.6: Stack layout during a callback.

There are other architecture-specific details like the stack alignment restriction on x64 or the FPU state on x86. These need to be addressed in the implementation, but do not contribute to the understanding of the basic idea, which is why we skip them here. In general, our solution implements a mechanism that is similar to a lightweight context switch and stores the ephemeral call data on a shadow stack in thread-local storage. As a result, we can preserve the shared state (in particular the stack) during the *before* and *after callbacks*, making our instrumentation transparent to the user code, with the exception of the return address and the timing. Because of this, we can instrument functions regardless of their signature, while maintaining the ability to read and manipulate arguments and the return value, should a signature be provided.

[1] There exists code that uses the return address to ensure that a function is only called from specific modules or functions. However, this check is not reliable in the presence of tail call optimization and there are multiple ways to circumvent this check, should one invest the necessary amount of criminal energy. Therefore, we do not handle this type of function.

2.5 Dynamic Module Tracking

Aside from statically referencing DLLs in the import table, DLLs can also be delay-loaded or dynamically loaded. In the case of delay-loaded DLLs, the compiler does not add a reference in the DLL import table, but instead generates a thunk for every referenced function, that dynamically loads the DLLs when it is first called. After the DLLs are loaded, all thunks that target the same DLL are forwarded to the actual implementation. In the fully dynamic case, the DLL is loaded using the `LoadLibrary` API which takes the path to a DLL on disk. By passing the resulting module handle and a function name to the `GetProcAddress` function, the address of a function in the DLL can be determined.

Since we can only hook functions that are present in memory, we need to keep track of all currently loaded DLLs, dynamically update the function repository when modules are loaded or unloaded and hook functions before the user has a chance to get the function address. We do this by hooking three internal functions (`LdrLoadDll`, `LdrResolveDelayLoadedAPI`, `LdrUnloadDll`) that the Windows loader uses when dynamically- or delay-loading/unloading a DLL. We determined these functions by hooking lower-level functions for mapping the DLLs into the address space, for which we knew that they were used to load DLLs and then analyzing the stack traces. After a module is loaded into or unloaded from the process, we enumerate all modules and calculate which modules were added or removed. We then update the function repository and attach hooks to the functions accordingly.

We use this approach because it is much simpler than trying to track the loading / unloading of every module separately. First, loading one DLL can result in multiple DLLs being loaded into the process (the requested DLLs and all transitive dependencies). Second, the loading process is parallelized internally and the corresponding functions are all undocumented and not DLL-exported. One drawback of this approach is, that modules are unloaded before they can be successfully unhooked (i.e. the unload event is only triggered when the module has already been unloaded). Naturally, this prevents Detours from successfully unhooking the functions. Depending on how Detours handles this issue, it could result in a memory leak.

2.6 Reentrancy and Safe Functions

Because we allow hooking of a wide range of functions, including low-level functions that may be used by the instrumentation itself, we need to deal with reentrancy in two different contexts.

First, we want to allow the user to limit the effect of a rule only to the outer most recursive call or to the surface call of an API. The later meaning, that when a hooked API uses other hooked functions, the rules only apply to the outermost function call (even if the call is not recursive). The reasoning behind this is, that we might introduce false positives when injecting faults into lower-level calls. The library developer could have used some source-level information about the behavior of a function in the same module, which is then invalidated by a rule. As an example, take a function whose documentation specifies more error codes than the function can actually return. This is always allowed and can prevent that a change breaks the API contract, thus making the API more stable. However, for intra-module function calls, the developer might only handle errors that the function can actually return and just expand the error handling code as needed. By injecting errors that the documentation specified, but are never actually returned, we might end up introducing false positives. In theory, we would like to only apply rules when a call crosses API boundaries. However, in practice, determining these boundaries is impossible, since modularity is only a design concept, but does not necessarily manifest itself in the compiled code.

Second, the instrumentation must handle the case, where functions used in the *before* and *after callbacks* have rules applied to them. If this occurs, we also do not want to apply the rule behavior, because it could lead to infinite recursion or compromise the integrity of the fault-injection logic itself.

We address these issues in two ways: By detecting reentrancy dynamically and where this is not possible, preventing it statically. The dynamic detection covers most of the functions and is implemented by simply keeping a flag in thread-local storage, indicating whether the thread is currently inside an instrumentation function. This flag is internally referred to as the *reentrancy gate*. Inside the *before callback*, we first try to acquire the *reentrancy gate* (i.e. check if the flag is currently not set and set it if this is the case). If we fail to acquire the *reentrancy gate*, we simply jump to the original implementation of the function, the *after callback* is never invoked for these calls. If we manage to acquire the *reentrancy gate*, we proceed by setting up the *after callback* and perform any other actions that are specified by the rule for this function. Before calling the original implementation, we release the *reentrancy gate* and reacquire it in the *after callback*. Reacquiring the *reentrancy gate* should always succeed, because of thread-locality. Should the user specify that reentrant calls should not be instrumented, we do not release the *reentrancy gate* for the duration of the call to the original implementation. With this simple approach, we can dynamically detect reentrant calls into the *before* and *after callbacks*, allowing the safe usage of any function inside these callbacks, even for hooked functions.

While the dynamic reentrancy detection works for most functions and prevents us from requiring static knowledge of all the functions that are used inside of callbacks, there is a subset for which we need a different approach. Concretely, these are the functions required for setting up the *reentrancy gate* itself (i.e. the functions for accessing thread-local storage) and the functions that need to run outside of the *reentrancy gate*. Internally, we refer to these as *safe functions*, because they are always safe to call. Instead of statically binding to these functions, we maintain a table of function pointers that is automatically updated by the instrumentation, whenever we hook or unhook a function inside the table. In the unhooked state, the table points to the normal function address. While a function is hooked, the table points to the trampoline function (i.e. the original implementation) instead. Therefore, whenever we call a function through this table, we are sure to always call the original implementation, regardless of any rules applied to them.

2.7 Metadata and Type Information

While *Brute* does not require type information for basic tracing and fault injection, it is still required for the more advanced features like argument value tracing or fault injection based on function arguments. *Brute* supports two different ways of specifying type information, either by listing the full signature of a function inside the configuration file or by importing types from debugging symbol information, emitted by the C/C++ compiler. While the code architecture is easily extensible and would allow for different formats of debugging symbols, only the *PDB (Program Database / CodeView)* format (used by the Microsoft Visual C++ compiler) is currently supported. For managed code, no additional type information is required, because it is already embedded in the IL code and (managed) assembly metadata.

Brute contains an implementation of a type system that is closely based on C's type system (with some additions from C++). The type system supports the usual fundamental types like signed and unsigned integers of different widths and floating point numbers, structs, unions, function signatures, type modifiers like const and volatile, pointers and reference types. Types can be created by lifting static C++ types to our type system (this is useful exposing internal structs and functions of the *Brute* API to the user), by importing debugging symbols or parsing

them as part of the configuration file. Should a type declared by the debugging symbols not be representable, it will be replaced by an opaque type called *unknown*.

The user can specify a symbol server, from which debugging symbols are automatically downloaded on demand. Debugging symbols will only be imported once for every module and then cached in a *metadata cache*. While this speeds up the metadata loading procedure, it has additional benefits. First, loading the debugging symbols from a symbol server requires HTTP(S) communication, which not only loads many additional dependencies into the process, but also relies on the network, thus having a big impact on the client application. Furthermore, because the PDB format is proprietary, accessing PDB files requires making use of the DIA (Debug Interface Access) API, which is a COM-based API. The problem lies in the fact that COM needs to be initialized on a per-thread basis, which means that accessing the DIA API could lead to conflicts with the COM configuration of the instrumented application. The COM infrastructure might not even be available at the time when a new DLL is loaded and its debugging symbols are imported (e.g. if this function is required for the initialization of COM itself). Therefore, we need to achieve the lowest possible impact on the client application. Should populating the metadata cache not be possible dynamically, the user can still pre-populate the cache. Loading the metadata from cache does not require any external APIs aside from reading a file.

2.8 Stack Frame Serialization

Section 2.4 describes a technique for hooking functions without requiring a signature for broad coverage, with tracing being the primary use case. In Section 2.7, we show how type information can be added to allow for more advanced features like argument tracing or fault injection based on argument values. In order to implement these features, we first need the following operation: Given the location of a stack frame (or more precisely a call frame) in memory, the contents of the registers at call time and a signature, construct a strongly-typed and reusable representation. However, before we look at how this deserialization operation works, we first need to discuss how *Brute* represents values internally.

Values are internally represented as *terms*. A term is an object that has a type associated to it (i.e. the type of the value that it represents). Terms also define a conversion to a string (for debugging purposes) and an evaluation function. The simplest type of term is a *literal term*, which represents a constant value of its respective type. The evaluation of a *literal term* is simply the identity function. More complicated terms, such as *binary terms* for arithmetic, become important when we look at the fault modeling language. For this section, we will just focus on *literal terms*. While a *literal term* is just a container for a constant value and much more expensive to create (in terms of memory and CPU cycles), it still provides two important features, which in some sense, are the inverse of each other. First, inherently provides type erasure, meaning a *term* can store a value of any representable type. Second, it allows for introspection, where, given a *term* we can dynamically check the type of its value. Now we can define *stack frame deserialization* as a function from a signature and a register state at a call site (which, in particular, includes the stack pointer) to a tuple of *terms*, where the tuple elements represent the argument values.

How the deserialization works, depends heavily on the calling convention, which is provided as part of the signature. The following calling conventions are supported (where multiple versions of the calling convention exist, we use the MSVC++ interpretation):

1. **x86/stdcall.** *stdcall* is the primary calling convention used by the Windows API and COM interfaces. It pushes all arguments onto the stack from right to left and the callee is responsible for cleaning up the stack (which implies that it does not support a variable

number of arguments). Return values are stored in EAX for 32-bit and EAX:EDX for 64-bit values. Floating point return values are passed via the FPU stack.

2. **x86/cdecl.** *cdecl* is the standard C calling convention. It differs from *stdcall* in that the caller is responsible for removing the arguments from the stack, thus allowing a variable number of arguments (varargs).
3. **x86/fastcall.** *fastcall* is similar to *cdecl* with the exception that the first two arguments (from left-to-right) are passed in registers ECX and EDX, provided they fit into 32 bits. The remaining arguments (or arguments that are wider than 32 bits) are pushed from right-to-left onto the stack.
4. **x86/thiscall.** *thiscall* is similar to *stdcall* with the exception that the first (implicit) argument (i.e. the this-pointer) is passed in ECX instead of the stack.
5. **x64.** On x64 there exists only a single calling convention. The first 4 arguments are passed in registers RCX, RDX, R8, R9 if they are integral values or XMM0-XMM3 if they are floating point values. The return value is passed in RAX (for integral values) or XMM0 (for floating point).

This list illustrates why it is important to abstract away the calling convention as soon as possible and work with a more versatile representation internally. After deserialization of the call frame, the resulting terms can either be written to disk for tracing or fed into other terms to perform calculations based on the arguments.

For now, we have looked at the conversion from a call frame to a term-based representation. The inverse operation is also required, in order to support calling external functions from a fault model. When a fault model calls an external function, we are given a signature and a tuple of terms for the arguments. Then we arrange the values on the stack and in registers, according to the calling convention, and jump to the given external function. When the call returns, we deserialize the return value (usually in EAX or RAX) into a term and continue the evaluation of the fault model. While the C(++) compiler emits code that reads and creates call frames for every function and function call, we still need the capability to do this dynamically, because the function signatures are only known at runtime and pre-generating code for all possible signatures is infeasible even for very short signatures.

2.9 Rule Specification Language

Brute employs a DSL (Domain Specific Language) to encode the rules that it applies to an application. In a first step, a C++ preprocessor transforms the configuration file. This allows the use of directives like `#include` or `#ifdef` and is compatible with preprocessor libraries like *Boost Preprocessor*[2] and *Boost VMD*[3]. Together with the abilities to specify include files and define additional macros over the command line, this allows for very flexible configuration files. To avoid dependencies on external tools, we include the *Boost Wave*[4] preprocessor implementation inside the *Brute Runtime*.

After preprocessing, the configuration can contain multiple rules and top-level definitions of the fault modeling language.

⟨*Configuration*⟩ ::= (⟨*Rule*⟩ | ⟨*TLD*⟩)*

[2] `http://www.boost.org/doc/libs/1_60_0/libs/preprocessor/doc/index.html`
[3] `http://www.boost.org/doc/libs/1_60_0/libs/vmd/doc/html/index.html`
[4] `http://www.boost.org/doc/libs/1_60_0/libs/wave`

2.9.1 Rule Definitions

A *rule* can target one or multiple functions. For each matching function, it sets one or more function-specific attributes. Rules are aggregated from top to bottom, meaning that multiple rules can target the same function and later rules overwrite the attributes set by previous ones. The grammar for defining rules is as follows:

⟨*Rule*⟩ ::= '`rule`' ⟨*Filter Module*⟩'`!`'⟨*Filter Name*⟩['`(`'⟨*Params*⟩'`)`'] ['`->`' ⟨*Type*⟩] ⟨*Attributes*⟩ '`;`'

⟨*Name*⟩ ::= Id | '"' Name '"'

⟨*Filter*⟩ ::= ⟨*Name*⟩ | '`' Regex '`' | '*'

⟨*Params*⟩ ::= ⟨*Param*⟩ ['`,`' ⟨*Param*⟩ ['`,`' ...]]

⟨*Param*⟩ ::= [⟨*Type*⟩] ⟨*Identifier*⟩

⟨*Attributes*⟩ ::= ⟨*Attribute*⟩ ['`;`' ⟨*Attribute*⟩ ['`;`' ...]]

⟨*Attribute*⟩ ::= ⟨*Include*⟩ | ⟨*Frequency*⟩ | ⟨*Repeat*⟩ | ⟨*Trace*⟩ | ⟨*CallStatic*⟩ | ⟨*Before*⟩ | ⟨*After*⟩

The set of targeted functions is determined by two filter expressions, one for the module name and one for the function name. A filter expression can either be an identifier (possibly enclosed in quotation marks if needed), a regular expression (enclosed by backticks) or an asterisk (to match everything). An identifier (designated *Id* in the grammar) consists of letters, underscores, dots and digits and has to start with either a letter or an underscore. Should a module or function name contain other characters, the name must be enclosed in quotation marks and the standard C-style escaping with backslashes applies. Following the filters, the user can optionally specify a list of parameters as well as the return type. The specification of type information is optional and automatically inferred from the debugging symbols, if available. After the signature follows a list of attributes, which are defined as follows:

⟨*Include*⟩ ::= '`include`' ('`false`' | '`none`' | '`leaf`' | '`recursive`')

⟨*Frequency*⟩ ::= '`frequency`' ('`always`'
| '`never`'
| '`every_nth(`' ⟨*Int Interval*⟩ '`)`'
| '`random(`' ⟨*Float Probability*⟩ '`)`'
| '`custom(`' ⟨*Int Interval*⟩ '`,`' ⟨*Float Probability*⟩ '`)`')

⟨*Repeat*⟩ ::= '`repeat`' ('`infinitely`' | ⟨*Int Repetitions*⟩)

⟨*Trace*⟩ ::= '`trace`' ('`automatic`' | '`none`' | '`call`' | '`arguments`' ['`(`' ⟨*Int Depth*⟩ '`)`'])

⟨*CallStatic*⟩ ::= '`call_static(`' ⟨*VarDecls*⟩ '`)`'

⟨*VarDecls*⟩ ::= ⟨*VarDecl*⟩ ['`,`' ⟨*VarDecl*⟩ ['`,`' ...]]

⟨*VarDecl*⟩ ::= ⟨*Type*⟩ Id

⟨*Before*⟩ ::= '`before`' '`{`' ⟨*Code*⟩ '`}`'

⟨*After*⟩ ::= '`after`' '`{`' ⟨*Code*⟩ '`}`'

The `include` attribute specifies in which cases a rule is applied:

- **false.** The targeted functions are not hooked. None of the other attributes apply.
- **none.** The targeted functions are hooked, but no action is taken for this call and all nested calls.
- **leaf.** The targeted functions are hooked and the actions are performed as defined, but only for this call. No action is taken for all nested calls.
- **recursive.** *(default)* The targeted functions are hooked and the actions are performed as defined, for this call and nested calls (unless overridden by a rule for a nested call).

While the `include` attribute defines if we include a function in the fault injection process in general, the `frequency` and `repeat` attributes provide more fine-grained control over the injection behavior. All three attributes combined represent the *injection strategy* for the targeted functions. The `frequency` attribute controls the regularity at which the fault injection runs. It is determined by two parameters, an *interval* and a *trigger probability*. Every time a function is called, the engine does a probabilistic test that succeeds with the specified *trigger probability*. If the test succeeds, a per-function *interval counter* is increased by one, until it reaches the *interval* value. On the call where we reach the *interval* value, we decrement a per-method *repetition counter* which is initialized to the value specified by the *repeat* attribute. If the counter (before decrementing) is larger than zero, we inject a fault and reset the *interval counter* to zero.

The `frequency` attribute specifies the *interval* and *trigger probability* and provides some convenience options for a more readable configuration:

- **always.** Injects a fault for every call. Sets the *trigger probability* to 1 and the *interval* to 1.
- **never.** *(default)* Never injects a fault. Sets the *trigger probability* to 0.
- **every_nth(**n**).** Injects a fault for every n^{th} call. Sets the *trigger probability* to 1 and the *interval* to n.
- **random(**p**).** Injects a fault with a probability of p. Sets the *trigger probability* to p and the *interval* to 1.
- **custom(**n**,** p**).** Sets the *trigger probability* to p and the *interval* to n.

The `trace` attribute specifies the extent to which a call to a targeted function will be reported in the trace file. Tracing occurs independently of the *injection strategy* for all (or no) calls.

- **none.** Does not write calls to this function into the trace.
- **call.** Writes an entry into the trace file every time this function is called or returns from a call.
- **arguments(**d**)** Same as **call** with the addition of including the function arguments (on entry) and the return value (on exit) in the trace. d indicates the number of pointer-indirections that are followed (e.g. 2 means that the dereferenced value for all pointers and pointers-to-pointers is included as well). If no type information for the targeted function is available, a warning is written to the log and **call** is used instead. If d is not specified, it defaults to zero.
- **automatic.** *(default)* Automatically determines the tracing behavior based on the availability of type information. If no type information is available for a given function, this is the same as **call**. If type information is available, it turns into **arguments(0)**.

The remaining attributes define the *fault model* for the targeted functions. The `before` and `after` attributes specify a snippet of code that will be executed as part of the *before* and *after*

callback. We refer to these snippets as *before* and *after actions*. This snippet of code is written in a DSL described in the next section (Section 2.9.2). The `call_static` attribute allows to declare a set of variables that are scoped to a single call to the function (i.e. they are available in both the *before* and *after action* and keep their values in between the two callbacks).

2.9.2 Fault Modeling Language

The fault models are encoded using a simple programming language that is strongly based on C. However, compared to regular C there exist two major differences.

The first difference lies in the way type information is consumed. While C usually requires header files for interfacing with an API, *Brute* uses the type information it has extracted from the debugging symbols, instead. This eliminates the need for parsing arbitrary header files, which would be beyond the scope of this thesis. Furthermore, it prevents version differences between the header files and the actual implementation. However, it also introduces an additional challenge, since parsing C syntax is not context-free and requires a type symbol table while parsing. Because of this, C only allows to reference types that were defined previously in the file. For cyclic dependencies *forward declarations* are required. Since our type information is only available after the associated module has been loaded, we cannot fully parse the configuration file during initialization. Instead, we only tokenize certain sections of the configuration and parse them on demand. For example, the *before* and *after actions* of a rule are only parsed when the function is first called, ensuring that the type information for that module is already available.

The second difference is that while C is usually compiled, fault models are executed depending on the environment of the target function. For native (i.e. unmanaged) functions, a parsed and type-checked intermediate representation is interpreted for every call. While the execution speed is lower than for compiled and optimized code, the lower performance is often not relevant because fault models are rarely computationally intensive. Should the performance of the fault model still be an issue, a simple solution is to delegate parts or even the whole model to an external DLL, which can be fully optimized. For managed code, we take a different route and translate the code into *IL*, which can then be merged with the target function. The just-in-time compiler will translate the instrumented version of the function, including the *before* and *after action*.

Aside from the *before* and *after actions*, the fault modeling language provides several top-level definitions for defining functions and variables. The following definitions are supported:

⟨*TLD*⟩ ::= ⟨*GlobalVar*⟩ | ⟨*Function*⟩ | ⟨*Import*⟩

⟨*GlobalVar*⟩ ::= (‘`global`’ | ‘`thread_static`’) ⟨*Id*⟩ ‘`->`’ ⟨*Type*⟩ ‘`;`’

⟨*Function*⟩ ::= ‘`function`’ Id(⟨*Params*⟩) [‘`->`’ ⟨*Type*⟩] ‘`{`’ ⟨*Code*⟩ ‘`}`’

⟨*Import*⟩ ::= ‘`import`’ [⟨*CallingConv*⟩] ⟨*Name*⟩‘`!`’⟨*Name*⟩ [‘`as`’ Id](⟨*Params*⟩) [‘`->`’ ⟨*Type*⟩];

⟨*CallingConv*⟩ ::= ‘`__stdcall`’ | ‘`__cdecl`’ | ‘`__fastcall`’ | ‘`__thiscall`’;

Global and thread-static variables are accessible within all functions, *before* and *after actions*. They are instantiated as implied by the declaration (i.e. a single shared variable for `global` variables, an individual instance in thread-local storage for `thread_static` variables). The type of the variable is specified after the variable name, followed by an arrow (`->`) and must be known when the variable is first accessed. Similar to the *rule actions*, top-level definitions are lazily parsed when they are first needed. That is also the reason for the arrow syntax for specifying the type of a variable. While this syntax may look strange from the viewpoint of a C programmer,

C++ uses a similar syntax to specify the return types of lambda expressions and functions that require the use of *decltype* on parameters as part of the return type.

Functions allow to refactor common code in multiple *rule actions* into a single subprogram. If the return type is omitted, `void` is assumed by default. Should the function not require any parameters, we write a pair of empty parentheses (`void foo()`), instead of the C-syntax that uses `void foo(void)`.

Imports allow the use of arbitrary DLL-exported functions in *rule actions* and functions written using the fault modeling language. The `import` keyword is followed by an optional calling convention specification, which defaults to *stdcall* as most of the Windows APIs use this convention. After that, follows the name of the module and function to import. Since some function names are mangled and contain characters that are not allowed in identifiers, the optional `as` keyword allows to specify a different name for the function. The remaining part of the import definition contains the parameter list and the return type. As with functions, omitting the return type uses the default type `void`.

The order in which global variables, functions and imports are defined does not matter, because their resolution is performed on-demand when a referencing *rule action* is used for the first time. In particular, forward declarations of functions are not required.

For the remainder of this section, we look at the statements and expressions that the fault modeling language provides. As mentioned earlier, many elements are taken from C and the language should immediately feel familiar to a C programmer. The following statements are supported and can be used in functions and, with the exception of `return`, also in *rule actions*:

⟨*Code*⟩ ::= ⟨*Statement*⟩*

⟨*Statement*⟩ ::= '`{`' ⟨*Statement*⟩* '`}`'
| '`if`' '`(`' ⟨*Expr*⟩ '`)`' ⟨*Statement*⟩ ['`else`' ⟨*Statement*⟩]
| '`while`' '`(`' ⟨*Expr*⟩ '`)`' ⟨*Statement*⟩
| '`do`' ⟨*Statement*⟩ '`while`' '`(`' ⟨*Expr*⟩ '`)`' '`;`'
| '`continue`' '`;`'
| '`break`' '`;`'
| '`return`' [⟨*Expr*⟩] '`;`'
| ⟨*DeclarationStmt*⟩ '`;`'
| ⟨*Expr*⟩ '`;`'

⟨*DeclarationStmt*⟩ ::= ⟨*RootType*⟩ ⟨*CVMod*⟩ ⟨*Declaration*⟩ ['`,`' ⟨*Declaration*⟩ ['`,`' ...]] '`;`'

⟨*Declaration*⟩ ::= ⟨*DeclMod*⟩ Id ['`=`' ⟨*Expr*⟩]

⟨*DeclMod*⟩ ::= '`*`'* ['`&`']

The standard conditional and looping constructs are supported, with the exception of for-loops. However, these can always be replaced with a corresponding while-loop. Jump instructions like `break` (to exit the inner-most loop), `continue` (to jump to the next loop iteration) and `return` (to exit the function) are supported as well, however the general `goto` is missing. Local variables can be declared anywhere and are scoped to the inner-most compound statement (i.e. pairs of `{}`). Types are written as in C, with additional support for reference types and `bool`.

⟨*IntType*⟩ ::= ('`char`' | '`short`' ['`int`'] | '`long`' ['`long`'] ['`int`'] | '`int`')

⟨*BaseType*⟩ ::= '`void`'
| '`bool`'

| (‘`unsigned`’ | ‘`signed`’) [⟨*IntType*⟩]
| ⟨*IntType*⟩

⟨*CVMod*⟩ ::= ‘`const`’ [‘`volatile`’] | ‘`volatile`’ [‘`const`’]

⟨*RootType*⟩ ::= ⟨*CVMod*⟩ (⟨*BaseType*⟩ | Id)

⟨*Type*⟩ ::= ⟨*RootType*⟩ ⟨*TypeMod*⟩*

⟨*TypeMod*⟩ ::= ⟨*CVMod*⟩ ‘`*`’* [‘`&`’]

The fault modeling language supports a wide range of C operators and expressions, some additions from C++ like booleans and nullptr, as well as some Microsoft-specific language extensions (like the `i` suffix for numeric literals). *HexLiteral* and *DecLiteral* are terminals for hexadecimal numbers (prefixed with `0x`) and decimal integers and floating point numbers, respectively. String literals prefixed with `L` are wide-char unicode literals.

⟨*Expr*⟩ ::= ⟨*Unary*⟩ [⊕ ⟨*Unary*⟩] *where* ⊕ *in Figure 2.7*

⟨*Unary*⟩ ::= ‘`(`’ ⟨*Type*⟩ ‘`)`’ ⟨*Unary*⟩
| (‘`-`’ | ‘`~`’ | ‘`!`’ | ‘`*`’) ⟨*Unary*⟩
| (‘`++`’ | ‘`--`’ | ‘`&`’) ⟨*Atom*⟩
| ⟨*Atom*⟩ [‘`++`’ | ‘`--`’]

⟨*Atom*⟩ ::= ⟨*Value*⟩ (‘`.`’ Id | ‘`->`’ Id | ‘`(`’ ⟨*ExprList*⟩ ‘`)`’ | ‘`[`’ ⟨*ExprList*⟩ ‘`]`’)*

⟨*Value*⟩ ::= ‘`(`’ ⟨*Expr*⟩ ‘`)`’
| ‘`true`’ | ‘`false`’ | ‘`nullptr`’
| HexLiteral
| DecLiteral [‘`u`’] [‘`l`’ | ‘`ll`’ | ‘`i32`’ | ‘`i64`’ | ‘`i8`’ | ‘`f`’]
| [‘`L`’] ‘`"`’ QuotedString ‘`"`’
| Id

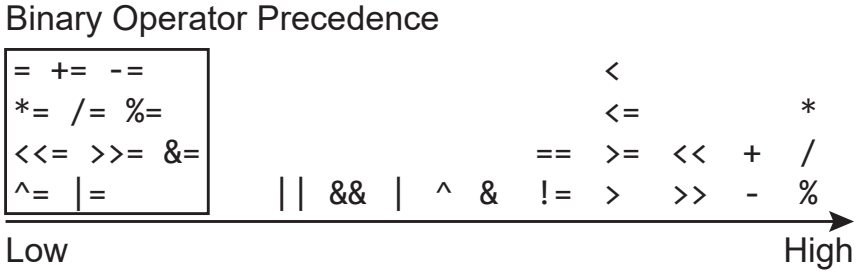

Figure 2.7: Precedence of supported binary operators.

Additionally to global and thread-static variables, code in *before* actions also has access to arguments and call-static variables. The *after* action can access the same variables as the *before* action as well as two implicitly defined, supplementary variables. The first variable is `last_win32_error` of type `unsigned long` and contains the value returned by `GetLastError` when leaving the targeted function. The second variable is `result` of the same type as the return type of the targeted function and contains the return value. Both variables are writable and therefore can be used to manipulate the outcome of the function call.

2.10 Support for Managed Code

While the main focus of this thesis has been on native code, feedback we received during development lead us to the idea to implement a proof-of-concept to demonstrate how we can extend the existing infrastructure to other platforms. While the hooking approach works well for native code, the requirements for instrumenting managed code are vastly different. The low-level approach of modifying assembly code on the fly collides with the just-in-time compiler and garbage collector in many different aspects. First, the address of the assembly code is not known until a function is just-in-time compiled. Second, when we want to work with managed types, we would need to know the stack and object layout for managed functions and types, which is internal to the chosen CLI implementation. Lastly, the instrumented code needs to cooperate with the garbage collector to prevent premature destruction of objects. While the more complex runtime system makes the Detours approach for hooking very difficult, it provides other means of integration that solve all of these problems.

To make the instrumentation of unmanaged and managed code very similar from a user perspective, we keep all the restrictions and the approach that we chose for unmanaged code. Concretely, we do not rely on the source code being available, use type information from the metadata that is embedded in managed assemblies and use the same rule specification language. Internally, the way we instrument the code, however, is very different. While we discussed the implementation for unmanaged code previously, the next sections explain how we transferred the original ideas to managed code.

2.10.1 Approach

As we cannot use Detours to instrument managed code, we leverage the *Profiling API*, provided by the CLR, instead. The *Profiling API* is a COM-based interface that, as implied by the name, was originally conceived as a way to implement managed profilers, i.e. to measure the timing of managed code to find performance bottlenecks. However, the *Profiling API* is much more versatile than that, as it exposes callbacks for various events in the CLR. To mention a few examples, a profiler gets notified when a module is loaded or unloaded and before a function is jitted. This last callback allows a profiler to arbitrarily rewrite the IL code of a function before it is compiled. Therefore, it also allows us to insert code for *before* and *after actions*.

One way of implementing the instrumentation would be to simply add a *before* callback when a function is entered and a *after* callback when leaving a function and use the interpreter to execute the fault model. However, doing this would still not solve the second and third problems that were outlined in the previous section (i.e. object layout and garbage collection). Instead, we chose to compile the *rule actions* directly into IL and inject them into the function at the appropriate locations. Since the *rule actions* are thus embedded into the remaining function code, we can use the same IL instructions that the function uses for accessing object members, therefore solving the aforementioned issues.

Readers familiar with Pex[26] might recognize some similarities, as both technologies use the *Profiling API* to rewrite managed code. Pex inserts callbacks for every IL instruction, which allows to track the execution in a symbolic way with the ultimate goal of implementing dynamic symbolic execution. *Brute* only inserts callbacks on entering and exiting a function, but allows the modification of argument values to inject faults.

2.10.2 Implementation

Support for managed code is directly built into the *Brute Runtime* and is activated by the CLR when it is loaded into the process. During the *late initialization phase*, we inject a set of environment variables into the current process' environment. These environment variables indicate the presence of a profiler to the CLR and consist of a path to the DLL that implements the profiler (which is the *Brute Runtime Library* itself) and a GUID that identifies a COM class that implements the `ICorProfilerCallback5` interface, which is expected by the CLR. Should an external profiler be detected, *Brute's* profiler is automatically disabled.

During the subsequent initialization of the CLR, it loads the profiler DLL (which is a no-op as the module is always already present) and creates a new instance of the profiler. The *Brute Profiler* then locates the remaining parts of the *Brute Runtime* in the process and uses this connection to access the configuration and rule definitions. As part of the initialization, the profiler registers callbacks for module load, unload and rejitting. ReJIT is a new feature, added to the profiling API in version 4.5 of the CLR. It provides a set of functions that instruct the just-in-time compiler to call the profiler before compilation to rewrite the code. Should the code already be compiled, it will discard the current version and recompile the code with instrumentation. While rewriting IL was already possible in earlier versions of the CLR, the ReJIT API facilitates many of the operations.

The IL rewriting itself consists of three phases. First, all the instructions are analyzed and branch offsets are converted into symbolic values. This is necessary because we invalidate all non-relative jump targets by inserting instructions at the beginning of the instruction stream. In a second phase, we attach the *rule actions*. This is accomplished by compiling the rule actions into IL code and inserting them at the respective locations. For the *before action* this is simply done by prepending the instructions to the instruction stream. For the *after action*, we need to deal with the fact that functions can have multiple exit points and can also be exited by throwing an exception. To ensure that the *after action* runs in every scenario, we wrap the function in a `try` ... `finally` block. To call unmanaged functions (i.e. to determine if we should inject a fault) from managed code, we can emit additional *P/Invoke* functions into the assembly. *P/Invoke* is a mechanism that allows managed code to call DLL-exported, unmanaged functions. Finally, in the third phase, we convert the symbolic jump targets back into concrete offsets and reassemble the IL into a single, contiguous buffer. The new instruction stream replaces the original implementation.

2.10.3 Conclusion

The approach is powerful enough to transfer the concepts from unmanaged to managed code and target both runtime systems in roughly the same manner. Due to time constraints, the current implementation only supports a subset of the *fault modeling language* for managed code. Achieving feature parity would imply the implementation of a full-fledged IL compiler, which was out of scope for this project. However, many of the challenges of interfacing with the CLR have been solved as part of this proof-of-concept, leaving the implementation-heavy parts for future work. Another limitation is, that only a single profiler can be attached to a process at the same time. This implies that *Brute* cannot be used to inject faults into managed code while another profiler (e.g. Pex) is also attached. The unmanaged code injection, however, is not affected by this restriction.

Support for managed code provides an interesting opportunity to show how many of the concepts and ideas that were developed for unmanaged code can be translated into a different execution environment. By having a DSL for specifying the fault models, we can target different languages. The effort of targeting other languages is greatly reduced by sharing the common

infrastructure code. As an example, using the *JVM HotSwap API*, an adaptation to JVM-based languages is conceivable.

2.11 Tracing

The built-in tracing feature collects information on function calls and writes it into a binary trace file. After the target application terminates, the launcher analyzes the trace file to generate a report. The contents of the report can be customized using command line options. An existing binary trace can also be reanalyzed later (offline analysis). As shortly discussed in Section 2.9, tracing can be configured separately from fault injection and provides three levels of detail. The tracing level can be different for every function.

At the lowest tracing level, we log function entry and exit events. This allows to reconstruct a coarse-grained execution trace. Additionally to the function name, we also trace the last Win32 error code when leaving a function. By employing generic hooks, this level of tracing does not require type information. On the next level, we include the arguments and the return value for each call, therefore type information is needed. On the highest tracing level, we also follow pointers up to a configurable limit. The heap serialization algorithm can deal with null(ish) pointers (i.e. addresses $\leq$ 0x0000FFFF), invalid pointers and cyclic data structures. Pointers into the first 64KB of virtual memory are treated as null pointers, because Windows prevents the usage of this area of memory. Otherwise, we assume that the pointer is valid and use SEH (structured exception handling) to gracefully handle the access violations that occur should we touch an invalid address. To detect cyclic data structures, we keep a set of all previously seen addresses.

During a run of the target application, we write the trace in binary form for smaller file size and higher tracing speed. At a later point in time, the binary trace can be analyzed and different aspects can be exported separately. The information that should be extracted can be specified as part of the command line options. The following aspects can be exported:

- **Summary.** The summary is always exported and contains high-level statistical information. We document the number of known functions (i.e. the function entry points that we found in all DLL export tables), the number of successful and failed hooks, the number of intercepted function calls as well a summary of the imported type information. Following that, we print the names of the 20 functions with the highest call count. Finally, we print a list of all functions that were called at least once with the number of function calls and the average call depth. The call depth allows an estimation of whether the functions were used primarily internally (if the number of on the higher end of the spectrum of all averages) or as a surface API function (close to zero).

- **Function Listing.** Prints a map of all known functions, grouped by module and sorted by name. For each function, the map includes to which extent type information is available and if it is compatible with *Brute* (i.e. if the function uses types that are representable using the internal type system).

- **Call Trace.** A textual representation of all function entry and exit events. The stack depth is visualized using indentation. Every event contains the ID of the thread on which it occured, allowing the reconstruction of a function trace for concurrent programs. The events are serialized using a monitor during the capture of the trace.

- **Call Data.** Prints the captured arguments and return values converted to a string. By default, pointers are included in the trace, followed by their value, provided we successfully

followed the pointer indirection. To reduce the noise and simplify the further analysis of the data, the concrete addresses can also be excluded and replaced with an opaque <PTR> expression, followed by the value (if available). We used this representation in the evaluation as one possible method for finding functions that are suitable for fault injection (see Section 3.2.2). If both the call trace and call data are exported, the arguments and return values are always printed on the line directly following the call trace event during which it was captured.

The tracing feature was implemented as the first component of the framework for two reasons. First, it allowed us to check the compatibility of our instrumentation on a large number of functions. Second, we used the results of the tracing report to guide our search for target functions during the evaluation.

2.12 Challenges

In this section, we highlight some of the unexpected problems that we experienced and some of the more challenging aspects of the implementation.

2.12.1 Finding Function Entrypoints

In order to support wildcards and regular expressions in rules, we first need a collection of all available functions. On a first level, we maintain a collection of all loaded modules by enumerating all statically loaded modules and tracking the dynamically loaded modules. Assuming we have a way for enumerating all exported functions in a module, this would provide us with the full list of exported functions in the process. While this sounds trivial, given that every DLL contains an export table, some unexpected difficulties arose in practice. In this section, we describe the list of different solutions that we tried to apply before reaching the final implementation.

As mentioned before, every DLL contains a section that encodes a table of all exported symbols with their RVA (relative virtual address). This is the address where a symbol is stored in memory relative to the base address at which the module was loaded. An example of such a table can be found in Table 2.1.

Oridnal	Name	RVA
0x0001	ud_decode	0x01553E73
0x0002	ud_init	0x01553EA4
0x0010		0x01553F9A
0x0011	ud_set_mode	0x01553FB0
0x0012		0x01553FC4

Table 2.1: Example of a DLL export table.

Every row contains a 16-bit ordinal number, an optional name and an RVA. Other modules can reference these symbols either by ordinal number or name (should one be specified in the table). The symbol name is also used for function filtering, should no debugging symbols be provided. The difficulty arises from the fact that these symbols may reference different types of exported symbols, without specifying their type (e.g. function or data). For the standard use case, this information is not necessary, because it is available during compile time and not required at runtime anymore. If we use the full table and the user specifies a wildcard, we run into the danger of instrumenting a supposed function, which later turns out to be data instead. “Instrumenting

data" has detrimental effects on the target application. Some of the data is interpreted as code and rewritten by Detours, should it resemble something similar to assembly. This introduces random changes to these exposed variables, which usually results in a crash. Thus, having a precise list that contains only functions is critical.

Our first approach was to dynamically analyze the memory at the specified RVA before instrumenting it and checking for signs, indicating that the memory does not contain a function. Should a function contain invalid instructions or instructions that require CPL-0 (i.e. kernel mode), this is a strong indicator that the given address does not actually refer to a function. While this approach was sufficient for certain scenarios (e.g. starting *Notepad*), it failed for more complex cases (e.g. opening a common file dialog). We increased the restrictiveness of this filter by not only excluding based on the aforementioned criteria, but only instrumenting functions that start with either a `push` or `mov` instruction. According to a histogram that we built from our set of test applications, this covers 97% of all exported functions. This approach worked well for x86 applications, where most functions create an EBP-based stack frame and thus start with one of the two instructions. On x64 however, most compilers do not emit stack frames anymore, rendering the histogram approach unusable.

A next approach was to check in which *section* an address was contained. The *PE* file format contains a section table, which indicates, amongst other information, which areas of the image are executable, read-write data or read-only data. Should an exported symbol reside in a non-executable section, it cannot contain code and thus cannot be a function. This approach worked better, because it is applicable for x86 and x64 code and increased the compatibility of *Brute* to a larger set of DLLs. Unfortunately, some DLLs put read-only data inside of code sections and thus diminish the applicability of this approach.

As a final resort, we decided to use the debugging symbol information. We search for each address from the DLL export table and only include addresses if we find a corresponding debugging symbol of a function that starts at the same address. Using this approach we were able to precisely recover the list of functions and exclude any other kinds of symbols.

There have been different approaches to filtering out functions from the export table. The only successful way of enumerating only the functions was using symbol information. This is not optimal, as debugging symbols are not available for many closed-sourced libraries. However, the need to enumerate all functions of a library is only present when wildcards or regular expressions are used in a rule. For precisely targeting a single function, the question of the type of an exported address does not even arise. Should symbol information not be present, we automatically fall back to the section-based approach, which also works for many of the tested DLLs.

2.12.2 Function Aliasing

An issue that was faced early on in the development, was the problem of function aliasing. During the evolution of the Windows API, the implementation of some of the subsystems and functions were moved between DLLs. To keep compatibility with older applications, the DLLs that originally implemented these functions still export them, but only contain a redirection to a different DLL. Two prime examples are `kernel32.dll` and `kernelbase.dll`. Introduced in Windows 7, `kernelbase.dll` contains many of the function implementations that were previously found in `kernel32.dll`. The entry points in `kernel32.dll` are only redirections and contain no implementation. Aliases occur in three different variants:

1. **Direct aliases.** Two entries in the export table with the same RVA. This type of alias is only possible for functions in the same DLL.
2. **Indirect aliases.** A function which only contains an unconditional jump to a different function. Detours does not instrument this type of method and will instrument the target

method instead, up to a certain number of indirections. The number of indirections is determined by the type of jump instruction used (relative, absolute or indirect) and the order in which these instructions occur. The maximum number of indirections followed is three.

3. **Complex aliases.** A function which contains instructions that have no side effects (e.g. `mov edi, edi` or creating and immediately removing a stack frame) followed by an unconditional jump. Depending on the size of the *NOOP* instructions, Detours can instrument these functions, however, it will never follow the indirection.

Aliased functions are problematic for various reasons. In case of a direct alias, we hook the same function twice, resulting in unexpected behavior of the *before* and *after actions*. For indirect aliases, we are not sure which version of the function we instrument, because of the erratic rules for following indirections employed by Detours. Depending on the entry point that a caller uses, it might see different behavior for the same function (one entry point is instrumented, the other one is not). Also, it would be possible to have contradicting rules on aliased functions (because they have multiple names) which ultimately only exist once in memory. In this situation it would not be clear, which rule would apply.

To address all these problems, we only allow rules to be applied to the implementation of a function, not to an alias. We build a DAG (directed acyclic graph) of all exported functions and add an edge for every indirection. When a rule is applied, we follow the edges (i.e. the indirections) until we reach a sink (i.e. a vertex with no outgoing edges, which must exist and represents the function containing the implementation). Should rules from different aliases apply conflicting settings to a function, the rule closest to the implementation is used. Conflicting rules are ignored and a warning is issued for each one.

The alias detection can handle direct and indirect aliases. Complex aliases can not be handled, because automatically distinguishing them from an implementation is very difficult. We would need some sort of analysis that can proof, that a sequence of arbitrary assembly instructions is side-effect free, which is beyond the scope of this thesis. The problems connected with complex aliases are also less severe, as *Detours* will not follow the indirection and instrument an unexpected function. The problem, that a client can experience different behavior depending on the alias that is used to call the function, however, can still occur.

2.12.3 Automatic Recovery

In Section 2.2 we discussed the way *Detours*[9] instruments a function. This technique of instrumenting assembly code does not always work and can result in invalid code. *Brute* contains an automatic recovery mechanism that tries to transparently unhook a function, should its instrumentation lead to an exception. In this section we explain how this invalid instrumentation can occur, how we detect it and how we attempt to automatically recover from the exception state.

Consider the assembly code of the function `RtlSplay` outlined in Figure 2.8. On the last line (address ending in `...77B`), we have an unconditional jump to the second instruction of the function. Up to this point, all code is valid and the function would run normally.

However, when *Detours* instruments this function, it will place an unconditional jump at the beginning of the function. Because the jump instruction is longer than the original instruction (`xchg ax, ax`), it will replace not only the first but multiple instructions. As shown in Figure 2.9, for the instrumented function, the jump on the last line now points to an address inside of the jump instruction to the hook. When the execution reaches this point, there will most likely be an invalid instruction and the CPU will trigger an exception. This forces the execution to jump

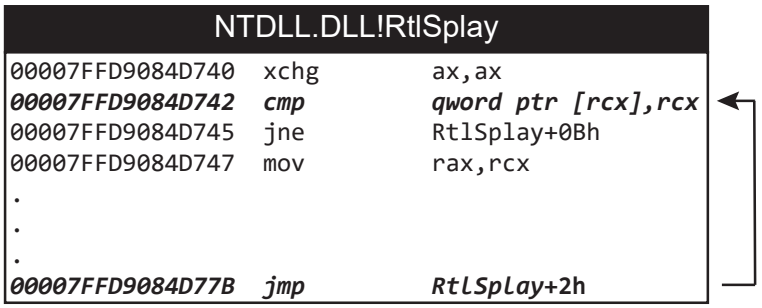

Figure 2.8: Function before hooking.

to the installed exception handler in kernel mode. The Windows Kernel converts the hardware exception into a SEH exception and transfers the control back to the responsible process.

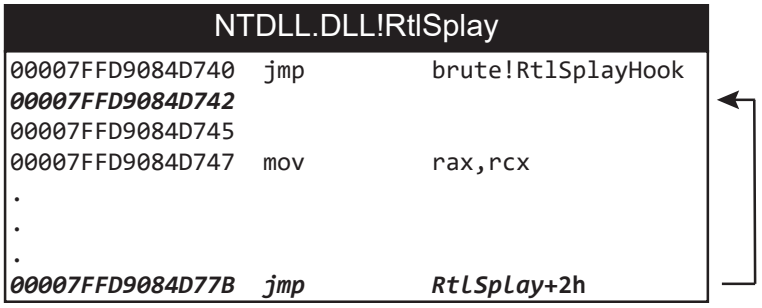

Figure 2.9: Function after hooking.

When Windows delegates the exception to the process, it jumps to an assembly stub exported as `KiUserExceptionDispatcher` in `ntdll.dll`. The stub prepares the process for calling the exception handler. As part of this, it calls a function named `RtlDispatchException`, which is responsible for finding the appropriate exception handler or terminating the process if no handler was found. This function receives two arguments, one containing the exception information (including the type of the exception) and the other points to the exception context, a structure that contains the register state of when the exception occurred. Under normal circumstances, `RtlDispatchException` would start to unwind the stack until it finds an exception handler (i.e. a `__try ... __except` block) and then invoke it. It is not possible to install a global first-chance exception handler using a public API.

In order to recover from the illegal instruction exception, we need to unhook the function that caused the exception and retry the same instruction that previously failed. Therefore, we need to install a handler that runs before any stack unwinding occurs. To accomplish this, we install a hook on `RtlDispatchException` and first check if the exception occurred inside of an instruction that was inserted by *Detours*. If this is the case, we unhook the function, thus restoring the original instructions. By passing the exception context to the Windows API function `NtContinue`, we try to continue the execution at the same address where the exception occurred. The difference being, that with the original instructions having been restored, the execution should now succeed.

A complicating fact is, that `RtlDispatchException` is an internal function and therefore, we

do not know its address. We solved this issue by automatically decompiling the assembly stub at `KiUserExceptionDispatcher` (which is at a well-known address) during initialization, finding the instruction that calls `RtlDispatchException` and extract its address from there.

2.12.4 Shared Resources

The application under test and the *Brute Runtime* share the same address space. This simplifies many of the interactions between the two and is a prerequisite for the Detours-type hooking to work. At the same time, we need to minimize the impact of the instrumentation to prevent false positives. While some resources, like the virtual address space or CPU time, are inevitably shared between the application and *Brute*, we introduce counter-measures to prevent sharing of the C runtime and the process heap.

As a motivating example, consider the following scenario: An application creates two threads and calls two different functions `foo` and `bar` in parallel using these threads. Both functions have tracing enabled, therefore entering these functions requires *Brute* to acquire a lock that protects the access to the log file. Additionally, it also needs to allocate memory while holding the tracing lock for capturing the function arguments. A potential execution trace is shown in Figure 2.10. For this execution, we run into a lock ordering problem, because thread 1 first acquires the tracing lock and later tries to allocate memory while thread 2 first locks the heap and then tries to acquire the tracing lock. The result is a deadlock, that was introduced solely because of the additional lock required by the tracing functionality.

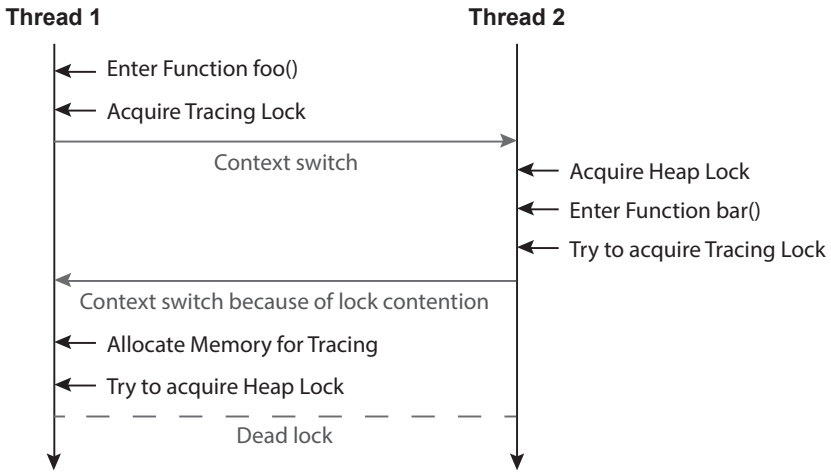

Figure 2.10: Execution trace leading to deadlock because of a shared heap.

The usual approach to solving deadlocks caused by lock ordering is to introduce dependencies between the different locks and to include all dependencies when acquiring a lock. If the locks are always acquired in the same order (for example given by a fixed topological sorting of the dependency graph), this type of deadlock cannot occur. For our situation, this solution is not applicable, because the ordering problem is between locks owned by *Brute* and the client application, over which we have no control. We also do not want to restrict the use of *Brute* to functions where no ordering problem occurs.

The solution we chose introduces a secondary heap that is used for *Brute* only, thus isolating the application heap from our private heap. Since every heap has a separate lock, the user code

is not able to lock the heap that we need for tracing etc. anymore, thus avoiding the deadlock. We accomplish the heap isolation by multiplexing the access to low-level heap API functions, depending on the type of code that is currently running on the thread.

The low-level process heap management is implemented in `ntdll.dll` and exposed to the user through indirect aliases in `kernel32.dll`. A process can contain multiple heaps, each identified by a heap handle. A default heap, whose handle can be determined using the `GetProcessHeap()` in `kernel32.dll`, is created as part of the process initialization. When requesting memory using a higher-level API function (e.g. `GlobalAlloc`) or the C runtime (e.g. `malloc`), the call is forwarded to the respective lower-level API (e.g. `RtlAllocateHeap`) and the default heap is used. *Brute* instruments these lower-level APIs (i.e. `Rtl...Heap` functions) and redirects the requests to a different heap, depending on the calling context. If the thread is executing user code, no redirection occurs and the default heap will be used. Should the current thread be inside of instrumentation code, we redirect the calls to the private heap by replacing the heap handle that is passed in to the `Rtl*Heap` function. The calling context can be determined by checking the same thread-local structure that is used for keeping the shadow stack, necessary for the generic hooks (see Section 2.4). Internally, we use an additional third heap to keep track of the sections of memory that were allocated, to prevent errors, should a pointer be used in a wrong context (e.g. a block that was allocated on the private heap is released on the default heap).

By instrumenting the heap functions at a low level, not only built-in allocations are redirected to the private heap, but also allocations performed by functions that are called as part of the *rule actions*. Therefore, all code that runs as part of *rule actions* is automatically subject to heap isolation. This is by design, because otherwise the implementations of the called function would also have to be aware of the lock ordering problem. For user-implemented code this would be tedious at best, while closed-source libraries that allocate memory simply could not be used as part of *rule actions*.

While we need to hook the low-level heap functions for heap isolation, it is still possible to define rules for these functions. From the view of a rule developer, the original implementation for the heap functions already contains the multiplexing logic (i.e. the original implementation from the view point of the rule developer is the instrumented implementation from the view point of the heap isolation, the final implementation provided by the operating system is not visible for the rule developer).

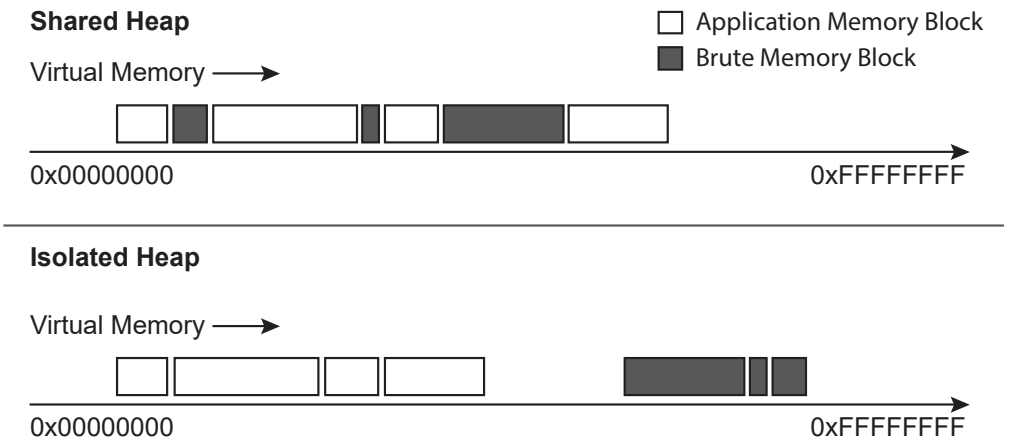

Figure 2.11: Memory layout for shared vs. isolated heaps

Separating the memory allocations has an additional advantage. Without heap isolation, the allocations of the application and *Brute* are interspersed, as shown in the upper half of Figure 2.11. By having two separate heaps, there is a gap in virtual memory between the memory used by the application and the memory reserved for *Brute* (lower half of Figure 2.11). This gap reduces the risk of memory corruption inside of *Brute's* data structures, should the application overflow a buffer because of an injected fault. This simplifies the crash analysis, as the origin of the heap corruption is easier to pinpoint.

While the heap is the most critical shared resource, there are other resources that need to be multiplexed (e.g. standard io, current locale, etc). Luckily, handling concurrency for most of these resources is done by the C runtime and isolating them can be done by statically linking the standard library into the *Brute Runtime*. As a result, we get a private copy of all the internal C runtime state, including all its locks. However, contrary to the heap isolation, which affects all code, statically linking the C runtime must be done for each library separately. This means, that when a DLL is written specifically to be called as part of *rule actions*, static runtime linkage should be enabled. Third-party libraries that link dynamically to the C runtime can still be used, as long as no sharing problem occurs. However, for this scenario *Brute* does not support the developer with additional safeguards, like it does for heap isolation.

Chapter 3

Evaluation

Strong aspects of *Brute* are its low entry threshold and the applicability to a wide spectrum of programs. For the evaluation, we wanted to take advantage of these properties as much as possible. Therefore, we did not focus on a very specific testing scenario, but tested the robustness of commonly used software products against well-known deviant behavior. As the primary objective, we wanted to determine if *Brute* can fulfill the requirements motivated in the introduction of this thesis. Concretely, we want to verify the following:

(1) Can we successfully apply *Brute* to an unmodified, production-grade application?
(2) Can the same rule be applied to different applications without modification?
(3) Are the observed crashes reproducible? When running an application with fault injection twice, will it crash at the same location?

Furthermore, we wanted to extract the following information from the test data:

(4) Are there cases where our testing methodology or infrastructure introduces false positives? If yes, are there discernable patterns where this occurs and is it an implementation or conceptual issue?
(5) How do the different injection strategies compare to each other? Is there a clear advantage to using a specific strategy?
(6) Are the crashes that we find unique?

In the course of this evaluation, we selected a set of commonly used applications to test against. Next, we selected a set of functions using different strategies. For each of these functions we developed a simple fault model by checking for potential error codes online. We then combined the fault models with different injection strategies, in order to see which strategies perform better and determine why this is the case. To simplify the testing process, we have developed a testing infrastructure that performs the evaluation automatically and generates a report containing different operating figures. The following sections explain the evaluation environment, our test infrastructure and how we selected the different sets of applications, functions and injection strategies.

3.1 Infrastructure

To collect the evaluation data, we implemented an automated evaluation infrastructure. In addition to running the different scenarios unattended, it also provides interaction automation

(i.e. simulation user input), collection of the measured figures and simplifies the reproduction of a specific scenario for further investigation. Figure 3.1 shows a conceptual overview of the evaluation infrastructure.

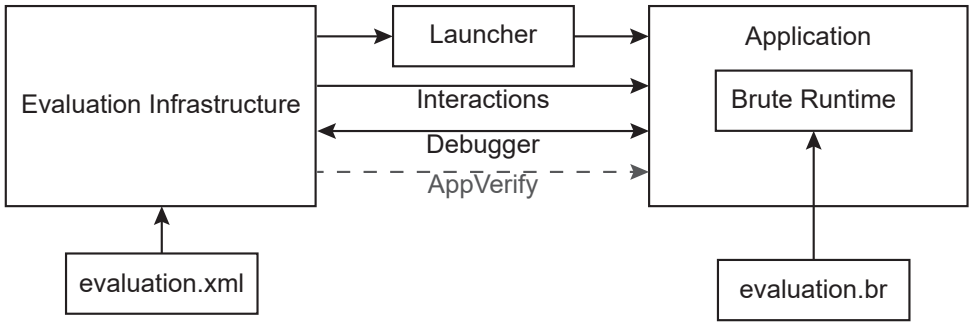

Figure 3.1: Overview of the evaluation infrastructure.

The infrastructure is driven by a configuration file (`evaluation.xml` in the figure) that provides a list of executables. The executables we used for this evaluation are listed in Section 3.2.1. For each executable we specify a list of fault models, a list of injection strategies and instructions for automated interactions. We run the launcher for every combination of application, injection strategy and fault model. The implementation of the fault models is stored in `evaluation.br` and we use the C preprocessor feature to enable the desired fault models over command line arguments. The automated interactions are a list of actions (e.g. send keystrokes, wait for a window, delay, etc.) that simulate user input and are used to gracefully close the application or perform other activities like opening a file. Furthermore, we support to enable *AppVerifier* analyses for each application. This was originally developed to improve the detection of buffer overflows, but later disabled because of conflicts between *Brute's* instrumentation and the modifications that *AppVerifier* performs.

While the application runs, the infrastructure monitors the application over the debugging interface. All thrown SEH exceptions are being counted and logged. We differentiate between first-chance and second-chance exceptions. First-chance exceptions are reported whenever an exception is thrown, even when an exception handler is installed. Second-chance exceptions are unhandled exceptions, which result in a crash of the application. The second type of exception was used to count the number of detected crashes.

3.2 Setup

3.2.1 Selection of Applications

For our experiments, we selected twelve well-known applications, based on their extensive use in production and their sizable test suites. Table 3.1 shows the applications along with a short description. We tested the latest version of all applications (at the time of writing) on a machine running Windows 10.

For each of the applications we selected a test scenario, which is a sequence of operations that are to be performed under active fault injection. For most applications, this is simply starting the application and closing it again once it is fully loaded. While this is a seemingly simple scenario,

Application	Description
Internet Explorer	Browser
Microsoft Excel	Spreadsheet
Microsoft Notepad	Text editor
Microsoft OneNote Launcher	Launcher for note-taking application
Microsoft Outlook	Email and calendar application
Microsoft Paint	Graphics application
Microsoft PowerPoint	Presentation application
Microsoft Publisher	Desktop publishing application
Microsoft Visio	Diagram and graphics application
Microsoft Word	Word processor
Notepad++	Source-code editor
Skype for Business	Communications platform

Table 3.1: The target applications used in our experiments.

the initialization process of an application is often fairly complicated, as many of the resources (e.g. GUI components, file handles for configuration files, etc.) are allocated during this operation.

For the applications from the Office suite, we needed additional automation to ensure, that they always start in *normal mode*. When an Office application crashes during initialization, it will show a dialog box asking the user if the application should start in *safe mode* on the next restart. *Safe mode* prevents add-ins from loading and uses a clean configuration to prevent stability issues caused by third-party extensions. Since the dialog box suspends the initialization and we always want the application to run in *normal mode*, we have an automation script in place to automatically dismiss this dialog.

For notepad we defined a more complicated scenario. After notepad is initialized, we simulate the keystrokes for `Ctrl+O` to show the *open file dialog*. This dialog is very complex to initialize, as it contains a *Windows File Explorer Frame*, which can display previews of various file formats and supports plugins called *shell extensions*. Opening this dialog more than doubles the number of loaded DLLs in notepad. After the *open file dialog* has loaded, we cancel it again and close notepad.

3.2.2 Selection of Functions

We used the argument tracing feature to generate a list of all functions that Microsoft Notepad and Notepad++ use for their respective scenario including the input/output pairs (i.e. the combination of argument and return values) for each individual call (see Section 2.11 for more information about tracing). From this pool of close to 3000 functions and 4 million function calls we distilled out 40 functions using two different strategies.

First, we searched for functions containing the word `Create`, since these functions usually allocate some sort of resource. Such allocation attempts might fail sporadically, should the system run out of a certain resource or because of insufficient privileges of the caller. For a subset of these functions, we consulted the documentation to verify, that the functions indeed depend on external resources. In the end, we selected 11 functions using this strategy. A good example for such a function is `kernelbase.dll!CreateFile`, which creates a handle for a file on disk or an I/O device. Should the caller have insufficient rights to access the file or device, or should the given path be invalid, this function will return `INVALID_HANDLE_VALUE` and set the corresponding error code.

As a second strategy, we searched for functions which potentially have external dependencies. Here we used the collected input/output pairs to reduce the set of functions to the ones that return different values for the same arguments. We dealt with pointers in arguments and results by comparing the value that they point to, up to three degrees of indirection. Since the degrees of indirection are limited, this introduces a certain imprecision, but is still far more precise than comparing pointers by reference. The reasoning is, that the results of these functions are likely to be affected by some external dependency, as the returned value is not fully defined by the captured argument values. Since external components are beyond the control of the target application, they are potentially subject to change, e.g., by other applications or the operating system. Consequently, if there is a dependency to an external component, such as the Windows registry or environment variables, any unexpected change in that component could cause the target application to fail.

We selected 29 functions with this strategy, for which we verified that they indeed depend on an external component by consulting the documentation. An example for a function from this set is `kernelbase.dll!GetEnvironmentVariableW`, which retrieves the value of an environment variable. It could happen that an environment variable is not defined or its content was changed. If an application makes any assumption about the existence or contents of an environment variable, it might crash if the variable was changed or deleted by the user or another tool.

Although both strategies attempt to find functions with external dependencies, neither one is subsumed by the other. In fact, the two subsets yielded by the different strategies were fully disjoint.

3.2.3 Selection of Fault Models

After choosing 40 functions, we defined a fault model for every function. When a system call fails, it usually indicates this with a special return value. An error code is set, which can be determined using the function `GetLastError` and describes the error more closely. The currently defined range of error values lies between 1 and 16000 (zero indicates success). Depending on the error code, the failure may be handled differently. In order to reduce the risk of introducing spurious errors, we wanted our fault models to return error codes which are also encountered in practice. For this, we searched the MSDN, StackOverflow and online forums for error codes that people have encountered in real situations. To illustrate this, take for example `advapi32.dll!RegQueryValueExW`, which copies the value of a registry key into a user-allocated buffer. Since registry values can be of arbitrary length, it is possible that the value will not fit into the allocated buffer. If this is the case, the function will fail and set the error code `ERROR_MORE_DATA`. The function indicates the necessary buffer size to the caller, which allows to allocate a new buffer of sufficient size to read the value successfully.

To build fault models based on error codes, we used the *after action* which changes the result of the function to indicate an error and set the appropriate error code. We provided 39 *after* and 1 *before* action. 30 of the *after* actions needed to set an error code. The high number of *after* actions compared to the single *before* action is explained by the fact, that we mostly simulated system calls which failed because of an external dependency. In these cases, it is valid for the operation to have completed and the failure being indicated via the return value and, in some cases, the error code.

3.2.4 Selection of Injection Strategies

Finally, we selected five different injection strategies. With the exception of Never, we wanted to compare the effectiveness of the different injection strategies and motivate them with different

scenarios:

- Never. Instrument the function but never inject a fault. We use this scenario to get a baseline, which ensures that *Brute* does not interact with the application in a way that could result in a crash. If running the application with *Brute* in this scenario reveals a crash, we have found an instrumentation bug in our framework and the results for this configuration are invalid.
- Always. Inject a fault every time the function is called. An example for this would be a function that is permanently not executable, i.e. opening a file without the necessary privileges.
- Once. Inject a fault for the first call during the process life time. This could simulate a function that requires a resource that has not yet been initialized.
- EveryOtherCall. Inject a fault for every second call. This can simulate a fault that can be solved by simply retrying the call.
- FiftyFifty. Inject a fault with a probability of 50%. This can simulate a scenario where a function depends on an unstable resource, e. g. a weak network connection.

3.2.5 Function Grouping

For the base evaluation, we run a test for every combination of function × injection strategy × application, which means that for each run, we injected faults in only a single function. To evaluate whether certain faults are only found when we inject faults into multiple functions at the same time, we wanted to perform additional tests with function groups. However, an exhaustive evaluation over all possible function combinations yields far too many configurations to test and probably does not reveal any additional bugs, if the functions are completely unrelated. Based on this reasoning, we defined the function groups by API (i.e. a group for file related APIs, a group for registry APIs, etc.) This only adds a few additional configurations and has a higher chance of finding interactions between multiple functions, since they contextually belong together.

3.3 Results

Table 3.2 shows the results of our evaluation for the ungrouped tests. The first column shows the name of the application, the second column the injection strategies, the third column the number of crashes in each application per injection strategy, the fourth column shows the total number of calls to instrumented functions and the fifth column the number of injected faults. By successfully evaluating over the specified applications, we have achieved objective (1) as stated in the introduction of this chapter. To verify objective (3), which states that the injected faults should be reproducible, we run the full test set three times. Each table row shows the accumulated number of crashes, injected faults and total calls over all runs and all functions.

In the following sections, we focus on different aspects of the results and try to verify the remaining objectives that we defined at the beginning of this chapter. We conclude the results by analyzing one of the crashes that we found in depth.

3.3.1 Fault Models

The column *Rules* in Table 3.2 shows the number of rules that where used for each injection strategy and application. Recall that we performed the original selection over functions from Microsoft Notepad and Notepad++. However, the extracted rules were also useful to find bugs in other applications. As noted in objective (2), this is one of the benefits we strived to achieve.

Application	Inj. Strategy	Crashes	Rules	Total calls	Inj. 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

Table 3.2: The crashes detected by *Brute* in each target application.

One concern during the development of *Brute* was the introduction of false positives. This was also stated in objective (4) of the evaluation. *Brute* could report false positives for two different reasons: either because the instrumentation interacts with the application in way that leads to a crash or because the injected fault models are unrealistic or overly aggressive. To check for the first type of false positive, we used the NEVER injection strategy. We do not show the numbers for NEVER in Table 3.2, as the instrumentation did not lead to a crash. However, it is to note, that this only holds true for the scenarios where we targeted a small, specific set of functions. When using the tracing infrastructure on all DLL-exported functions, some applications crashed or terminated, because they try do detect such modifications as part of the protection against software piracy and tools that try to circumvent the license verification.

The test for unrealistic fault models is more complicated and we use a probabilistic approach. We employ the following heuristic, which for each fault model f_i computes the *real-bug indicator* rbi_i:

$$rbi_i = 1 - \frac{\text{number of crashes by } f_i}{\text{number of injected fault by } f_i} \tag{3.1}$$

The number of crashes and injected faults is summed over all injection strategies and applications. rbi_i indicates the likelihood that a fault model f_i reveals real bugs based on the following reasoning: The denominator shows the number of injected faults that the applications were able to withstand. The larger this number is, the more confident we can be that it is possible to handle an injected fault properly. If the numerator (i.e. number of crashes) is small but larger than zero, only a small fraction of the call sites were not resilient to the fault, which indicates that we found a bad call site that could be fixed. On the other hand, when every injected fault leads to a crash at every call site, there is a high chance that it cannot be handled properly, since none of the applications are resilient to it. This makes it more likely, that the introduced behavior does not occur in practice, otherwise there would probably be more call sites, which handle the fault.

We have calculated the indicators for all 25 fault models that caused at least one crash and visualized them in Figure 3.2. The x-axis shows the index i for each of the 25 fault models f_i and the y-axis the respective indicator rbi_i calculated by the formula (3.1). Note the range on the y-axis of 0.8 - 1.0. According to this data, the probability that we found real bugs is very high.

3.3.2 Injection Strategies

As described in evaluation objective (5), we wanted to determine how the different injection strategies compare to each other and how they influence the target application.

When we look at the number of crashes per injection strategy, we can immediately see that the choice of injection strategy has an impact on the number of crashes. For all target applications, ALWAYS, being the most aggressive strategy, detects the most crashes, with a total of 62. Following that are the FIFTYFIFTY strategy with 52 and EVERYOTHERCALL strategy with 51 crashes. The least crashes are detected by ONCE, which is to be expected as it injects by far the fewest number of faults. When we divide the number of crashes by the number of injected faults, we get a crash/fault ratio of 5% for ONCE, whereas the ratio for ALWAYS is below 0.1%. This seems to indicate, that the initial calls are usually less resilient than subsequent calls.

Next, let us look at the *Total calls* column of Table 3.2. While Microsoft OneNote Launcher has similar call counts for all strategies, the number of calls varies largely between different injection strategies for other applications. If we look at Microsoft Publisher as an example, there are only 1883 calls to functions for which we provide a rule when applying the ALWAYS strategy. However, for the ONCE strategy, we get 21522 calls, which is more than eleven times the number of calls. On

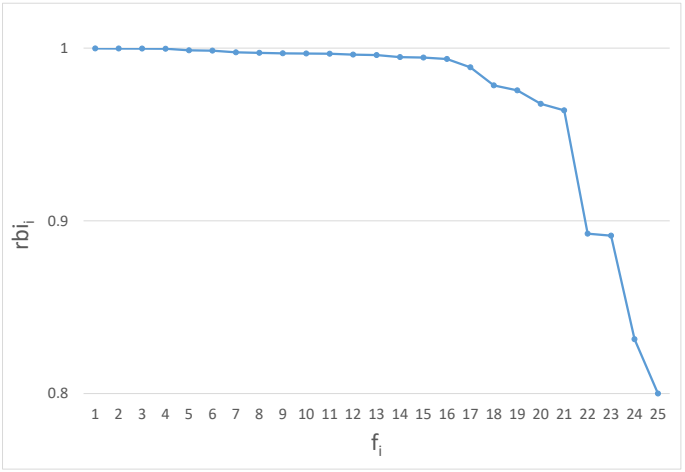

Figure 3.2: Indicators rbi_i that fault model f_i finds real bugs.

the other hand, Microsoft Notepad reacts in an opposite way, where Always has a larger number of calls than any other injection strategy.

These numbers imply that different injection strategies force the application to take different paths during its execution. For instance, an application might be executing some error recovery code if a certain operation fails or might cancel an operation prematurely because of an injected fault. Regardless of the concrete deviations to the execution path, we have thousands of calls to instrumented functions, indicating that we are not testing shallow execution paths, an exception being Microsoft Paint and Microsoft OneNote Launcher.

If we compare the different injection strategies, we can see that Always has found the most crashes. However, it is still beneficial to combine multiple different strategies, as they find different crashes. This is because of the different paths that the execution can take encountering different, potentially buggy call sites.

3.3.3 Crashes

As stated in objective (6), we wanted to determine if the crashes that we found are unique. Since the uniqueness of a crash is hard to determine using only an automated analysis, we inspected all the crashes in two applications. We picked Microsoft Notepad and Microsoft Visio at random.

For Microsoft Notepad we recorded eight crashes in total. After manual inspection of the crash sites with a debugger, we found that six of them are unique. The remaining two crash sites were duplicates found by the EveryOtherCall and FiftyFifty strategy. This is to be expected, as the FiftyFifty strategy can randomly inject the same pattern as EveryOtherCall. For each specific injection strategy, we found no duplicate crashes. This is also to be expected, as we only apply one rule per test run and all rules with the same injection strategy modify the behavior of different functions.

For Microsoft Visio the total number of crashes is 23. Again, by manually inspecting the crash sites in a debugger, we found two pairs of duplicates. Same as the situation for Microsoft Notepad,

both duplicates were introduced by the FiftyFifty strategy, once by having the same pattern as Always and once by having the same pattern as EveryOtherCall. Again, all crash sites detected by a single injection strategy were unique.

As explained in Section 3.2.5, we have also performed test runs with groups of functions. The results are not shown in Table 3.2 as we did not detect any additional crashes. All crashes found by the grouped configuration could also be found by only injecting faults into a single function.

The data in the *Crashes* column combined with the *Total calls* and *Injected faults* columns could be considered as a health index, which compares the resilience of two applications for a given set of rules. When we look at the figures for Microsoft Notepad and Microsoft Excel, Notepad is more resilient to our set of rules than Excel. Despite the lower number of injected faults for Excel, there are more crashes than for Notepad.

3.3.4 Example

In this section, we look at a concrete crash site that we discovered using *Brute*. As the running example, we take a crash from Microsoft Excel, which is related to concurrent access to the registry. The *Windows Registry* can be viewed as a concurrent data structure that is shared between the system and all running processes. Its predominant use is the storage of configuration data. While the *Registry* is primarily used by the operating system, many application also use it to store their configuration settings.

The *Registry*'s hierarchical structure resembles the one of a file system, with the difference that directories (i.e. nodes that can contain other elements) are called *keys* and instead of files they store name-value pairs. The value of a registry key is typed (e.g. REG_DWORD for a 32-bit integer or REG_SZ for a zero-terminated string). If the type of the registry key allows it, the length of a registry value is unbounded. Because of that, an operation that reads from the registry can require multiple attempts. Since the user needs to provide a buffer of sufficient size to store the value, which can in general be of an unbounded size, the caller cannot allocate a static buffer, such that the read operation will always succeed. Should the value of a registry key change during two read attempts, it could even be necessary to make multiple read attempts and resize the buffer accordingly after every failed attempt.

Listing 3.1 shows the signature of the function `RegQueryValueExW` in `ADVAPI32.dll`, which reads a value from the *Registry*. The first (`hKey`) and second (`lpValueName`) parameter identify the value to read. The `lpType` out parameter is set to the type of the registry value and the value itself is copied into the buffer `lpData`. The last parameter `lpcbData` must be initialized with the length of the buffer and will be updated to contain the actual length of the value. If the buffer was large enough to hold the value and no other errors occurred, the function returns `ERROR_SUCCESS`. If the buffer was too small, `ERROR_MORE_DATA` is returned instead.

```
LONG WINAPI RegQueryValueExW(
  _In_         HKEY    hKey,
  _In_opt_     LPCTSTR lpValueName,
  _Reserved_   LPDWORD lpReserved,
  _Out_opt_    LPDWORD lpType,
  _Out_opt_    LPBYTE  lpData,
  _Inout_opt_  LPDWORD lpcbData
);
```

Listing 3.1: Signature of `RegQueryValueExW`

Now that we are familiar with the interface for reading registry values, let us focus on the rule

that reveals the bug in Microsoft Excel. The rule is shown in Listing 3.2 and simulates a concurrent modification of a registry value. On the first two lines we declare two thread-static variables. On line 1 we declare `last_key`, which stores the handle of the last accessed registry key on this thread. The counter declared on line 2 stores the number of repeated calls to `RegQueryValueExW` with the same registry key.

Inside the *before action*, we check if the registry key has changed since the last read attempt (line 8). If yes, we reset the counter and store the new registry key handle (lines 9 and 10). Note that we do not store and compare the name of the actual value. While this is possible and would increase the strictness of this rule, it was not necessary to reveal the bug in question and was therefore left out for simplicity reasons. If the access is to the same registry key, we increment the `counter` (line 12). In the *after action* we check if the counter is less than 2, i.e. the function has been called less than twice for the same key. If that is the case, we inject the `ERROR_MORE_DATA` return value, which indicates that the provided buffer was too small. That only the third read attempt can succeed is motivated by the following reasoning: on the first call, the caller cannot know the size of the value, i.e. the buffer can be too small. On the second call, we simulate the concurrent modification, rendering the buffer too small, again.

```
1  thread_static last_key -> void*;
2  thread_static counter -> int;
3
4  rule ADVAPI32.dll!RegQueryValueExW(hKey, lpValueName, lpReserved, lpType,
       lpData, lpcbData)
5    include recursive;
6    frequency always;
7    before {
8      if (last_key != hKey) {
9        last_key = hKey;
10       counter = 0;
11     } else {
12       counter++;
13     }
14   }
15   after {
16     if (counter < 2) {
17       last_win32_error = ERROR_MORE_DATA;
18       result = ERROR_MORE_DATA;
19     }
20   }
```

Listing 3.2: Rule for simulating concurrent registry access.

Listing 3.3 contains a snippet of the call site, which is not resilient to this type of deviant behavior. In general, this call site is fairly robust, in that it handles many of the possible pitfalls correctly. It deals with all the different error codes that `RegQueryValueExW` can return (line 8) and also resizes the buffer, should the value exceed its size (line 7). However, we can see that this is only attempted twice (line 2), which is the minimum number of attempts required for a successful read, given that we do not know the size of the value at the beginning. When we inject the rule described above, we run through the two iterations and exit the loop. After the loop, we hit an assertion (line 10), which states that the read operation was successful. This indicates that the developer did not think of this scenario and the program is now in an unexpected state. While a violated assertion would usually terminate the process, it is not present in the code in

release builds for performance reasons. At a later point in the function, the program tries to dereference a pointer, which it assumes was initialized on line 5. However, this assignment has never occurred, thus the program is reading from an uninitialized pointer.

```
1  // [...]
2  for (int cRetry = 0; cRetry < 2; cRetry++) {
3    lError = RegQueryValueExW(key, [...], buffer, &buffer_size);
4    if (ERROR_SUCCESS == lError) {
5      value_ptr = buffer; break;
6    }
7    else if (ERROR_MORE_DATA == lError) buffer.resize(buffer_size)
8    else return INSTALLSTATE_BADCONFIG; // unknown registry error
9  }
10 Assert(lError == ERROR_SUCCESS); // debug build only
11 // [...]
12 while (*value_ptr++);
```

Listing 3.3: Snippet from Microsoft Excel which is vulnerable to concurrent registry modifications.

This example illustrates a rule, which injects a fault that is handled by most call sites, but not the specific one that we examined here. Since there exist other correct call sites, it would be possible to propose code fixes from these other call sites to the developer.

Chapter 4

Related Work

As discussed in the introduction, the standard way of dealing with external resources or operations that do not only depend on test inputs is to replace them with a stub (or mock). The behavior of the stub can be determined by the testing harness and thus, reproducibility of the test is guaranteed. By combining this functionality with the need for testing exceptional program states and language-independence, *Brute* positions itself between the classic fault injection and mocking frameworks.

Traditional *software implemented fault injection* (SWIFI[16]) can be structured into three groups[5]. The *data error category* contains tools like FERRARI[10], GOOFI[1, 25] or Xception[2], which focus on low-level data corruption, such as bit-flips in memory or the malfunctioning of a processor component like a faulty ALU. A second category of tools like DEFINE[11] and G-SWFIT[6] inject *code changes*, either by simulating faulty instruction decoding or modifying code according to common error patterns. Finally, the *interface errors category* focuses on corrupting values that are passed between two modules (e.g. a library and an application). However, the most prominent tools, like Jaca[19], MAFALDA[24] and BALLISTA[13] often focus on the correctness and resilience of the callee (i.e. the library or kernel in case of a system call), instead of the caller. While *Brute* fits best into the *interface error category*, our focus is on the caller and not limited to robustness testing. Another fundamental difference is that the mentioned tools do not deal with false positives, as they assume that arbitrary memory or program changes can occur. In contrast to this, we assume that the hardware and libraries work as expected and that the crashes we detect are the result of the application not handling the results correctly (in both, the successful or error case). This decision is supported by the results of Natella et al. [22], claiming that up to 72% of faults, which are injected by a state-of-the-art approach (making the former assumption) are not representative of residual software bugs. Furthermore, an empirical study by R. Moraes et al. [20] suggests, that the corruption of values at interface boundaries does not represent the same class of errors as implementation errors in the components themselves (like, for example, faulty error handling code). The LFI framework[18, 17] also focuses on the correctness of the caller, but is limited to fault injection and does not try to avoid false positives as strictly as we do, as injected faults cannot generally be dependent on the function arguments.

Mocking frameworks[15] like JMock[7], EasyMock[8] and Moq[23] simplify typical mocking scenarios. However, these framework are usually restricted to a single programming language. Furthermore, they are often limited to dynamically dispatched functions (i.e. virtual or abstract functions) and cannot instrument statically bound functions.

There exist multiple tools for performing the actual stub injection and usually, they are specific to a runtime environment. Detours[9], the library we are using for native stub injection

internally, allows to intercept arbitrary function calls by replacing the beginning of the callee with an unconditional jump to a user-specified replacement. The original function can still be called as part of the replacement function, thus providing a flexible way to alter the behavior of any function with a well-known address. As we have established, hooking a function with Detours usually requires knowledge of its signature, which is why we extend the Detours hooking mechanism with generic hooks (see Section 2.4). For managed code, there exists Moles[27], which allows to replace any managed function with a different function. Similar tools exist for Java, for example Javassist[4], which allows the generation of types dynamically at runtime and was used by Jaca[19] to implement its instrumentation.

Chapter 5

Conclusion

In the course of this thesis, we have developed a framework that allows alterations to the behavior of managed and unmanaged functions in a generic way. While we initially focused on fault injection, the resulting framework targets a broader spectrum of applications, such as mocking and test isolation. The main reason for this high degree of flexibility can be found in the presented *fault modeling language*, which allows to execute arbitrary code before and after any DLL-exported function or any managed function. This moves *Brute* away from the classic fault injection scenario and positions it closer to a multi-language aspect weaver[12]. We accomplish this without requiring access to the source code and hide much of the implementation complexity associated with this type of instrumentation, providing the developer with an easy-to-use solution.

The evaluation reveals, that even simple fault models can find crashes in well-tested, heavily used applications. We have found that a combination of multiple different injection strategies can be beneficial to find different defective call sites, as the program is forced to take different paths depending on the injected faults. Finally, the only false positives that we found were caused by unrealistic or overly aggressive fault models.

5.1 Future Work

Brute provides a basis for various different types of future works. The framework itself could be improved in different aspects and its applicability widened by implementing the following extensions:

The DSL could be extended with the elements from C that are currently missing to increase the usability of the language. Furthermore, the translation to IL does not support many of the constructs that are supported for unmanaged code. Regarding type information, the architecture would allow to support different formats of debugging symbols to be imported. However, at the moment we only support PDB/CodeView, which limits the framework to importing type information from projects compiled with the *Microsoft Visual C++* compiler. Reading type information from other formats, like DWARF, or by parsing header files directly would allow *Brute* to be used with other compilers as well. With support for managed code, we have shown how we can transfer the ideas originally developed for native code to a new execution environment. However, the CLR only represents a single alternative and porting the same ideas to other virtual machines, like the JVM, could further expand the applicability of the framework. Adding an API that allows to enable or disable rules at runtime would increase the usability and integration of *Brute* with existing unit test suites, allowing to write test cases for deviant behavior directly embedded into the existing unit tests. Finally, many of the ideas covered in this thesis could also

be transferred to other operating systems and adding portability would present an interesting problem.

We employed very simple fault models for the evaluation. However, by combining *Brute* with other tools, much more intricate fault models could be written. As an example, an existing fuzzing framework could be adapted to not only fuzz e.g. input files but also data from sockets or the registry. By performing lightweight static analysis of the injection call sites, error codes that are handled specially could be extracted automatically to improve the coverage of error handling code. By fully replacing an API (e.g. file API, socket API, etc.), it would be possible to provide test isolation that is not specific to the application under test, reducing the otherwise required development effort. As mentioned earlier in the thesis, it would also be interesting to propose code fixes based on a database that contains the resilience functions to certain rules. Concretely, when we have a function that does not handle a certain fault but we know of multiple other call sites that do, one could try to either display the code which handles the fault as a proposal for a fix or even try to transfer the error handling code automatically.

References

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: generic object-oriented fault injection tool. In *Dependable Systems and Networks, 2003*, pages 668–668, June 2003. doi:10.1109/DSN.2003.1209977.

[2] J. Carreira, H. Madeira, and J. G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, Feb 1998. doi:10.1109/32.666826.

[3] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP*, volume 2374 of *LNCS*, pages 231–255. Springer, 2002.

[4] S. Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998. URL: http://www.javassist.org.

[5] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. Experimental analysis of binary-level software fault injection in complex software. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 162–172, May 2012. doi:10.1109/EDCC.2012.12.

[6] J. A. Durães and H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, Nov 2006. doi:10.1109/TSE.2006.113.

[7] S. Freeman, T. Mackinnon, N. Pryce, M. Talevi, and J. Walnes. JMock, 1999. URL: http://www.jmock.org/.

[8] T. Freese. EasyMock, 2001. URL: http://easymock.org/.

[9] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Usenix Windows NT Symposium*, 1999.

[10] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, Feb 1995. doi:10.1109/12.364536.

[11] W.-L. Kao and R. K. Iyer. DEFINE: a distributed fault injection and monitoring environment. In *Fault-Tolerant Parallel and Distributed Systems, 1994., Proceedings of IEEE Workshop on*, pages 252–259, Jun 1994. doi:10.1109/FTPDS.1994.494497.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. doi:10.1007/BFb0053381.

[13] P. Koopman and J. DeVale. The exception handling effectiveness of posix operating systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, Sep 2000. doi:10.1109/32.877845.

[14] B. Laing. Summary of windows azure service disruption on feb 29th, 2012. URL: https://azure.microsoft.com/en-us/blog/summary-of-windows-azure-service-disruption-on-feb-29th-2012/.

[15] T. Mackinnon, S. Freeman, and P. Craig. Extreme programming examined. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, chapter Endo-testing: Unit Testing with Mock Objects, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *Dependable Systems and Networks, 2000*, pages 417–426, 2000. doi:10.1109/ICDSN.2000.857571.

[17] P. Marinescu, R. Banabic, and G. Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the USENIX Annual Technical Conference*, 2010.

[18] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 379–388, June 2009. doi:10.1109/DSN.2009.5270313.

[19] E. Martins, C. M. F. Rubira, and N. G. M. Leme. Jaca: a reflective fault injection tool based on patterns. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 483–487, 2002. doi:10.1109/DSN.2002.1028934.

[20] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira. Injection of faults at component interfaces and inside the component code: are they equivalent? In *2006 Sixth European Dependable Computing Conference*, pages 53–64, Oct 2006. doi:10.1109/EDCC.2006.16.

[21] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[22] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, Jan 2013. doi:10.1109/TSE.2011.124.

[23] K. Pederson, D. Cazzulino, and T. Kellogg. Moq, 2012. URL: https://github.com/moq/moq4.

[24] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat. *Dependable Computing — EDCC-3: Third European Dependable Computing Conference Prague, Czech Republic, September 15–17, 1999 Proceedings*, chapter MAFALDA: Microkernel Assessment by Fault Injection and Design Aid, pages 143–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. URL: http://dx.doi.org/10.1007/3-540-48254-7_11, doi:10.1007/3-540-48254-7_11.

[25] D. Skarin, R. Barbosa, and J. Karlsson. Goofi-2: A tool for experimental dependability assessment. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 557–562, June 2010. doi:10.1109/DSN.2010.5544265.

[26] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[27] N. Tillmann and P. de Halleux. Moles: tool-assisted environment isolation with closures. In *TOOLS'10*. Springer Verlag, July 2010.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Testing program robustness against deviant behavior

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Emmisberger	Patrick

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 16.12.2016	P. Emmisberger

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.