Universiteit Leiden
The Netherlands

Opleiding Informatica

Hiding in plain sight:

how location affects memory error detectability by fuzzers

Vincent den Hamer

Supervisors:
Dr. E. van der Kouwe & Dr. K.F.D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/01/2019

Abstract

Fuzzing, a testing technique that automatically generates new test cases, has gained a significant amount of attention in recent years and has been successfully used to find security bugs in programs. However, the influence of the location of a bug in a program on the probability of it being found by a fuzzer is not well understood.

To investigate this probability, we employ a novel framework that can automatically inject arbitrary memory errors at different locations of an existing C program. Subsequently, the modified program is fuzzed using two different kinds of fuzzers, and the effectiveness of the fuzzer is correlated with the number of times the encapsulating function of the bug is executed. Besides injecting memory errors, the framework also features optimizations for the initial input seed to aid the fuzzers performance and furthermore employs strict checks to prevent existing bugs from influencing the final result.

When using an optimized version of the Bash regression tests as a seed for the fuzzers, we found that the location of the memory error with respect to the number of times the function containing the error is executed in the initial seed has an effect on the fuzzer finding the error. We also observed that there was no significant difference between the two fuzzers we tested.

Contents

Chapter 1

Introduction

Fuzzing, an automated way of generating new (random) test cases as input to a program, is used by many large software companies [GLM12] to aid their testing process of programs. A fuzzer, the application that fuzzes the program, can help to generate many negatives and boundary test cases that generally would be infeasible to generate manually. Instead of solely helping with finding normal programming bugs, a fuzzer can also help to find memory errors, an error that occurs when an invalid pointer is dereferenced. In a system language such as C and C++, memory errors are a large problem because the programmer can directly manage memory allocations. A memory error can have serious consequences. For example, an attacker could use the memory error to crash the application creating a denial of service. The attacker could also sometimes use a memory error to change the logic of the program or execute code the attacker has supplied to the application. This could lead to serious security issues if for example the application runs with higher privileges or has access to sensitive files.

In this thesis, we investigate if the location of memory errors in the program and type of fuzzer influences the probability of finding a memory error. We investigate this by building a framework that injects memory errors in random locations of a program and after that runs a fuzzer on it.

Testing a fuzzer can be a difficult task. Fuzzers are usually tested by the number of crashes they find on a certain application. In most research, the errors that are causing the crashes are inserted into an existing application. The problem with inserting errors into real-world programs is that often there are already existing memory errors in the unmodified code that could crash the application leading to false positives. Another problem is that errors could be injected into parts of the program that are never executed and thus that the fuzzer could never find the error.

Besides the issues with injecting errors into an existing program, there are also problems with the data that is given to the fuzzer about the program. One of such problem is that many fuzzers depend on the quality of the initial inputs they are given. If the initial inputs do not cover much of the program or contain a lot of duplicate coverage, it could make the process of finding the injected errors nearly infeasible or be slowed down significantly.

In this research, we try to minimize some of these problems by checking the crashing inputs the fuzzer generated on an unmodified copy of the program, injecting memory errors only in reachable code, and providing minimized initial input files to the fuzzer.

1.1 Contributions

We make the following contributions:

- We present an easy to extend open source framework that can inject a memory error in a random location in any C application.
- We evaluate some of our results based on executing the program instead of static analysis. We check if the number of function calls of a function has an influence on the chance of finding memory errors with a fuzzer.
- We run our evaluation on 64-bit applications where other research only focuses on 32-bit applications.

1.2 Thesis Overview

In this thesis, we will investigate if the location of a memory error has an influence on the ability of the fuzzer to find the error. We will first give a background on fuzzing and memory errors in chapter 2. Then we will briefly explain the framework that injects the memory errors in chapter 3. Following that, we will explain how the framework was designed and discuss the types of memory error injections that are implemented in the framework in chapter 4. An in-depth explanation of how the framework was implemented is given in chapter 5. Then in chapter 6, we benchmark a greybox and blackbox fuzzer with different error injection locations and review afterwards why some memory errors were more likely to found than others. In chapter 7, we discuss related work. Finally, in chapter 8 we conclude our findings and give recommendations for future work.

Chapter 2

Background

In this thesis, we will investigate the effectiveness of fuzzers on programs in which several different types of memory errors have been injected. We will now first give a concise background on fuzzing and memory errors, before describing the overview and implementation of the memory error injection framework.

2.1 Fuzzing

Fuzzing [Oeh05] is an automated software testing technique that focuses on testing boundary cases of applications by most of the time using invalid (random) data. This generated invalid data is often semi-valid meaning that it is based on valid data but with a few mutations so that it does not get immediately rejected by an application as bad input. It will also sometimes use valid data if this increases the code coverage.

The fuzzer, the tool that fuzzes an application, works by having a feedback loop. It first starts by generating data (in some specified format) or by mutating data from an existing seed that consist of (valid) data. The next step after getting the data is to send it to the application and monitor the application for changes. This monitoring is then used to create new inputs for the application so that this process can start again.

The way of monitoring and generating new inputs depends on the knowledge that the fuzzer has of the target application. A fuzzer be either a blackbox, whitebox or greybox type of fuzzer. A blackbox fuzzer has no knowledge of the code of the application and can only see simple things such as if the application crashes. In contrast to the blackbox fuzzer, the whitebox fuzzer [GLM+08] has access to the source code of the application and uses symbolic execution to analyze constraints on inputs. These constraints are solved using a constraint solver to create new inputs so that a higher coverage of the program is reached. Greybox fuzzers such as AFL [Zala] and VUzzer [RJK+17] are using some techniques of whitebox fuzzing such as employing analysis on the source code of the program but usually only for simpler purposes such as monitoring code coverage of the program. This code coverage tracking by greybox fuzzers is usually done by instrumenting comparison or branch instructions in the assembly code.

2.2 Memory errors

In system languages such as C or C++, the programmer has full control over the memory management. This makes it easy for the programmer to make a mistake and introduce a memory error which can be dangerous. Memory errors can for example be used by attackers to leak information, crash an application or even worse take full control over an application.

A memory (access) error occurs when dereferencing a pointer that is not pointing to the original object it belongs to. Memory errors can be categorized [ABS94], [SPWS13] as spatial memory errors and temporal memory errors.

A spatial memory error occurs when a pointer is dereferenced outside the address bounds of the object it points to. A simple example of this would be a buffer overflow where it is possible to write outside the bounds of an array. Spatial memory errors can have serious consequences. For example, an attacker could overwrite a variable to alter the behavior of the application. Another possibility would be that the attacker overwrites the return address or a function pointer and can therefore direct the application to run code at the address that attacker specified. This can then be used to jump to an attacker created shellcode, to the c library, or be used for more advanced techniques such as return-oriented programming.

A temporal error occurs when a pointer is dereferenced outside the lifetime of the object it points to. In other words, using a pointer from an object that does not exist anymore. These types of pointers are often called dangling pointers. Common examples are the use-after-free and double-free vulnerabilities.

In use-after-free, an object is for example deleted using a `free` or `delete` call, and then a pointer that points to that deleted object is used again without assigning a new object to the pointer. A use-after-free could, for example, be exploited when a part of the freed data is allocated to a new pointer that is under the control of an attacker. Using this new pointer, the attacker could overwrite data that belongs to the freed pointer and when the program then uses the freed pointer the data is not the same as before.

A double-free is a specialized case of use-after-free. Instead of using directly using a dangling pointer, `free` or `delete` is called again on the pointer and therefore freeing memory that does not belong anymore to the pointer.

.

Chapter 3

Overview

In this chapter, we briefly explain how the framework injects a memory error into a program, runs a fuzzer and collects the results. In order to use the developed framework, the user has to supply one or multiple C source files so that an error can be injected during the compilation process of the program. Furthermore, the user needs supply test cases so that the fuzzer performances better. After that, an unmodified compiled copy of the program is needed to differentiate injected memory errors that are found from memory errors that already exist in the program.

As seen in figure 3.1, the first step is performed by the front end of a compiler. The front end takes the source files and converts them into an Intermediate Representation (IR) form, a type of code that is independent of the source code language and is used internally by the compiler. The next step is to link all IR files into a single IR file to make further analysis easier. After the analysis is complete, we inject a memory error and pass the modified code to the back end of a compiler. The back end then generates an instrumented binary so that the fuzzer can use it for the coverage tracking. The fuzzer takes the binary it received from the back end and also takes the optimized test cases it receives from the seed optimizer. The fuzzer is then executed for a certain amount of time and when the time is elapsed the crashes are sent to the crash analyzer.

The goal of the crash analyzer is to verify that the crashing input triggers the injected error and not an error that already exists in the target program. The crash analyzer accomplishes by checking if the input does not crash on an unmodified copy of the program and uses a profiler to check if the function where the error is injected is reached.

The framework that this thesis uses was built in collaboration with Vincent van Rijn and Wampie Driessen. For a detailed overview of the author's contribution to the framework, the reader is referred to Appendix A.

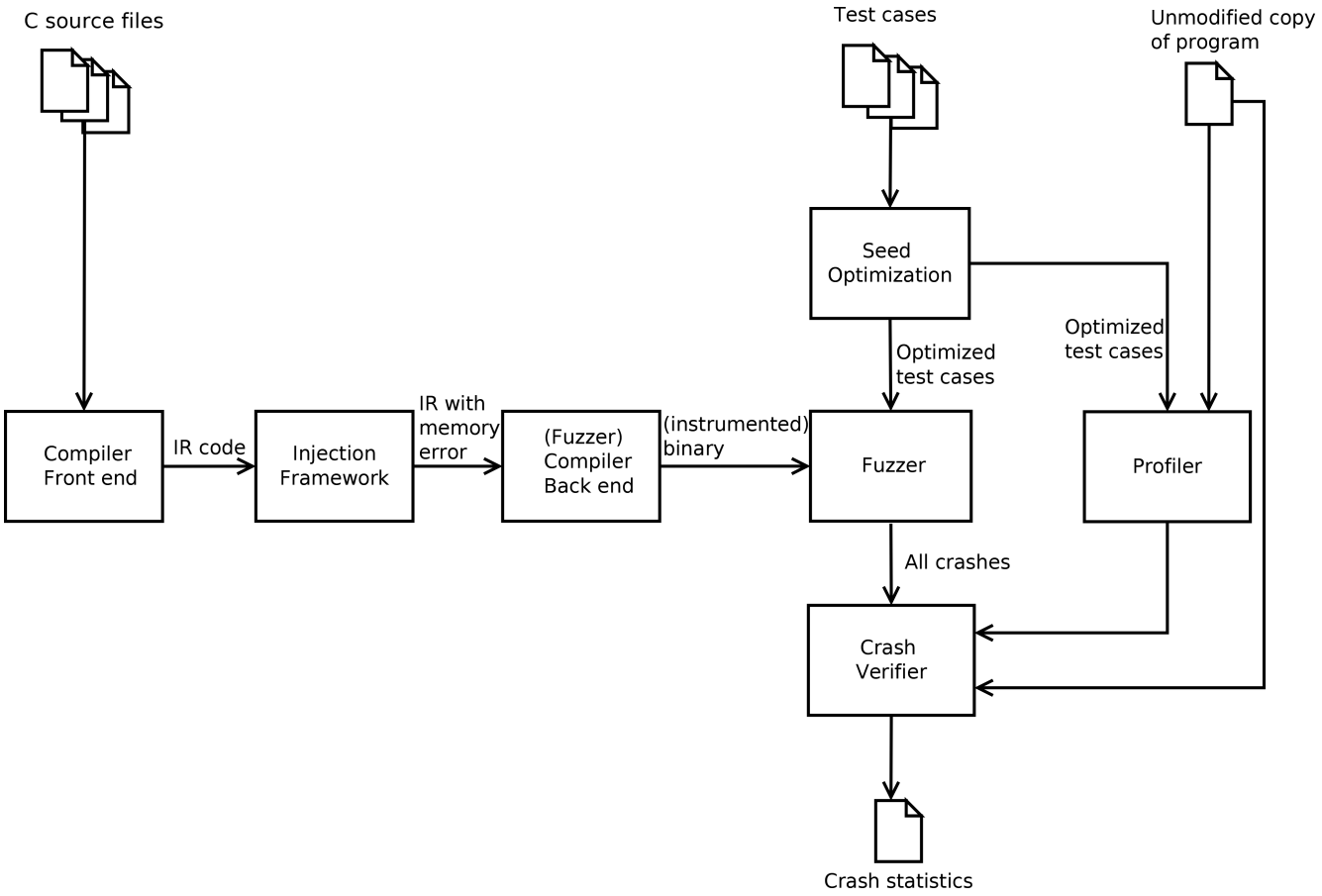

Figure 3.1: Overview of the framework

Chapter 4

Design

As mentioned in the previous chapter, our framework is capable of injecting memory errors in existing code that could be exploited by a user of the program. When designing the framework that is going to inject memory errors into a program we used the following criteria:

- Triggering the memory error should depend on input from the user of the program. Otherwise, the fuzzer has no influence on the triggering of the memory error.
- The memory error should not be injected into unreachable code. If the injected code was not reachable, the fuzzer could never exploit the memory error.
- The injected code needs to represent a memory error that programmer would likely make.
- The program that contains an injected memory error should be performing the same operation so that the application does not crash due to modification for other reasons than our possible memory error.
- When exploiting the memory error, the fuzzer only needs to crash the application as the scope of our research is only to find memory errors.

4.1 Components of the framework

In this section, we describe the design of the framework. The framework could be divided into two parts: the injection part of the framework that has the responsibility to inject the right memory error into the target program and another part of the framework that manages all the necessary input for the fuzzer and parses the results of the fuzzer.

4.1.1 Injection framework

The injection part of the framework, figure 4.1, starts with the component that eliminates unreachable and dead code before searching for possible places to inject memory errors. We designed this part to prevent injecting possible memory errors in places that could never be reached by a fuzzer. It also prevents that later compiler optimization removes the code where the memory error is located.

The next component in our framework is the memory error analyzer component. This component analyzes a given instruction of the target program for the suitability as an injection target. The component was designed from the ground up to recognize if there are multiple types of memory errors injections possible in one instruction. This was needed because in our selection of memory errors there are some memory errors that share the same function call before modification. For example, a `strncpy` is in our framework a function call that can be converted to be an off-by-one type of error or be an unsafe C type when we convert it to `strcpy`.

This component only analyzes if there are one or more errors injection possible and returns them instead of directly modifying the instruction. This was done so that another component can make a fair random selection over all possible memory errors in the whole target program. The component also gives the same type of interface back for all types of memory errors so that our framework can later be easily extended for new types of memory errors.

Another component of our framework is the selection component. This component iterates over the whole program and collects the possible memory errors injection locations from the memory error analyzer component. After that is completed, it will filter out certain types of memory errors injections or other criteria depending on what the user of the framework requested. Following that a random selection over the remaining possible memory errors injections takes place. We then take the instruction that belongs to the memory error that is selected and remove all possible memory errors injections with that same instruction from the list. This step is essential as it prevents multiple injections based on the same instruction. This process occurs multiples times if the user has requested for more than one memory error injection. Once this step is completed, the component returns the list of randomly selected memory errors to be injected and sends it to the injection component.

The injection component receives the list of memory errors that have to be injected from the selection component. It first adds the new modified instruction, updates values that depend on the old instruction and then removes the old instruction. After this step one or more memory errors are now injected into the program, and the program is now ready to be analyzed by the fuzzer.

4.1.2 Framework connecting with the fuzzer

This part starts with the component that connects the injection framework to the fuzzer toolchain. This component takes the program with our modifications (in Intermediate Representation form) to the fuzzer toolchain. Then the component will instruct the fuzzer toolchain to instrument the code if necessary and

afterwards directs the fuzzer to run on our modified code with the initial seed of test cases we get from our seed selection component.

The next component is the seed selection component. This component optimizes the seed files before giving them to the fuzzer. The selection of which test cases are included in the seeds is generated by using a size minset seed selection algorithm, as described in [RCA+14], on the supplied test cases. This algorithm takes a set of seed files together with their code block coverage and then tries to solve the weighted minimal set cover problem. This means the algorithm creates a minimum set out of the seed files so that the set contains the fewest number of seed files that still covers the same code blocks of the complete set. Because the set is weighted on size, this process prefers smaller seed files over larger ones. This minset process essentially eliminates redundant seed files and could increase the likelihood or speed up the process that the fuzzer finds a memory error.

The last component of the framework is the component that collects and processes the results. When the fuzzer is done, this component verifies the crashing inputs that the fuzzer has generated. We verify the results because a memory error that a fuzzer has found could potentially be a bug in the original program, which could negatively affect the correctness of our results. The verification step is done by running all the crashing inputs generated by the fuzzer again on an unmodified copy of the program.

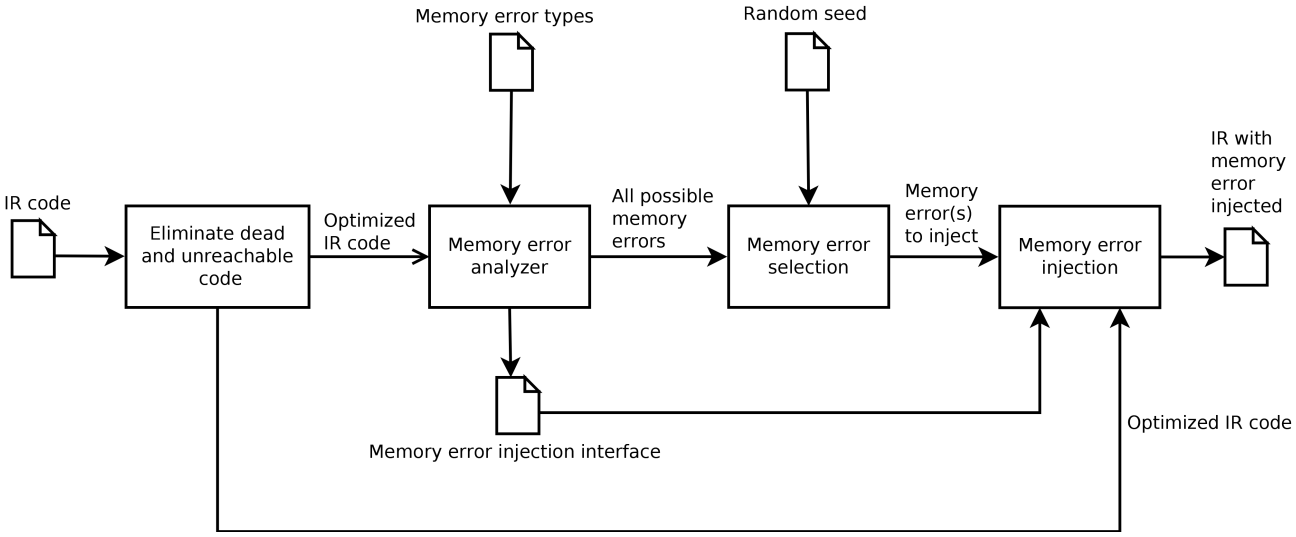

Figure 4.1: Design of the injection framework

4.2 Memory errors

Just like [SPWS13], we divided the memory errors into two main categories: spatial and temporal memory errors.

4.2.1 Spatial errors

The spatial memory errors in our framework are divided into allocation errors, unsafe C library functions, off-by-one errors and format string errors. Below we explain for each category how we designed the modification, why we chose it and what the limitations are.

In the allocation error category, we try to create a possible buffer overflow on the stack and heap by making the size of the to be allocated buffer smaller than was initially intended. This modification could then be exploited because if there is a memory allocation next to this allocation, we could overflow it. We think that this error represents an error that a programmer could make because in C it is possible to specify the size of the memory allocation manually and when the size is incorrect, the compiler will most of the time not give a warning about it.

For the stack allocation, we replace the size argument in the call to `alloca`, the function that allocates memory on the stack. There are some limitations on replacing the size because often the size requested is not the same as the size of the memory that is reserved. This is because often the stack is aligned to certain 2^n and because we test on a 64-bit platform, the alignment is 16 bits. This could lead to that some of our modifications could have no effect as we have no checks in place to detect this. Future work could improve upon this.

For heap allocation we, replace the size parameter in the call to `malloc`, one of the C functions that allocates memory on the heap. We have chosen only for `malloc` and have not implemented the other C heap allocation functions because we find that there are not enough differences between them and have instead chosen to work on other types of memory errors. Just as with the stack allocation, the allocation on the heap with `malloc` is also aligned, potentially making some of our modifications have no effect.

In the unsafe C library category, we try to create a possible buffer overflow by replacing some C functions with other C functions that are generally known to be unsafe when misused. When we chose the functions to be replaced, we took only functions that would not change the logic of the program, only that it could overflow. We find that using unsafe C functions is a reasonable mistake that a programmer could make. While there are some functions such as `gets` were the compiler warns the programmer that the function is unsafe, it does not warn for others such as `strcpy`. However, even if the compiler would warn for all possible unsafe functions, there is still legacy code that contains unsafe functions.

The first function we implemented is the conversion from the `strncpy` function call to `strcpy`. `strcpy` does the same operation as `strncpy`, namely copying a string from one buffer to another until "\0" is hit, but `strcpy` has no parameter where the maximum amount of chars it is copying can be set. This could lead to an overflow if the destination buffer is smaller than the source buffer. We implemented this conversion by copying the `dst` and `src` parameters from `strncpy` to `strcpy` and leaving the size parameter out.

In the off-by-one type of error, we create a possible memory error by subtracting or adding 1 to the size argument of various C library function calls and stop conditions of loops. An off by one error could cause a small overflow, but that can still cause a problem when for example the least significant byte of a pointer is overwritten. We think that this memory error is an error that a programmer would likely make for example an

easy of by off-by-one error is to make a <= instead of a < comparison in a stop condition of a loop. There are some limitations in the way we implemented this type of error. The effect could be negated in the same way as with our allocation injections due to alignment or for example, only cause an out of bound read that would not crash the application.

For the C library functions, we have chosen the `strncpy`, `strncat` and `fgets` functions and adjusted their size parameter by one. As mentioned above, these functions could cause a small overflow with size +1. However, for the `strncpy` function, we have chosen -1. With the size - 1 in `strncpy` another interesting case could happen. When we copy one character less than the original size, the null terminator could be lost. This could lead to an overflow if another function expects a null-terminated string. The limitation being that after the `strncpy` there must be another function that uses the destination buffer otherwise the modification will have no effect.

For creating off-by-one errors in a loop, we increment the stop condition of the loop by one. In order to create a possible memory error, we cannot just adjust a random loop. We first filter out loops that have a stop condition with a constant value of -1,0,+1. This will prevent destroying the logic of some loops that depend on a return of functions such as `strcmp` in simple while loops. In the next phase we filter out loops where adjusting them probably does not result in a memory error. We first check if the loop has an induction variable, a variable that is always incremented in every iteration of a loop, we then check if the induction variable is used in the body of the loop and as the last step we check if that usage creates a pointer. Now we know that there is a value that is incremented in every iteration and that value is used to calculate some memory address. If we then adjust the stop condition, that value increments one extra time possibly creating an out of bound memory address that could be exploited.

In the format string error category, we try to create possible memory errors by modifying function calls to make them vulnerable for format string memory errors. A format string vulnerability [LC03] is a vulnerability that is created when a format string argument in a call to a function is not static and can be influenced in some way by external input. It can cause reads from and writes to arbitrary memory locations. We think that this is still a reasonable memory error to include in our research, because even though a modern compiler warns about the error, in 2018 there are still format string bugs being discovered in programs that are exploitable.

For the realization of the format string error, we have chosen the `printf` function. In calls to this function, we replace the constant format string specifier with one of the other arguments of the function call. To choose the right function argument to replace the original format string with, we first check if the argument is a string, so that it could represent a format string. After that, we check if the argument is not a constant so that input could modify it.

4.2.2 Temporal errors

For the temporal errors, a single category has been implemented in our framework: the use-after-free memory error. In this category, we try to create a use-after-free memory error by letting the program use an object that has already been freed. We inject this error by searching for a `malloc` call and then insert a `free` call right

after it so that any code that uses the object now uses a dangling pointer. Because this way of injecting can also cause a double free memory error, we try to remove all old free instructions that depend on this memory allocation.

The limitation of this design is that it might not be the most realistic approach as a programmer would likely not call `free` directly after a `malloc`. However, this design has the advantage that every usage is now a dangling pointer so that there is a larger probability that this memory error has an effect.

Chapter 5

Implementation

In this chapter, we go in-depth how some components of our framework that we explained in the design are implemented, explain some external tools we use and describe some challenges we faced during the implementation of the framework.

5.1 Integration into LLVM

For the implementation of the framework, we make use of the LLVM framework [LA04] by extending its functionality with an LLVM pass. A pass is a sort of plugin that runs during the compilation of a program. In a pass of a certain type, it is possible to analyze and edit the program.

The type of pass determines in which scope of the program the pass operates during the compilation. This is closely related to how LLVM structures a program into different parts. LLVM starts by dividing the program into modules for each source file. These modules are made up of functions, and these functions contain basic blocks. A basic block is made up of a series of instructions that are executed sequentially. In other words, only the last instruction of a basic block can be a branch instruction, and that branch can only jump to the first instruction of a basic block.

We use a module type of pass so that we can analyze and edit the whole program but because LLVM divides the program into multiple modules during the compilation we had some problems. It makes it more difficult to have a random selection over all the memory errors in the program and causes the pass to be executed multiple times making it more difficult to inject for example only one error. In order to solve these problems, we had to enable link time optimization so that our pass is being executed at linking time and therefore gets the complete program in one module.

5.2 Components of the injection framework

In this section, we describe how various components from the injection framework are implemented. Many of those components make extensively use of the functionality that LLVM framework provides.

We start with the unreachable and dead code elimination component. This component is implemented by running the LLVM unreachable block elimination pass and the aggressive dead code elimination pass on every function. We implemented this so that we do not inject memory errors into code that is never going to be executed or is going to be optimized away. We furthermore used the sparse conditional constant propagation pass, so that more values are being made constant. This helps other components in our framework eliminate some instructions that contain a constant value as possible memory errors because an input does not modify a constant value.

After the unreachable and dead code elimination component is finished the memory error analyzer and injection component are executed. Large parts of the memory error analyzer and injection component are implemented per memory error type to ensure the easy addition of new types. These memory error types share a common interface for other parts in the framework that do not need to know the exact type of memory error as each possible memory error in the program act as an object for the framework.

The main part of the analyzer works by iterating over each LLVM instruction in the program and collecting the possible memory errors. Each instruction is given to the analyze function of each memory error type that we implemented. This analyze function checks if a memory error of that type is possible and if so then collects all the information that is necessary to inject the error later on. Because of this implementation, it is possible that an instruction has more than one possible memory error.

For most types of memory errors, we start the analysis by checking if the instruction is a call instruction, as most error injections we implemented are C library functions. We then check if the name of the called function matches one of the functions we implemented in our framework. However, for loops, we employ a different strategy. First, we use LLVM LoopInfo class to determine the natural loops then we use the conical induction variable function from LLVM to check if a loop has an induction variable. After that, we check in all the usages of the induction variable if there is a `getelemptr` instruction, an instruction that calculates a memory address. If all those three checks are positive, we increment the right-hand side of the comparison instruction of the loop by one. This then causes the loop to iterate an extra time.

We implemented the memory injection component by having a set of shared functions and a set of specific functions for each type of memory error. This component receives the injectable memory errors objects it has to inject from the selection object. The injection component calls the inject method on all the objects. The inject method then calls the right functions for that type of memory error to create new instruction(s) that contain the memory error using the information that was recorded from the memory analyzer component.

The next thing that happens in the injection component is the replacement of values in instructions that depend on the value of the old original instruction without the memory error. For this step, we wanted to use the

built-in LLVM `replaceAllUsesWith` function, but our modification often changes the type of the values which conflicts with of the safety that LLVM IR tries to provide. We have therefore built our own function that after inserting an instruction, searches for instructions that use the value of the old unmodified instruction using the LLVM `uses` function. It then replaces the parameters in these instructions that depend on the value of the old instruction by the value of the newly inserted instruction and then removes the old instruction.

5.3 Framework connecting with the fuzzer

This part of the framework starts with the component that creates the initial seed for the fuzzer. It is implemented by using the AFL-cmin and AFL-tmin tools [Zalb]. The afl-cmin tool sorts the seed files by size and then eliminates seeds that do not add extra coverage. This essentially generates a smallest possible set of seeds that still have the same coverage of the program as the full set of seeds. Next, we run the afl-tmin over each of the remaining seed files. This tool minimizes each seed file by trying to zero blocks of data, removing blocks and replacing individual characters with '0' while still having the same execution path as the original test case. After all the optimization is done, the component removes some test cases which take much longer than average and also test cases that the fuzzer gets stuck on.

In building the component that connects the injecting framework to the fuzzer we encountered some problems and had to build some workarounds.

The first problem was that the LLVM framework preferred the built-in assembler over the AFL assembler, which prevented the application from being instrumented. The first thing we tried to solve this, was running the fuzzer in the non-instrumented QEMU mode but that had reduced the fuzzing performance and the program we tested froze after a few seconds. We finally solved this by exporting the program with the injected memory error as LLVM bitcode and then calling the modified Clang compiler that shipped with AFL and giving it bitcode of the program containing the memory error directly.

The next problem we had that while the code was now instrumented the program would crash directly on startup due to a stack overflow, no matter what the limit of the stack size was. This was caused by the instrumentation code from AFL that was injected into the program. This code got stuck in a loop before the main function of the program was called. We solved this problem by using LLVM pass mode of AFL to instrument the program.

Chapter 6

Evaluation

For the evaluation, we have chosen to evaluate if there is an association between how often a function is being called, the type of fuzzer and the number of times the fuzzer finds a memory error. We have chosen this research question because [RCA+14] showed that the quality of the seed has an influence on the number of bugs that a fuzzer finds in a program. We were therefore interested if memory errors in locations that are executed more frequently in the initial seed are more likely to be found.

6.1 Setup

We have chosen to test this using a blackbox and greybox fuzzer. For the greybox fuzzer, we used the AFL 2.52b fuzzer. For the blackbox fuzzer, we also used the AFL fuzzer, but with instrumentation disabled, so it behaves like a blackbox fuzzer.

The application that we chose to use for this evaluation is GNU Bash 4.4.18 [GNUa]. We have chosen this program because Bash is a rather large application written in C so that we can inject memory errors in many different places and it is used by many people making it a good real-world example. To fuzz this application, we used most of the regression tests that are supplied by Bash and optimized them as discussed in section 5.

This optimization step of the seed seems essential, as we discovered after running a few instances with memory error injected in them. Without the optimization, none of the instances of bash were crashing within a reasonable time. With the optimized seed, an instance found the first crash within 5 minutes. Upon further inspection this memory error it had found was even present in an unmodified copy of Bash. After a while, the fuzzer found an extra 160 unique crashes in the unmodified Bash. This was one of the reasons why we verify each crash in our framework.

To know how often a function in Bash is being called, we used GNU gprof [GNUb], a tool that profiles an application. We ran each optimized test case on Bash with gprof enabled to get the number of calls per function of a test case and then summed the results of each test cases. After that, we sorted the functions on the number

of function calls and then divided it into three categories: functions that are in the top 50% position, functions that are in the bottom 50%, and functions that are not called in the initial test cases. As seen in Table 6.1, the number of function calls were divided into 931037 to 82, 81 to 1 and 0 calls.

We setup our experiment in the following way: we first injected a single memory error into Bash in a random location for each instance. An instance in our test consists of the blackbox and greybox fuzzer, fuzzing the same modified bash binary. We then supplied the fuzzers with the optimized seed of test cases and only continue if all the test cases do not immediately crash the modified Bash. Because of the large size of Bash, the optimized seed contains many test cases. This causes the greybox fuzzer to spend around 12 hours on our supplied test cases before it is going to fuzz on its mutated test cases. We have therefore chosen to run each instance for 24 hours, so that time between our initial seed test cases and its mutated test cases is equal.

Each instance, thus a blackbox and greybox fuzzer, was run on a 64bit Ubuntu 16.04 system with two physical cores of an Intel Xeon E5-2673 v4 CPU.

When a fuzzing instance was finished, we tested for both the blackbox and greybox fuzzer if each crashing input still causes a crash outside the fuzzer, for example, the fuzzer imposes a 50mb memory limit which could be a reason of a crash. Then we check for each of the unique crashes that the fuzzer has found that the input does not crash the unmodified binary of Bash. This is because if it crashed on an unmodified Bash, it would be an error that is not injected by our framework. Later on in section 6.4, we also tested if the location of the injected error was reached as the previous check would sometimes fail to filter out crashes in Bash itself. When all checks are complete, we count the number of remaining crashes. When that is greater than 0, we say that the fuzzer has found the injected memory error.

Location of function	Maximum calls	Minimum calls	Average calls	Number of functions
In top 50%	931037	82	6441	538
In bottom 50%	81	1	19	538
Not in the initial set				1174

Table 6.1: Functions in the source code of Bash divided into three categories depending on the number of times the function was called in the optimized seed test cases.

6.2 Problems and limitations with fuzzing Bash

During the fuzzing process, we had some problems with Bash as the target for the fuzzer and therefore needed to implement some workarounds. One of the first problems we encountered is that while fuzzing, Bash would sometimes delete the home folder, the directory it runs in and would overwrite its own executable. All of these problems would stop the fuzzing process. We countered this problem by giving each fuzzer its own folders and adjusted folder permissions.

The next problem we had is that while fuzzing, Bash would fork or start itself recursively on seemingly random moments. This caused the system to be overloaded, starting the out-of-memory killer which would then kill

the fuzzing instance. We circumvent this problem by running a script that when the number of modified Bash instances are greater than 6, kills all the modified Bash instances except the ones that AFL uses as a fork server.

The last problem we had, was that after a while, Bash would misuse the `kill` command. Bash would execute the `kill 0` command which kills the process group it runs in and thus also killing the fuzzing job. Another problem with `kill` was that the fuzzer mutated variable names so that background processes could not be killed anymore. This was a big problem for test cases with the sleep command in it as it would overload the system with thousands of processes. To reduce this problem, we changed the `sleep 60 &` to `sleep 0.1 &` in two test cases. This has as limitation that it might influence the code coverage of the test case.

After all these modifications most of the fuzzing instances could run for an extended amount of time.

A limitation of our research is that many fuzzers are based on random mutation or generation. This causes that there might be a chance that the same memory error is found in one run of a fuzzer but not in another run.

Another limitation is that our initial seed that comes from bash regression tests with some test cases removed by us does not cover the complete program. Our seed only covers 1076 of the 2550 functions in bash. This could make it more difficult for the fuzzer to find all the functions. Another problem with seed is that the greybox fuzzer warned us that there might be too many input files in the seed. However, a few tests that we ran with less input files reduced the achieved program coverage after 24 hours of fuzzing in comparison to a run with the complete seed.

A different limitation occurs because we retest each crashing input to filter out crashes that were in Bash itself. It is difficult to replicate the exact environment in which the crash occurred. Therefore, there could be cases where the crash could not be replicated because of for example a crash that depends on a file that was randomly generated by a previous input and later overwritten by another input.

6.3 Initial results

In this section, we describe the results of the first run, why these results are not accurate, how we discovered that and how we tried to improve it.

For our evaluation, we have chosen to run 64 fuzzing instances. In 54 of those instances, both the blackbox and greybox fuzzer had successfully fuzzed for 24 hours. After we collected the crashing inputs of those 54 instances, we ran the inputs several times against a modified and an unmodified copy of bash and then categorized the results depending on the location of the function as described in section 6.1.

When we compared the performance of both the greybox and blackbox fuzzers, Table 6.2 and Table 6.3, we found that the blackbox fuzzer had found almost all memory errors and performed much better than the greybox fuzzer. This result differs from what would usually be expected as a greybox fuzzer employs more advanced techniques than a blackbox fuzzer such as exploring new paths, so we went on investigating what the reason of this could be.

Our first hypothesis was that the greybox might be slower due to bookkeeping of new paths. However, when checking the number of average executions we saw that this was not true, the greybox fuzzer was even faster, 25279122 executions for greybox against 19576674 for the blackbox fuzzer.

The next thing we analyzed were the crashing inputs that blackbox fuzzer had found for each injected memory error. What stood out was the distribution of the initial seed test cases where the crashing inputs originate from. As seen in Figure 6.1, is that mutations from the seed test cases src:000115 and src:000114 (redir4.sub and redir3.sub from the bash regression test) are almost always present in crashing input that a blackbox fuzzing instance has generated.

When we manually tested a small sample of inputs originating from those two seed test cases, some of the inputs would sometimes crash an unmodified copy of Bash. Furthermore, these inputs would not call the function where the memory error was injected when profiled. This is a problem as some fuzzing instances only have crashes that depend on those two seed test cases. This could mean that some of those fuzzing instances did not find the injected memory error. A reason why the blackbox fuzzer is performing better is that the blackbox fuzzer had found far more crashes in Bash itself, and therefore there might be a higher chance that some inputs slipped through the check on the unmodified Bash.

We tried to improve this situation by using gprof to check if the input reaches the function where the injected error is located, as will be described in the next section. Because if that function is not reached, the injected memory error could not have been triggered and the crash could therefore not have been caused due to our injection.

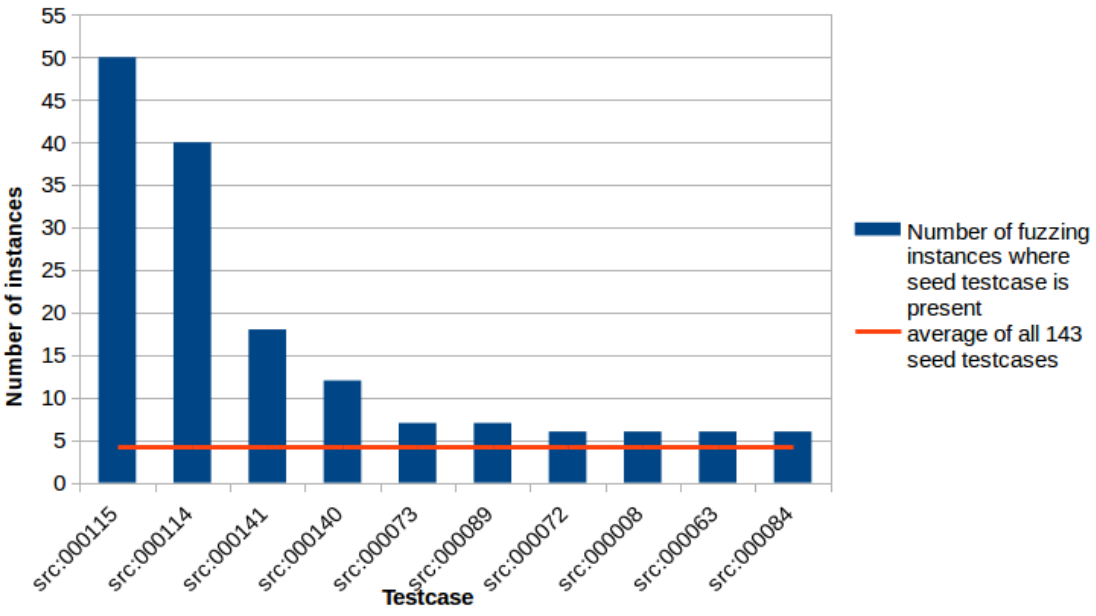

Figure 6.1: Top 10 of seed test cases where the test case is present in the crashes of a blackbox fuzzing instance.

Location of function	Blackbox found	Greybox found
In top 50%	9/9	8/9
In bottom 50%	9/10	5/10
Not in the initial set	34/35	22/35

Table 6.2: Location of the function and number of times the blackbox and greybox fuzzer have found the injected memory error.

	Greybox found	Greybox not found	Total
Blackbox found	35	17	52
Blackbox not found	0	2	2
Total	35	19	54

Table 6.3: Table comparing the performance of the blackbox and greybox fuzzer.

6.4 Results when checked if the injected error is reached

With now only marking an injected memory error as found if the crashing input calls the function where the error was injected, we checked the crashing inputs of the 54 fuzzing instances again. This time as seen in Table 6.4, the blackbox and greybox fuzzer are now finding the same amount injected memory errors, and as seen in Figure 6.2, the outliers of seed test cases are now gone. However, both the black and greybox fuzzer are now only finding 8 injected memory errors out of the 54 instances, which is large decrease in comparison to the results without the extra check.

We also analyzed the average time in hours that the blackbox and greybox fuzzer took to find the memory error. While we observed a large difference as seen in Figure 6.3, a two-tailed t-test gave a p-value of 0.143 and a confidence interval of the difference in mean of (-1.207,7.515). This indicates that the difference in mean was not statistically significant enough to reject the null hypothesis that both groups have the same mean. The difference in mean we observed is most likely caused by memory errors in the bottom 50 % and not in seed groups as seen in 6.7.

Another interesting metric was the unique number of crashes that the fuzzers reported. As seen in Table 6.8, the blackbox fuzzer reported far more unique crashes than the greybox fuzzer. An explanation for this large difference could be that because of the blackbox fuzzer having no instrumentation in the target, it can not reliably determine whether an execution path leading to the crash is unique. This hypothesis is further strengthened by that in almost all fuzzing instances the blackbox fuzzer generates thousands of crashes from mutations that come from the same test case with only very minor differences. This could indicate that only counting the number of crashes is not a good measure when comparing different fuzzers.

Besides comparing the performance of the blackbox and greybox fuzzer, we also wanted to test if there is an association between the location of the injection and the fuzzer finding the memory error. So we did significant tests for Table 6.5 and Table 6.6 both with a significance level of 0.05. Instead of chi-squared, we used the Fisher-Freeman-Halton exact test as in some cells of Table 6.5 and Table 6.6 the expected frequency is not large enough. For the blackbox fuzzer the test resulted in an of p-value 0.034. As the p-value ≤ 0.05 there seems to be enough evidence to reject the null hypothesis that both variables are independent in favor of the alternative

hypothesis that both variables are dependent. This also holds for the greybox fuzzer as with a p-value of 0.026. This seems to suggest that in general for both the blackbox and greybox fuzzer that the location of the function in the seed and the fuzzer finding the memory error are associated.

In addition to the previous test, we also compared the individual groups against each other using a two-tailed Fisher exact test. When we compared the top 50% group against the not in seed group, the blackbox fuzzer had a p-value of 0.024, and the greybox fuzzer had a p-value of 0.042. As the p-value ≤ 0.05, we could for both fuzzers reject the null hypothesis that there no difference between the groups in favor of the alternative hypothesis that there is a difference between the two groups. However, when comparing the bottom 50% against the not in seed group, we could not reject the null hypothesis for both fuzzers as the p-value was 1.000 for the blackbox fuzzer and 0.561 for the greybox fuzzer. When we compared the top 50% against the bottom 50%, the blackbox fuzzer had a p-value of 0.141 and the greybox fuzzer a p-value of 0.033.

	Greybox found	Greybox not found	Total
Blackbox found	7	1	8
Blackbox not found	1	45	46
Total	8	46	54

Table 6.4: Table comparing the performance of the blackbox and greybox fuzzer with the extra profiling check.

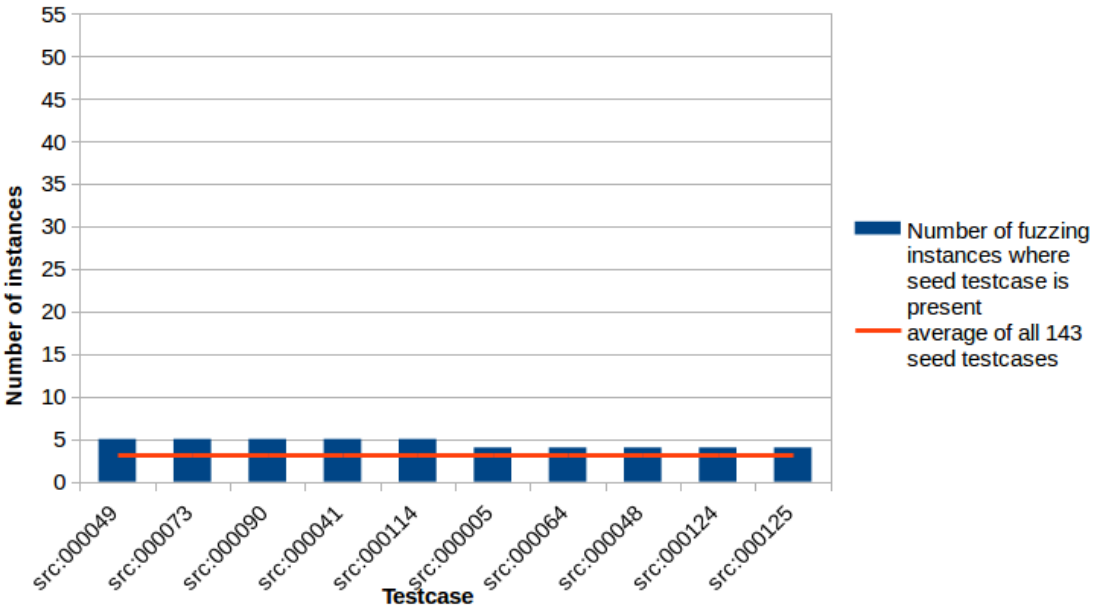

Figure 6.2: Top 10 of seed test cases where the test case is present in the crashes of a blackbox fuzzing instance.

	Injected memory error		
Location of function	Blackbox found	Blackbox not found	Total
In top 50%	4	5	9
In bottom 50%	1	9	10
Not in seed	3	32	35
Total	8	46	54

Table 6.5: A 3x2 contingency table showing the location of the function in the seed in comparison to the number of memory errors that the blackbox fuzzer has found.

Location of function	Injected memory error		Total
	Greybox found	Greybox not found	
In top 50%	4	5	9
In bottom 50%	0	10	10
Not in seed	4	31	35
Total	8	46	54

Table 6.6: A 3x2 contingency table showing the location of the function in the seed in comparison to the number of memory errors that the greybox fuzzer has found.

Location of function	Blackbox average first	Greybox average first
In top 50%	3.99	3.27
In bottom 50%	11.49	
Not in the initial seed	7.76	3.11

Table 6.7: The average time in hours that the blackbox and greybox fuzzer have found the injected memory error.

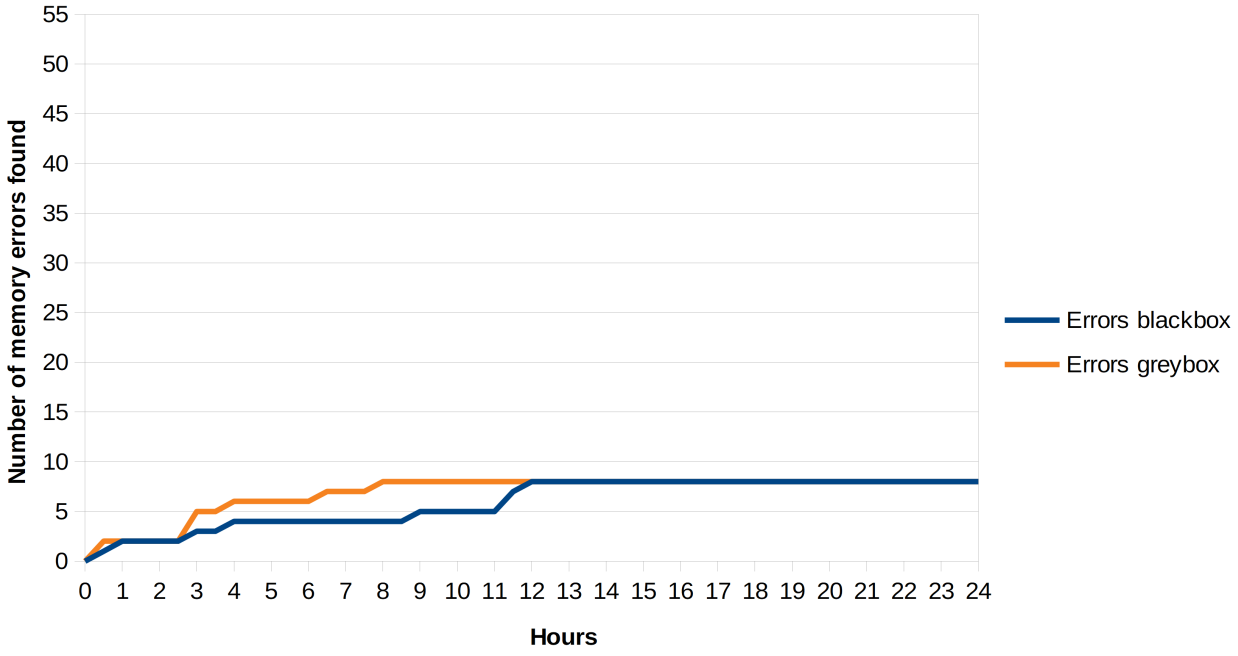

Figure 6.3: The number of memory errors that were found of all blackbox fuzzing instances and of all greybox fuzzing instances over a 24-hour period.

Type of fuzzer	Total unique crashes	Crashes per memory error
Blackbox	5760	720
Greybox	1224	153

Table 6.8: The total and average number of unique crashes per found injected memory error

6.5 Results when fuzzing for a longer period

Because of the small number of injected errors found in the results with the profiling check, we ran a small test where we fuzzed for 48 hours. For this test, we took 16 random samples out of the fuzzing instances where both the black and greybox fuzzer did not find an injected memory error in the 24-hour test.

Both the blackbox and greybox fuzzer did not find any new injected memory errors. For the greybox fuzzer,

the difficulty of finding new memory errors could be explained by comparing it against the 24-hour runs. As seen in Table 6.9, the coverage of the program did only increase slightly. However, it is still just above a third of the program. Furthermore, the number of remaining paths increased, indicating that in the last 24 hour the number of paths found by the fuzzer increased more than the fuzzer could process. Another problem that might hinder the finding of new memory errors is that the stability metric, the metric that measures when given the same input the program reaches the same code each time, is low and even decreases over time. This could make it for the fuzzer more challenging to see the effects of the input it mutated.

Because we did not see any new injected error being found for the first time in the 12 to 48 hour window, we decided not to fuzz with for more than 48 hours.

Type of fuzzer	executions	coverage	paths found	paths pending	stability	cycles
Greybox 24 hours	25279122	34.72%	11447	11123	34,33%	
Greybox 48 hours	49377472	35.95%	12733	12024	29.37%	
Blackbox 24 hours	19576674					99
Blackbox 48 hours	44585530					327

Table 6.9: The average number of: executions of Bash, branch coverage, paths found, paths still pending, percentage of paths that behaved the same, and completed queue cycles

6.6 Some error injections investigated

Because of the small number of memory errors found and the difference in performance depending on the location we decided to investigate some samples of each location to see why some were found and why some are not. For each case where we mention that the injected error was not found, the injection was present in both the 24 and 48-hour fuzzing sessions without being found.

6.6.1 Injections in top 50% functions

The printf injection from Figure 6.4, took place in `echo_builtin` function of bash that handles `echo` command. This injection removed the format specifier so that only the string that `echo` has to output remained in the call to printf. Because of this injection, the caller to `echo` command can directly influence the format string that is sent to printf. The first crash was found when one of the fuzzers typed `echo 0%0S0`, causing a crash by dereferencing a pointer to something that was not allocated. The greybox and blackbox fuzzer found this error quickly. This is likely caused by that the function where the error was injected was allocated a lot of fuzzing time. This is because the `echo` command is present in 115/143 of the initial test cases and was often called multiple times in those test cases.

The next memory error injection we investigated in this category was a modification to the strncpy call that was not by the fuzzers. In this case, Figure 6.5, the framework subtracted 1 off the length of a strncpy call in the `pat_subst` function. This function manages for substitution of patterns in strings. The injection took place

in the part of the function that copies the replacement text into a new string. The problem with this injection as seen in Listing 6.1, is that the at the end of the function either a null terminator is added, or it copies the last part of the input string which also has a null terminator. This problem also occurred for another injection of this category that was in the same function. The only thing the injection in this function can do is leave 8 bits of memory uninitialized in the new string for each match in the input string.

before injection

```
; <label>:108:
%109 = call i32 (i8*, ...) @printf(i8*
    getelementptr inbounds ([3 x i8], [3 x i8]* @
    .str.4061, i64 0, i64 0), i8* nonnull %92)
br label %110
```

after injection

```
; <label>:108:
%109 = call i32 (i8*, ...) @printf(i8* %92) <--
br label %110
```

Figure 6.4: Before and after the injection of a printf error in the echo_builtin function.

Listing 6.1: Small snippet of the original pat_subst code.

```
char * pat_subst (string, pat, rep, mflags) char *string, *pat, *rep; int mflags;{
  ...
  ret[0] = '\0';
  for (replen = STRLEN (rep), rptr = 0, str = string; *str;){
      if (match_pattern (str, pat, mtype, &s, &e) == 0) {break;}
    ...
    rstr = rep;rslen = replen;
    ...
    if (replen){ strncpy (ret + rptr, rstr, rslen); rptr += rslen;}
    str = e;            /* e == end of match */
    ...}
  /* Now copy the unmatched portion of the input string */
  if (str && *str){... strcpy (ret + rptr, str);}
  else ret[rptr] = '\0';
  return ret;
}
```

before injection

```
; <label>:160:
%161 = sext i32 %159 to i64
%162 = getelementptr inbounds i8, i8* %150, i64
    %161
%163 = call i8* @strncpy(i8* %162, i8* %2, i64
    %117) #13
%164 = zext i32 %159 to i64
%165 = add i64 %164, %117
%166 = trunc i64 %165 to i32
br label %167
```

after injection

```
; <label>:160:
%161 = sext i32 %159 to i64
%162 = getelementptr inbounds i8, i8* %150, i64
    %161
%163 = add i64 %117, -1 <--
%164 = call i8* @strncpy(i8* %162, i8* %2, i64
    %163) #13
%165 = zext i32 %159 to i64
%166 = add i64 %165, %117
%167 = trunc i64 %166 to i32
br label %168
```

Figure 6.5: Before and after the injection of a strncpy error in the pat_subst function.

6.6.2 Injections in bottom 50% functions

The memory error injection in Figure 6.6, was the only one to be found by the fuzzer in this category. The injection took place in the `bgp_resize` function which resizes the table that keeps track of the exit codes of background processes. To create this possible memory error, the framework incremented the stop condition of the loop by one. This makes it possible to write outside bounds of the array in the loop as seen in Listing 6.2 and therefore potentially corrupting something. This is because the array is allocated for 4096 elements but in the loop, it is now possible to write to element 4097.

The blackbox fuzzer was the only fuzzer to find this error but took 11 hours to find the first crash. All the 30 crashes had in common that the blackbox fuzzer found it by using process substitution to trigger the `bgp_resize` function and then used the `trap 0` command to run something on the exit of a program which caused a read of the contents of memory address oxFFFFFFFF likely because something got corrupted. A reason for that it took so long to crash could be that there was only 1 test case in the initial seed which called only one time the `bgp_resize` function, so the function had less fuzzing time. Furthermore, the fuzzer had to combine two test cases in the right order to achieve these crashing inputs. A reason of why only the blackbox found the error could be that the greybox fuzzer had not combined any test case yet as it was still finding new paths by mutations of old test cases.

The next injection we investigated, was interesting for us as a quick look already reveals why this case is very unlikely to be found by the fuzzer. The function where this error is located is executed with the bash `caller` command. This function returns information about who called the current function. The injection as seen in Figure 6.7, replaces the format string by the line number variable. The first problem with this injection is that the function only prints information if it is inside a shell script which is a problem as the fuzzer enter all the inputs line by line to the stdin of the program. However, an even bigger problem is that the `BASH_LINENO` environment variable which is read by this function as seen in Listing 6.3, is protected by bash. This means it can not be adjusted by for example an `export` command and thus is limited to the numeric value of the source code line number of the caller.

before injection

```
; <label>:3:
%4 = phi i64 [ %8, %6 ], [ 0, %0 ]
%5 = icmp eq i64 %4, 4096
br i1 %5, label %9, label %6
; <label>:6:
%7 = getelementptr inbounds [4096 x i32], [4096 x
     i32]* @pidstat_table, i64 0, i64 %4
store i32 -1, i32* %7, align 4
%8 = add nuw nsw i64 %4, 1
br label %3
```

after injection

```
; <label>:3:
%4 = phi i64 [ %8, %6 ], [ 0, %0 ]
%5 = icmp eq i64 %4, 4097 <--
br i1 %5, label %9, label %6
; <label>:6:
%7 = getelementptr inbounds [4096 x i32], [4096 x
     i32]* @pidstat_table, i64 0, i64 %4
store i32 -1, i32* %7, align 4
%8 = add nuw nsw i64 %4, 1
br label %3
```

Figure 6.6: Before and after the loop error in the bgp_resize function.

Listing 6.2: Small snippet of the original bgp_resize code.

```
static void bgp_resize ()
  if (bgpids.nalloc == 0){
      /* invalidate hash table when bgpids table is reallocated */
      for (psi = 0; psi < PIDSTAT_TABLE_SZ; psi++) {pidstat_table[psi] = NO_PIDSTAT;}
        ...
    }
...
```

before injection

```
; <label>:111:
%112 = call i32 (i8*, ...) @printf(i8*
    getelementptr inbounds ([10 x i8], [10 x i8]*
     @.str.6.2001, i64 0, i64 0), i8* nonnull
    %95, i8* nonnull %101, i8* nonnull %98)
br label %113
```

after injection

```
; <label>:111:
%112 = call i32 (i8*, ...) @printf(i8* %95) <—
br label %113
```

Figure 6.7: Before and after the injection of a printf error in the caller_builtin function.

Listing 6.3: Small snippet of the original caller_builtin code.

```
int caller_builtin (list) WORD_LIST *list;{
  ...
  GET_ARRAY_FROM_VAR ("BASH_LINENO", bash_lineno_v, bash_lineno_a);
  ...
      lineno_s = array_reference (bash_lineno_a, num);
          ...
      printf("%s %s %s\n", lineno_s, funcname_s, source_s);
  ...
}
```

6.6.3 Injection in functions that were not in the seed

The `strncpy` injection as seen in Figure 6.8, produced some strange results for this category as both type of fuzzers found it in around 10 seconds while the other errors in this category that were found took at least several hours. The memory error was injected into the `tilde_expand` function that is executed when a ~ character is in the input. The generation of a ~ character is often a simple mutation for the fuzzer, for example, it could be done with a bit flip of an existing character. This causes that even tough there were no example of it present in the initial seed, it was quickly found.

The `printf` error injection as seen in Figure 6.9, took place in the `dirs_builtin` function from Bash which displays a list of directories that are stored on a stack by `pushd` command. For a person, this injection would not be too hard to trigger as it only requires placing a format string on the directory stack using the `pushd` command and then calling the `dirs` command with the right arguments as seen in Listing 6.4. For example, entering the line: `pushd -n '%x%n'; dirs +1` will cause the program to crash. However, for a fuzzer, this case is difficult as both the `dirs` and `pushd` commands are not present in the seed. This means that the fuzzer has

to come up with names of both the commands and to make it even worse it has to execute them in the right order with the right arguments. The only way that this might be feasible is if we had supplied a dictionary containing the names of the built-in bash commands to the fuzzer.

before injection

```
; <label>:28:
%29 = phi i32 [ %24, %22 ], [ %16, %14 ]
%30 = phi i8* [ %27, %22 ], [ %17, %14 ]
%31 = sext i32 %15 to i64
%32 = getelementptr inbounds i8, i8* %30, i64 %31
%33 = sext i32 %19 to i64
%34 = call i8* @strncpy(i8* %32, i8* %18, i64
    %33) #13
%35 = getelementptr inbounds i8, i8* %18, i64 %33
%36 = call fastcc i32 @tilde_find_suffix(i8* %35)
%37 = or i32 %19, %36
%38 = icmp eq i32 %37, 0
br i1 %38, label %68, label %39
```

after injection

```
; <label>:28:
%29 = phi i32 [ %24, %22 ], [ %16, %14 ]
%30 = phi i8* [ %27, %22 ], [ %17, %14 ]
%31 = sext i32 %15 to i64
%32 = getelementptr inbounds i8, i8* %30, i64 %31
%33 = sext i32 %19 to i64
%34 = add i64 %33, -1 <--
%35 = call i8* @strncpy(i8* %32, i8* %18, i64
    %34) #13
%36 = getelementptr inbounds i8, i8* %18, i64 %33
%37 = call fastcc i32 @tilde_find_suffix(i8* %36)
%38 = or i32 %19, %37
%39 = icmp eq i32 %38, 0
br i1 %39, label %69, label %40
```

Figure 6.8: Before and after the injection of a strncpy error in the tilde_expand function.

before injection

```
; <label>:188:
%189 = phi i8* [ %187, %186 ], [ %185, %179 ]
%190 = call i32 (i8*, ...) @printf(i8*
    getelementptr inbounds ([3 x i8], [3 x i8]* @
    .str.4061, i64 0, i64 0), i8* %189)
```

after injection

```
; <label>:188:
%189 = phi i8* [ %187, %186 ], [ %185, %179 ]
%190 = call i32 (i8*, ...) @printf(i8* %189) <--
br label %219
```

Figure 6.9: Before and after the injection of a printf error in the dirs_builtin function.

Listing 6.4: Small snippet of the original dirs_builtin code.

```
int dirs_builtin (list) WORD_LIST *list;{
...
#define DIRSTACK_ENTRY(i) (flags & LONGFORM) ? pushd_directory_list[i] : polite_directory_format (
    pushd_directory_list[i])
  /* Now print the requested directory stack entries. */
  if (index_flag){
    if (vflag & 2){..}
    else{printf ("%s", DIRSTACK_ENTRY (desired_index));}
  }
...
}
```

Chapter 7

Related Work

The framework in this paper modifies programs by injecting faults into it. Fault injection [HTI97], a testing technique where a fault is injected into hardware or software in order to test a system, exists for a long time. In hardware fault injection, the hardware can be for example influenced by adjusting current and altering logic signals as tested in [ACL89]. Software-based fault injection can be divided into runtime, where for example the data in CPU registers, memory locations are corrupted [KKA95] and compile-time, where the source code of the application is modified. The use case of research and tools for compile-time fault injection could be divided into two purposes: to test the software or to test the effectiveness of bug finding software.

Research that tests the effectiveness of bug finding tools usually create a collection of bugs inserted into one or more programs that the tool has to find. The bugs can be manually assembled such as in [ZLL04], be based on existing bugs [LLQ+05], be generated programs containing bugs or be injected bugs into a real-world program. In this last category is our thesis and also recent research such as [DGHK+16] and [PH16].

Just as in this thesis, [DGHK+16] injects bugs into random locations of an existing program. However, while this thesis injects a single bug in a completely random location, their approach is to inject multiple bugs and prefer injections in locations that are deeper into the program by using taint analysis. They also verify that for each injection the bug can be triggered by generating an input for each bug, something that this thesis does not do. In [PH16], the authors use a different way to introduce bugs in a program. Instead of injecting a bug in the program, they remove security checks so that the program becomes vulnerable.

When benchmarking fuzzers, there are many parameters of an experiment that can be adjusted. Starting for example with the program that is fuzzed and the initial seed that is given to the fuzzer, to at the end of the experiment measuring the performance by for example counting the number of crashes that a fuzzer has found. In [KRC+18], the benchmarks of 32 fuzzing papers were evaluated. They showed that in many of the papers there were problems in the way the evaluation was carried out.

To improve this situation, they gave guidelines on how future research should evaluate fuzzers. We follow most of their guidelines such as: fuzzing for at least 24-hours as they showed that some fuzzers only show

results after a while and benchmark in terms of known bugs instead of counting unique crashes as it can overcount the number of bugs. However, according to their guidelines, the evaluation in this thesis could be improved by testing multiple times as they observed that performance of fuzzers varies each run, and having multiple seed configurations as they found that having a different seed had a significant impact depending on the type of the fuzzer.

Chapter 8

Conclusions and future work

In this thesis, we investigated if the location of a memory error has an influence on probability of finding it with a fuzzer. We tested this by building a framework that can inject possible memory errors in a random location of a program.

In the evaluation, we injected errors in Bash and found that the location does have an effect on finding the error. Furthermore, when we compared the individual groups, we found that for both fuzzers memory errors injected into functions that were in the top 50% position of function calls had a different chance of being found then functions that were not in the initial seed. When comparing the difference between the performance of the blackbox and greybox fuzzer, we observed that there was no difference in the number of memory error being found and that the difference in mean time for the memory error being found was not statistically significant.

Choosing Bash as application to be fuzzed for this research could have influenced the results, as we had to adjust some test cases and terminate some executions of Bash to prevent fuzzing instances from ending prematurely. Furthermore, because Bash had many memory errors of its own, we had to retest each memory error that the fuzzers found. This could have negatively affected the performance of the fuzzers as it is difficult to replicate the exact environments of the crashes. Some problems with the environment could be improved if a fuzzer reverts the changes from the fuzzed program such as files that were created for each execution.

In manually investigating some samples to check for differences in memory errors that were found or not found and their location, we found some other problems with Bash and our framework. Some of the injected errors had no effect because of how the code surrounding the memory error was programmed or due to how Bash was designed. We also found that some memory errors were not found likely because of the high difficulty of requiring the use of multiple functions that were not present in the initial seed.

Future work could expand this framework by implementing more types of injections, having additional analyses to prevent ineffective injections that do not crash the program and implementing more targets besides Bash that do not modify the environment. It would also be interesting to research if the location of injected error would have an effect on a basic block level instead of functions.

Bibliography

[ABS94] Todd M Austin, Scott E Breach, and Gurindar S Sohi. *Efficient detection of all pointer and array access errors*, volume 29. ACM, 1994.

[ACL89] Jean Arlat, Yves Crouzet, and J-C Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 348–355. IEEE, 1989.

[DGHK+16] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[GLM+08] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[GLM12] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[GNUa] Gnu bash. URL: `https://www.gnu.org/software/bash/`.

[GNUb] Gnu binutils. URL: `https://www.gnu.org/software/binutils/`.

[HTI97] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[KKA95] Ghani A Kanawati, Nasser A Kanawati, and Jacob A Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on computers*, (2):248–260, 1995.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[LC03] Kyung-Suk Lhee and Steve J Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, 2003.

[LLQ+05] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[Oeh05] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[PH16] Jannik Pewny and Thorsten Holz. Evilcoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 214–225. ACM, 2016.

[RCA+14] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, pages 861–875, 2014.

[RJK+17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[Zala] Michal Zalewski. American fuzzy lop. URL: `http://lcamtuf.coredump.cx/afl`.

[Zalb] Michal Zalewski. American fuzzy lop technical whitepaper. URL: `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[ZLL04] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.

Appendix A

The author's contribution to the framework.

The framework that has been built for this thesis was built in collaboration with Vincent van Rijn and Wampie Driessen. While many parts of the framework are built together, the author of this thesis is responsible for the following parts:

- The implementation of the first prototype of the framework which included the alloca and malloc modifications.
- The design and implementation printf and strncpy to strcpy injection.
- The complete rewrite of the base of the framework that features a more modular approach of adding new modification, random memory error selection, support for handling random selection over multiple possible memory errors for a single instruction, support for multiple memory injections into a program and saving information about the injected memory type information to disk.
- Migrated the strncpy to strcpy injection as Unsafe C class into the new framework as a reference implementation for other injections.
- Generation of the minimized seed from the Bash regression tests that the fuzzers use.
- Several workarounds to get AFL fuzzer working with our framework.
- Improved the loop modification by checking for induction variables and if it is used in a pointer calculation.
- Dead code and unused code removal.
- The scripts to prevent Bash from overloading the system.

Furthermore, the author worked on some failed or not used parts in our framework namely: checking the importance of a basic block using dominator trees and checking the importance of a basic block by calculating all paths in a function.