Universiteit Leiden
The Netherlands

Opleiding Informatica

Quantifying fuzzer performance on spatial and temporal memory errors

Vincent van Rijn

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

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
`www.liacs.leidenuniv.nl` 13/08/2018

Abstract

Using fuzzers is an effective way of finding security-related bugs in software. Quantifying exactly how effective fuzzers are at detecting bugs remains a difficult question to answer however, as there is a lack of information about how many bugs were in the program to begin with, a lack of a ground truth. We present a framework that can automatically inject vulnerabilities in real-world applications. The framework establishes a class hierarchy which enables for trivial introduction of new vulnerability injections, by providing both helper functions and built in examples of vulnerability injections. The general design is then further explained and the built in injections are discussed. Using this framework, we establish a ground truth, and we quantify fuzzer performance in detecting the two distinct types of memory errors, namely spatial and temporal memory errors. In an evaluation, we found that the two types memory errors exhibit different detection behaviour. Spatial memory error injections were almost always detected by the fuzzer, while the temporal memory errors were either immediately or never detected.

Contents

1 Introduction **1**

1.1 Thesis Overview 2

2 Background **3**

2.1 Fuzzing and AFL 3

2.2 Vulnerabilities 4

2.2.1 Common vulnerability examples 4

2.3 Basic blocks and control flow graphs 6

3 Related Work **8**

3.1 Injection frameworks 8

3.2 Datasets 9

4 Overview **10**

5 Design **12**

5.1 Built in vulnerabilities 12

5.1.1 Unsafe C function injection 13

5.1.2 Off-by-n injection 13

5.1.3 Loop detection 13

5.1.4 Erroneous memory allocation injection 14

5.1.5 Dangling pointers and use-after-free 14

5.2 Introduction of new vulnerabilities 15

5.3 LLVM 15

5.4 Passes 16

5.4.1 Dump pass 16

5.4.2 Strategic analysis pass 17

5.4.3 Modify pass 18

6 Implementation **19**

6.1 Class hierarchy of the framework 20

6.1.1 Replacing instructions 20

7 Evaluation **21**

7.1 Results 22

7.1.1 Spatial memory errors 22

7.1.2 Analysing spatial memory error injections 23

7.1.3 Temporal memory errors 26

7.1.4 Analysing temporal memory error injections 27

7.2 Evaluation summary 28

8 Conclusions **29**

Bibliography **30**

A Appendix **32**

Chapter 1

Introduction

Using fuzzers is an effective way of finding security-related bugs in software. Fuzzers find bugs that could otherwise be hard to find by providing the program with a large number of different inputs, hoping that it stumbles upon a crash. This in turn could be in indication of the existence of a bug. Determining exactly how effective a fuzzer is at finding bugs remains a difficult question to answer, as there often is a lack of a ground truth: when fuzzing a program it is unknown how many bugs there are in total, and it is therefore impossible to determine crucial statistics such as the percentage of total bugs found. Moreover, it also remains unclear whether fuzzers are more effective at finding certain types of bugs over other types. In particular we will be looking at the two types of memory errors: spatial memory errors (accessing memory outside of what was allocated) and temporal memory errors (accessing at a time which makes the access illegal).

Fuzzer performance in general can be evaluated using synthetic datasets but often do not represent real-world applications. Fuzzing real-world applications however lacks the ground truth as explained above. In this text, we present a framework that can automatically inject the two different types of memory errors in real-world applications. Using this framework, we will benchmark the well-known fuzzer AFL [Zal] on the widely used GNU Bash [Bas], and quantify how well it performs in detecting spatial- and temporal memory errors. With this, we make the following contributions:

- We present a framework that can automatically inject vulnerabilities, that can be used for software testing tool evaluation
- We implement a prototype of the framework in a way so that it can be easily expanded on
- We evaluate the AFL fuzzer in general, and compare spatial- and temporal memory error detection rates

1.1 Thesis Overview

This chapter contains the introduction; Chapter 2 briefly explains background information necessary to understand the text; Chapter 3 discusses related work; Chapter 4 shows an overview of the developed framework. Chapter 5 highlights the general design of the framework and touches upon the implemented vulnerabilities, while Chapter 6 explains the implementation of the framework in more detail. Chapter 7 evaluates the performance of AFL on Bash; Chapter 8 concludes and mentions some potential future work.

This bachelor thesis was supervised by Dr. E. van der Kouwe at the Leiden Institute of Advanced Computer Science (LIACS) from Leiden University.

Chapter 2

Background

2.1 Fuzzing and AFL

Fuzzing is a popular automated technique of finding bugs in software. A fuzzer finds bugs by supplying a large volume of different input data to an executable file, in the hope of finding new and interesting program states, with crashes in particular.

American Fuzzy Lop (commonly referred to as AFL) is the fuzzer we picked as our main benchmarking fuzzer in this research [Zal]. AFL is a grey-box fuzzer, meaning that it uses some form of instrumentation to retrieve data about the program it is fuzzing. AFL in particular injects instrumentation instructions at certain locations to track at run-time which paths have been visited already, and which inputs trigger the discovery of new paths. A limitation of AFL is that this instrumentation is done during compile-time using wrappers for Clang and GCC. The source code of the tested program therefore has to be in possession by the testers. Note that this possession of source code is not strictly necessary, AFL can fuzz without instrumenting the target binary, but this significantly reduces the fuzzing performance as it is in that case blindly supplying input data, program state agnostic.

AFL generates new input in a mutation-based fashion (akin to how evolutionary algorithms evolve their solutions). Input that triggered new program states are more likely to be mutated again, instead of using CPU cycles on inputs that have not revealed any new program states for a while. AFL implements a wide range of possible mutations on the input such as byte/bit flipping, applying simple arithmetic operations to byte/words/dwords and splicing inputs at a random location. A drawback to the particular approach in AFL is that the mutation is chosen arbitrarily. Other fuzzers, such as the VUzzer, base the mutations on control- and data-flow analysis of the program [RJK$^+$17]. This can be leveraged to prioritise deeper paths over frequent paths, which arguably yields more interesting paths. The Angora fuzzer adds context to branch coverage, a technique that enables the fuzzer to distinguish executions of the same branch in different contexts, allowing for better program state exploration [CC18]. When it comes to fuzzers, it is generally desirable that the input it generates is both good, meaning that it has a high chance of discovering new program states, and at the same time can be generated relatively fast. This provides for a tough challenge, and as a result existing fuzzers usually perform a balancing act between the two.

2.2 Vulnerabilities

A software vulnerability is a weakness in a program that could potentially be exploited by an adversary, usually by means of exploiting faults (bugs) in software. There are many kinds of software vulnerabilities, weaknesses and bugs (the Common Weakness Enumeration is a community-developed list of software weaknesses, and at the time of writing lists 716 different categories [CKMM13]).

There is a distinction between two specific types of memory bugs: spatial memory errors and temporal memory errors. [SPWS13]. Spatial memory errors are caused by dereferencing memory addresses out of the bounds set by a certain memory allocation. The buffer overflow/underflow is an archetypical example of a spatial memory error. By being able to write out of bounds, adversaries could potentially overwrite crucial data. For example, by overwriting the return pointer an attacker could manipulate the flow of control of a program, and let the return pointer point to malicious shellcode that has been inserted by the attacker.

Temporal memory errors are essentially caused by (de)referencing memory at the wrong time. This can happen when a memory address has been declared free (by a call to `free` in C or `delete` in C++), ready to be allocated again. Whenever a pointer points to a memory address that has been freed already, we call that pointer *dangling*. Dereferencing a dangling pointer can lead to unpredictable behaviour and can be taken advantage of by attackers.

In the next subsections, we will briefly discuss some typical examples of vulnerabilities and also touch upon how these vulnerabilities can be taken advantage of. The vulnerabilities that are built-in to the framework, most of which are similar to the vulnerabilities discussed here, are discussed in section 5.1

2.2.1 Common vulnerability examples

An infamous and typical example of a function vulnerable to spatial memory errors is the `gets` function: this function takes user input and writes it to a variable but checks no bounds when writing the input data to the specified variable. Another typical example is `strcpy` function, where the unsafe functions differs in just the one argument with its safe counterpart:

Yet the difference is crucial: `strcpy` copies a string until the the null terminator is found, while `strncpy` copies only a set length of characters (note that this is not always used in a safe way either , although it is generally

```c
char * strcpy(char * dst, const char * src);
char * strncpy(char * dst, const char * src, size_t len);
```

Listing 2.1: Difference between safe and unsafe C function prototypes

```
1 int main(int argc, char **argv) {
2         char *p = malloc(512);
3         if (argc >= 2)
4                 strcpy(p, argv[1]);
5         free(p);
6         return 0;
7 }
```

Listing 2.2: Example of a vulnerable malloc call due to a lack of bound checking

considered a safe function [Kli18]). If somehow an attacker could write out of bounds of the input buffer `src` and overwrite the null terminator, crucial data placed after `dst` in memory could be overwritten.

In addition to unsafe C functions, manual memory allocation is a rich source of bugs as well, especially for programmers that are not used to having to deal with memory management. An example of such a mistake: a call to malloc `malloc(10)` for an integer of length 10, or simply forgetting to check bounds:

Which leads to being able to write past the allocated 512 bytes on line 2. This becomes problematic when a second buffer `q` gets allocated right after `p`, as the adversary is now able to overwrite the metadata of *q* and flip the in-use status bit to zero and manipulate the size attribute. Then, when `free()` is called on the buffer `p`, it will also try to free the adjacent buffer `q` and merge it into one big buffer and the pointer pointing to the beginning of `q` can be overwritten [Bro]. This can have serious consequences, as is for example seen in CVE-2004-0200 [jpe], where a heap overflow allowed attackers to execute arbitrary code via a JPEG image, completely compromising confidentiality, integrity and availability of the system. In general, when the user data is larger than the variable its allocated size, it will simply write beyond the bounds of the allocated memory and therefore introduce a memory error.

A third example of a spacial memory error are called *off-by-one*. Off-by-one errors are errors in which, as the name suggests, an argument or boundary check is off by one. Off-by-one errors are especially commonplace in C library function calls because whether or not the length argument includes the null terminator is inconsistent. For example, the *manpage* for `strncpy` reads "If src is less than `len` characters long, the remainder of `dst` is filled with \0 characters. Otherwise, `dst` is not terminated." while the `fgets` *manpage* reads "If any characters are read and there is no error, a \0 character is appended to end the string". This discrepancy naturally leads to off-by-one errors, and can only be overcome by simply knowing (or looking up) how it is implemented for every similar function. The consequence of this is that the least significant byte of the return pointer can sometimes be overwritten.

The most common temporal memory error occur when memory allocated on the heap is released too early, while the reference to the, now deleted, object remains (this is called a dangling pointer): In the example in listing 2.3, the pointer becomes dangling after line 111, yet still gets dereferenced at line 15, in the case the `if` statement evaluates to true. This leads to an immediate vulnerability, as the memory that was declared free could have been claimed and written to again (potentially by an attacker).

```c
typedef struct account {
    char *name;
    double balance;
} account_t;

int main(){
    account_t * account1 = (account_t*) malloc(sizeof(account_t));

    if (!check_balance(account1)){
        error_found = 1;
        free(account1);
    }
    //...
    if (error_found)
        printf("Error found in account %s", account1->name);
    return 0;
}
```

Listing 2.3: Use-after-free vulnerability example

2.3 Basic blocks and control flow graphs

In general, a program consists of data (variables) and instructions. These instructions will (after loading them into memory) be executed whenever their corresponding function has been called. Instructions by themselves are simple, but can exhibit complex behaviour when strung together, in particular when constructs like conditionals and functions calls are introduced.

A basic block then is a list of consecutive instructions which will always be executed in exactly the listed order, even when conditional statements and function calls are taken into account. This consecutive execution is guaranteed by the property that there is no other way of entering the basic block other than at the first instruction (entry), and no way to branch out other than by the final instruction (exit). Basic blocks can be found by using *leaders*, instructions that satisfy any of the following three properties:

- It is the first instruction of a program (by definition)
- It is a target of an (un)conditional jump or goto instruction
- It directly follows an (un)conditional jump or goto instruction

Then, for each leader, its corresponding basic block is found by adding all instructions, beginning from that leader until the next leader is found (or until the program ends).

The basic blocks of a program can be used to construct the control flow graph (CFG) of the source code by connecting the blocks in a way that aligns with the branching instructions of the source code. An example of source code, basic blocks and a corresponding CFG can be seen in figure 2.1. A control flow graph can be used to both optimise and statically analyse source code.

```
for $i = 1$ to 10 do
    for $j = 1$ to 10 do
        $a[i, j] = 0.0$;
for $i = 1$ to 10 do
    $a[i, i] = 1.0$;
```

```
 1)   i = 1
 2)   j = 1
 3)   t1 = 10 * i
 4)   t2 = t1 + j
 5)   t3 = 8 * t2
 6)   t4 = t3 - 88
 7)   a[t4] = 0.0
 8)   j = j + 1
 9)   if j <= 10 goto (3)
10)   i = i + 1
11)   if i <= 10 goto (2)
12)   i = 1
13)   t5 = i - 1
14)   t6 = 88 * t5
15)   a[t6] = 1.0
16)   i = i + 1
17)   if i <= 10 goto (13)
```

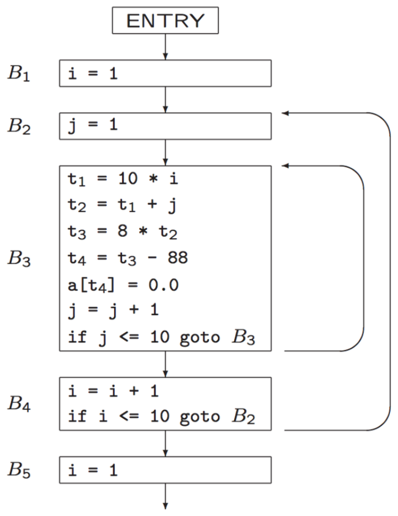


Figure 2.1: From left to right: pseudo code, three-address code intermediate representation (leaders are lines 1, 2, 3, 10, 12, 13), control flow graph. Taken from [Aho03]

Chapter 3

Related Work

Fault injection has been around for at least since 1989 [ACL89]. Back then it was primarily used for checking the fault tolerance (being able to withstand different kinds of failures) of complex computing systems [BCSS90] [HSR95] [HTI97]. Fault injection comes in all shapes and sizes, ranging from electromagnetic injections on the hardware [RNR$^+$15] to simulating instance (server) failures in cloud environments [sim]. The most relevant category are the fault injection frameworks and techniques that inject simple bugs in software, as the framework presented in this text belongs to that category.

3.1 Injection frameworks

One recently developed technique for automatically injecting bugs is LAVA [DGHK$^+$16]. In addition to being able to inject those bugs, the authors of LAVA have focussed primarily on injecting bugs that would be considered realistic. Every bug LAVA injects is accompanied by an input that triggers that bug, in order to ensure that the bug can be hit during execution (meaning that there is a path from the entry to the program state of that bug). It needs an execution trace to do this, which in this case is done by running a program under a dynamic taint analysis for a specific input. The framework presented in this text does not need an execution trace.

An older but relevant framework for injecting bugs is FERRARI [KKA95]. FERRARI inserts software traps (a special type of interrupt to inform the operating system something out of the ordinary has happened) in the target program that can be triggered during execution. This trap alters the program state, resulting in unpredictable and sometimes exploitable behaviour. As opposed to LAVA, there is no guarantee that injected bugs (traps) will be hit during execution. On the other hand, FERRARI is able to remove the traps after they are used, something neither our framework nor LAVA can do.

A third approach taken by existing fault injection frameworks is implemented in Xception [CMS98]. Xception does not modify the target application nor does it insert software traps. Unlike LAVA, it does not require an execution trace. Xception leverages the advanced debugging options on modern CPUs to jump to Xception its own (faulty) code. It does this by interrupting the CPU using five different types of exceptions, making the CPU jump to the injected code and triggering the injection. Injecting faults using Xception therefore comes down to testing the fault tolerance of a system, rather than being a suitable way of benchmarking a fuzzer, as Xception just injects and runs faulty code, rather than injecting faults introduced by insecure programming.

3.2 Datasets

In the frameworks above, injections are performed on existing software projects. This is the most relevant to us, as we are the framework we present in this text does exactly that. There are other ways of benchmarking fuzzer performance too however, mainly by using datasets that contain vulnerable programs. These programs are specifically made to be vulnerable, to be used by software testing tools such as fuzzers. The downside of such datasets is that they are often limited in both volume (number of programs) and size (of the programs), as the programs are usually written by hand.

One such dataset is the Juliet dataset, a collection of more than 81,000 artificial C/C++ and Java programs. All of these programs contain known vulnerabilities, in total covering 181 entries in the Common Weakness Enumeration [CKMM13]. Most of the programs in this dataset contain around 70 lines of code, severely limiting realistic simulation of real-world programs.

Chapter 4

Overview

Benchmarking fuzzers in general is difficult because it is hard to specify exactly how many bugs there are in a program. If a fuzzer stumbles upon a certain number of crashes, it is in the general case impossible to determine the ratio of detected bugs versus the total number of bugs that were in the program. It lacks a certain ground truth.

One way of determining this ratio is by benchmarking on a dataset with a set number of bugs injected in them. Using such a dataset establishes a ground truth, because the exact number of bugs is known beforehand. Because of this ground truth, not just the detection rate, but other metrics such as the false negative rate on that particular dataset can be accurately determined.

Existing modern injection frameworks generally analyse execution traces of programs to determine at which locations in the source code injections should be performed [DGHK+16]. This detailed execution trace makes it possible to effectively determine instructions where input bytes are available that do not determine control flow. Altering variables that do affect flow control leads to unpredictable behaviour when executing programs. This behaviour would be fixed before releasing the program, and is therefore not a realistic bug. Then, using the execution trace, the possible injection targets are found that are near (near in the sense of being close to each other in the execution trace) those instructions and are finally injected. This approach leads to what we would consider realistic bugs, as variables that affect control flow are not modified.

What we present here is a framework that can automatically inject vulnerabilities in a configurable target, with static analysis built in to the framework, establishing a ground truth for accurately benchmarking fuzzers. The fact that static analysis is built in eliminates the aforementioned need of having an execution trace and therefore makes it simpler to use. The obvious drawback is how realistic the injected bugs are, which is, depending on the context, not always a necessity. For example, when performance of fuzzers relative to each other is quantified realistic bugs are not a strict necessity. In the case of wanting to quantify exactly how a fuzzer performs on real-world applications, it would be advisable to use a fault injection framework like LAVA instead. When benchmarking fuzzer performance on finding memory errors, not necessarily in real-world applications but in applications in general, using the presented framework here is appropriate.

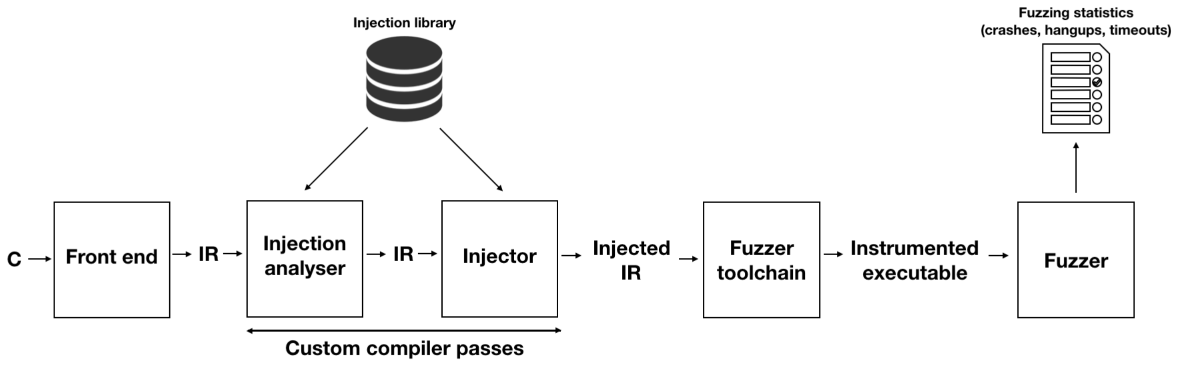


Figure 4.1: Overview of how all components work together

The framework has been built on top of the LLVM framework (see chapter 6 for implementation details), and has been built with modularity and extensibility in mind. A general overview of how the framework works including all utilized components, from initial source code to fuzzing statistics, is shown in figure 4.1. The front end used is Clang. The used fuzzer toolchain in this case is the AFL toolchain, which primarily consists of wrappers of existing tools like assemblers (`afl-as`) and compilers (`afl-clang` and `afl-gcc`). Our framework consists of the injection analyser and injector components, and because of its modularity any other front end or fuzzer can be adapted to use the framework. The two main components of our framework, the injection analyser and injector, are custom LLVM passes, which are thoroughly explained in section 5.4.

This framework has been built by the author of this text in collaboration with Vincent den Hamer and Wampie Driessen. We have worked together on most parts within the framework. I am primarily responsible for the following parts:

- Off-by-n injection (strncpy, fgets, loops)
- Temporal memory error injection (use-after-free)
- Reimplementation of alloca and malloc injection in new framework
- Initial injection implementation and improvement of general framework
- Loop detection and dominator tree construction
- Instruction deletion and replacement

All injections and the general workings of the framework are discussed in this text, with emphasis on the parts I am primarily responsible for.

Chapter 5

Design

In this chapter we will be looking at the general design of the framework. We will briefly discuss the implemented injections, why we have chosen those particular injections and which way they properly emulate real-world vulnerabilities.

5.1 Built in vulnerabilities

The built in injections in this framework primarily focus on memory errors, errors that are prevalent in programming languages where the programmer is responsible for memory management. Because there is no guarantee of memory safety in such languages, as the programmer is solely responsible, attackers are often able to exploit programs written in those languages. The reason why we have chosen to focus on memory errors is because these types of mistakes are easily (and often) made by programmers. Therefore injections of this type resemble mistakes that could realistically be made in real software projects. In table 5.1 a list containing all implemented vulnerability injections can be found.

In the next subsections, we give reasons why we have picked certain injections and also explain how we implemented them.

Category	**Subcategory**	**Injection**
Unsafe C	strncpy	Replace by strcpy
Unsafe C	strtol	Replacement by atol
Unsafe C	strtoll	Replacement by atoll
Memory allocation	alloca	Allocation size manipulation
Memory allocation	malloc	Allocation size manipulation
Off-by-n	strncpy	N argument manipulation
Off-by-n	fgets	N argument manipulation
Off-by-n	strncat	N argument manipulation
Off-by-n	loop	N argument manipulation
Temporal	use-after-free	Insert `free` calls right after initial allocation

Table 5.1: List of all implemented injections

5.1.1 Unsafe C function injection

The built-in category UnsafeC implements several different injections that replace safe C functions by unsafe C functions. Most unsafe C functions are explicitly marked as such (in the form of a compiler warning) but are nonetheless still used in production software today. As an example, whenever a `strncpy` function call is detected, it is replaced by a function call to `strcpy`.

The injections in this category are all similar to that particular example. We argue that this is a mistake easily made by inexperienced programmers, as programs will work exactly as they should when using the unsafe functions. The only difference is that when given unexpected input, the program will exhibit undefined and unstable behaviour.

5.1.2 Off-by-n injection

The second injection of spatial memory errors we inject are *off-by-n* errors, with a configurable *n*. *Off-by-n* errors can occur in both loops as well as in function calls. In loops it is quite natural and not erroneous to have one extra iteration, as long as pointer in the final iteration does not get dereferenced [SPWS13]. When pointers do get dereferenced in the final iteration this can cause direct vulnerabilities, as we have seen that for example overwriting the null terminator at the end of a string can have serious implications in `strcpy`. The details of the loop detection that is needed to inject such vulnerabilities can be found in the following subsection 5.1.3.

5.1.3 Loop detection

Loop detection is extensively used within the framework. First, for every user-defined function call a control flow graph is constructed. Then if the following two statements hold for any pair of basic blocks `h` and `b`:

- b $\rightarrow$ h is a back edge (h has already been visited through a depth-first-search before b)
- h *dominates* b (basic block b will always be executed before b)

Then `h` and `b` induce a natural loop (a loop in a control flow graph). A visual representation of a general natural loop, a back edge and a basic block dominating another can be seen in figure 5.1

LLVM has already implemented this natural loop detection algorithm as a standalone function. In our framework, we check for every `icmp` integer comparison instruction whether it its references has been previously detected as a loop. If that particular `icmp` instruction was detected as part of a natural loop, it is added to the list of potential injection targets (in the `off-by-n` category). The implementation is done this way so that it works exactly the same as all the other built-in vulnerabilities, in order to retain consistency across the framework. The limitation of this approach is that it only works with the `icmp` instruction, excluding loops based on `fcmp` instructions for example. Another limitation is that loops might get optimized away, leading to a situation where a loop was written in the program but not detected as a potential vulnerability injection target.

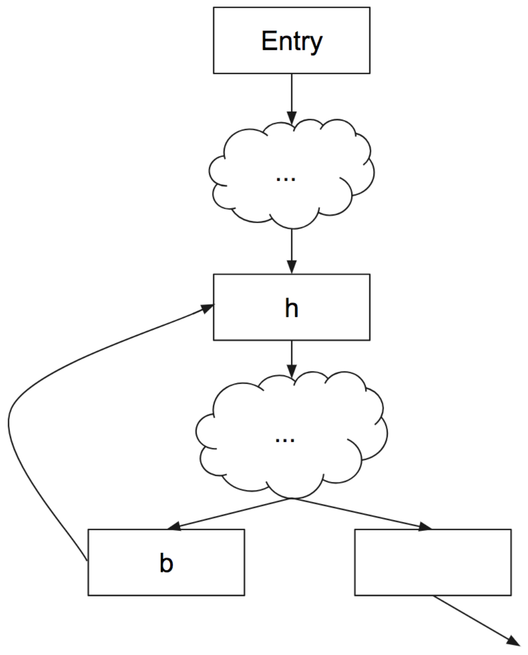


Figure 5.1: Example of a general natural loop, where h *dominates* b and b $\rightarrow$ h is a back edge

5.1.4 Erroneous memory allocation injection

Another spatial memory error category called memory allocation addresses the common error of allocating too little (or too much) data on either the stack or heap. This could happen in cases such as where the programmer is not familiar with how a certain data type might look like in memory (and bound checking might not align with the actual allocated number of bytes). We have chosen to implement this vulnerability because mistakes using memory allocations are made often (by inexperienced programmers in particular).

The root of all allocations in C lie at the functions `malloc` and `alloca`. We have chosen to implement injections on these functions rather than on its derivatives like `calloc` exactly because they are the most generalised way of allocating memory on the stack and heap. The actual injection changes the number of allocated bytes, which can be adjusted to anything the user wants in the source code of the corresponding subcategory. The default built in adjustment subtracts 5 bytes from the original allocation size.

5.1.5 Dangling pointers and use-after-free

We have implemented one temporal memory error, use-after-free. This is the most important vulnerability in this category. The way we have implemented this injection is:

1. Whenever a `malloc` call is found;
2. Find all its usages, and if the usage is a call to `free`, delete it;
3. Insert a new `free` call right after the initial `malloc`

Performing those steps on the code in figure 5.2, it would effectively move the `free` call on line 5 to line 3. This way we can introduce a temporal error while preventing injecting a (temporal) *double-free* error as well. Although this is one other important injection in this category, calling `free` on an already freed memory object caused immediate crashes.

5.2 Introduction of new vulnerabilities

Because of the way this framework has been set-up, it is not difficult to introduce new vulnerabilities that can be injected. In general, when implementing a new vulnerability, three specific things about the vulnerability should be addressed:

- The (sub)category the vulnerability belongs to. This may be an existing or entirely new category, both of which can be implemented with relative ease (in the latter case, most of the implementation can be copied over from existing vulnerability categories).
- When a vulnerability can be injected. In most cases this boils down to when a certain instruction has been found with a particular number of arguments, it can be injected. Another common case is when a call instruction has been found with a particular name (`malloc`, `strncpy`, `fgets`).
- How the vulnerable instruction should be constructed or modified. This can be done by cloning and subsequently altering the original instruction, or alternatively, the vulnerable instruction can be constructed from scratch.

After the desired vulnerabilities are implemented and the target has been properly configured, the target will be built from source using the appropriate building tools. The program then goes through our compiler passes, which are described in detail in the next section. The framework will assess all of the potential vulnerable injection targets and list them by category and their subcategory. The list of potential vulnerable injection targets found by our framework in Bash is shown in subsection 5.4.2. The injection target will be arbitrarily picked (or based on a given seed, passed as environment variable. See Appendix A), and then the actual injection will be performed on the target that was picked.

5.3 LLVM

We have built our framework on top of the LLVM 4.0 framework. LLVM is a compiler infrastructure for code generation and low-level optimisations. [LA04] LLVM is extensively used in academic research, but also in wide use in commercial and open source projects. LLVM has gained huge popularity over the past few years because of two particular reasons that is of special interest to developers and academics:

- LLVM is built in a very modular fashion. LLVM can be used for more than just merely writing optimisation passes. Two prime examples that demonstrate LLVM its versatility, and nothing more than its versatility, is that it can for example be used to make multithreaded processes deterministic [BAD+10] and verifying pointer integrity [KSP+14]

- The LLVM intermediate representation (IR). The LLVM IR is both language and CPU independent, and preserves most of the knowledge of the code. LLVM's IR can be represented in a way that is syntactically similar to assembly languages (just like machine code can be represented as assembly language). An example of LLVM IR and its corresponding C code is shown in figure 5.2.

Example function in C

```c
int main(int argc, char *argv[]){
    char* a = (char *)malloc(15);
    gets(a);

    printf("%s", a);
    free(a);
    return 0;
}
```

Example function in LLVM IR

```llvm
define i32 @main(i32, i8** nocapture
    readnone) local_unnamed_addr #0 {
  %3 = tail call noalias i8* @malloc(i64
      15) #3
  %4 = tail call i8* @gets(i8* %3)
  %5 = tail call i32 (i8*, ...) @printf(i8*
      getelementptr inbounds ([3 x i8], [3 x
      i8]* @.str), i8* %3)
  tail call void @free(i8* %3) #3
  ret i32 0 }
```

Figure 5.2: Original C code example (left) and a corresponding LLVM Intermediate Representation (right)

5.4 Passes

The compilation process using our passes, in short, is as follows. First, all the source code gets tokenised and parsed by Clang (the compiler front end), one file at a time. This results in LLVM IR separate for each source file. We then link this intermediate representation together, resulting in one large file in linked LLVM IR. This goes through an initial optimisation step. This step is called link time optimisation (LTO). This one large optimised file of LLVM IR then gets passed down several LLVM passes, where the actual injection analysis and the injection itself happens. We introduce several new passes in addition to using existing passes. The order in which the passes are run can be seen in figure 5.3. What each pass does is described in detail in the subsections below.

5.4.1 Dump pass

The `dump` pass simply writes out all intermediate code to a file. It is called twice: once before injection and once after. This is useful for debugging purposes. Note that the final dump produces linked bitcode for the injected

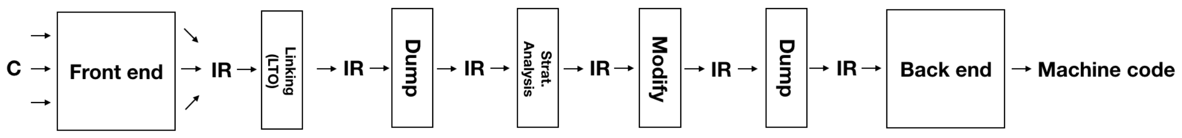


Figure 5.3: Overall passes

target (GNU Bash) that can later on be handed to the AFL toolchain, namely the AFL assembler wrapper `afl-as`, in order to generate a fully instrumented GNU Bash binary. Instrumenting using the other option, something that AFL calls QEMU mode, implicated two major disadvantages:

1. Significant fuzzing performance overhead. In our measurements, performance dropped to approximately 25% of the initial exec speed, which is line with the reported slowdown in the original QEMU paper [Bel05].
2. In this particular case, fuzzing GNU Bash would always hang AFL after some time, thwarting reliable fuzzing.

Which is why we opted for using our dump pass in cooperation with `afl-as` to generate an instrumented and injected Bash executable. Note that these two things are separate, but are called in succession merely to generate the executable, and therefore does not harm the modularity of the framework.

5.4.2 Strategic analysis pass

This pass determines exactly which instructions are injectable and how they should be injected. It goes through every single LLVM IR instruction, checking whether they satisfy the requirements of at least one of the injections in the framework. These requirements are the requirements mentioned in item 2 in section 5.2. Every time an injectable instruction is found, a new instance of the relevant subcategory class is created, and a reference to this instance is returned. The `StrategicAnalysis` pass collects these references in a list and prints them out to show the user which potential injection targets are found (target is GNU Bash, the number following the subcategories denote the number of times an injection in that subcategory has been detected):

```
------ Potential targets ------
offbyn->fgets x1
unsafec->strtol x2
unsafec->strncpy x50
offbyn->loop x398
allocation->alloca x113
allocation->malloc x36
offbyn->strncopy x50
temporal->uaf x36
```

And then is handed to the `Modify` pass. In cases where two potential targets have a reference to the same instruction, as is the case with `allocation` $\rightarrow$ `malloc` and `temporal` $\rightarrow$ `uaf`, a coin is flipped for each such conflict to determine which target is added as an injection target. Because a coin is flipped and only one target is added as an injection target, the problematic situation of trying to inject the same instruction twice cannot occur.

5.4.3 Modify pass

From the list of injectable instructions an arbitrary injection target is picked and injected, by calling the `modify` function on the chosen instance. By default, one random injection is picked, but this can changed as desired in the source code. At this stage the instruction to be modified and original instruction have been set by the previous pass, so as the name suggests, `modify` only modifies the instruction (injecting the vulnerability).

In general, the modify function asserts that no prior modification has been performed on the target instruction and sets up some essential variables for adding new instructions. In most `modify` implementations the target instruction is copied, altered to introduce the chosen vulnerability and then inserted in the IR code. The old instruction is then deleted. In general a distinction is made between whether the argument (of a call to `alloca` for example) is a constant number or a variable. In the constant number case, a simple arithmetic operation is directly applied to change the value. In the case of a variable, an instruction is added to alter the value of the variable, because the argument value is not known compile-time.

An alternative to this implementation is having one bigger `modify` function, that takes a vulnerable instruction and the type of injection as arguments, and then performs the injection. The main advantage of this implementation is that there is no redundant code at all (the current implementations of the `modify` function share some code, but not enough to warrant a helper function). The drawback, why we decided to not take this road, is that it leads to a lack of structuredness, something the framework in its current state excels at.

Chapter 6

Implementation

In figure 6.1, a class diagram of one branch (only the `OffByN` class is shown here, for the sake of simplicity) shows how the general structure of the framework plays out. This diagram is then followed by an explanation of the general framework hierarchy.

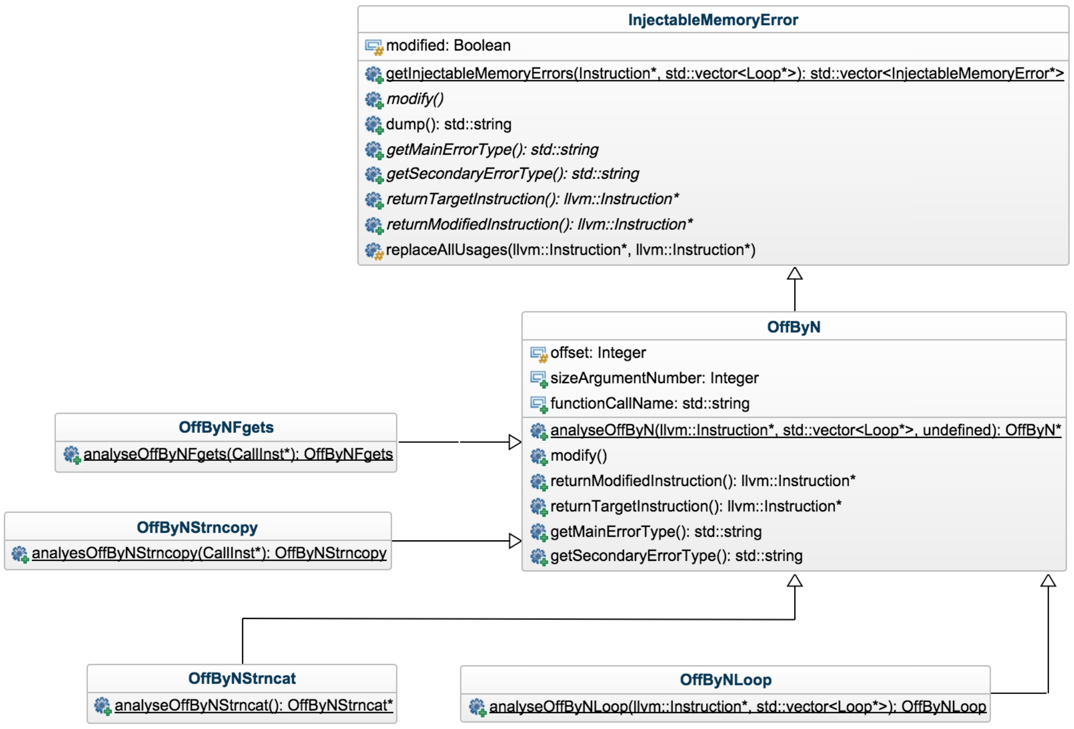

InjectableMemoryError
modified: Boolean
getInjectableMemoryErrors(Instruction*, std::vector<Loop*>): std::vector<InjectableMemoryError*>
modify()
dump(): std::string
getMainErrorType(): std::string
getSecondaryErrorType(): std::string
returnTargetInstruction(): llvm::Instruction*
returnModifiedInstruction(): llvm::Instruction*
replaceAllUsages(llvm::Instruction*, llvm::Instruction*)
OffByN
offset: Integer
sizeArgumentNumber: Integer
functionCallName: std::string
analyseOffByN(llvm::Instruction*, std::vector<Loop*>, undefined): OffByN*
modify()
returnModifiedInstruction(): llvm::Instruction*
returnTargetInstruction(): llvm::Instruction*
getMainErrorType(): std::string
getSecondaryErrorType(): std::string
OffByNFgets
analyseOffByNFgets(CallInst*): OffByNFgets
OffByNStrncopy
analyesOffByNStrncopy(CallInst*): OffByNStrncopy
OffByNStrncat
analyseOffByNStrncat(): OffByNStrncat*
OffByNLoop
analyseOffByNLoop(llvm::Instruction*, std::vector<Loop*>): OffByNLoop

Figure 6.1: Class diagram illustrating the class hierarchy of the off-by-n injection

6.1 Class hierarchy of the framework

At the root of every potential target instruction lies the abstract standalone `InjectableMemoryError` class. This class describes an interface for injections (such as pure virtual functions for getting the error type and applying the injection).

Every vulnerability category is a derivative of the `InjectableMemoryError` class, as is seen with the `OffByN` class in figure 6.1. The primary reason for this is to group all similar injections to retain orderliness and being able to easily enable or disable those categories when injecting targets. Additional fields relevant to all child classes can be introduced in this class as well, as is for example seen in the protected field `offset` in `OffByN`, where `offset` represents the *n* in *off-by-n*. Another responsibility of the category classes is having a general analysis function, which calls the analysis function for each of its children. In the figure above, this function is called `analyseOffByN`.

Every subcategory (synonymous with injection in this context) is a derivative of the category it belongs to, and therefore an indirect derivative of `InjectableMemoryError`. Its main functions are a static boolean function for analysing whether an instruction is injectable or not, whether it has been injected already and how to inject that particular vulnerability. In the diagram above, the modification function `modify` could be abstracted one class up, to the `OffByN` class. In general, the `modify` functions work as described in subsection 5.4.3. Other functions as implemented in the subcategories are for getting the original and modified instructions (`returnTargetInstruction` and `returnModifiedInstruction`) and getting the main and secondary error type (`getMainErrorType` and `getSecondaryErrorType`).

6.1.1 Replacing instructions

The `InjectableMemoryError` class implements a helper function `replaceAllUsages`, as the built-in functions `llvm::replaceAllUsesWith` and `llvm::ReplaceInstWithInst` perform checks for whether the old and new instruction are of the exact same type. If the types are not equal it does not perform the replacement. Those checks do enforce type safety, but this safety is exactly what we are trying to break when injecting vulnerabilities. Therefore, to break that type safety, a custom function was necessary that implements a general way of replacing any instruction with any other possible instruction.

Chapter 7

Evaluation

This chapter presents an evaluation of fuzzer performance of AFL on spatial and memory errors, injected using the developed framework. Fuzzers in general can fuzz programs that take all kinds of input, ranging from images to mouse input. To limit the scope of this evaluation, we will be focusing on injecting vulnerabilities into a single target, GNU Bash. GNU Bash is the most popular Unix shell and has been distributed as the default shell for Linux and macOS. GNU Bash being both extensively used and only accepting textual input makes it a realistic target.

In this evaluation, we perform one injection on Bash at the time, aiming to quantify whether AFL can detect injected vulnerabilities in Bash or not. During our initial testing phase, injecting more than one vulnerability in one run lead to unstable behaviour unsuitable for evaluating detection rates. This injected executable will then be fuzzed using AFL for a maximum of 5 hours each, the time usually given when benchmarking fuzzers [DGHK$^+$16] [RJK$^+$17] [CC18]. This process will be repeated for the categories *off-by-n* and *allocation* representing the spatial memory errors, and *user-after-free* representing the temporal memory error type. Note that this does not include the built in Unsafe C category. For each category, there are 10 runs on different vulnerabilities in that category. This means that for each run, one unique injection has been performed, and is then fuzzed for 5 hours. We do not repeat runs on the same injected vulnerability. AFL does generate new input based on the given testcases (which are always the same testcases and in the same order), but because we perform 10 different runs we argue that this does not harm the evaluation. The results are shown in the next section, providing possible explanations for the results.

We evaluated on a Ubuntu 16.04 LTS system, equipped with a 4-core Intel i5 CPU and 8GB of RAM. The custom testcases handed to AFL (which is used by AFL to generate new input) are based upon the regression tests included in Bash. More information on how and where they were collected can be found in the bachelor thesis of Vincent den Hamer [VdH].

It is worth noting that during the initial tests when fuzzing Bash without injections, AFL already detected a maximum of 12 unique crashes. These crashes were only found when using our own custom testcases (mentioned above). Supplying AFL with the the general testcases included in Bash did not find any crashes. It

Run	Crashes
1.	12
2.	11
3.	11
4.	12

Table 7.1: Fuzzing results on Bash without injections

turned out this is a integer overflow resulting in Bash crashing, and was already reported in 2015 but never fixed [Bas15]. The results of fuzzing Bash without any injections can be found in table 7.1. These are 4 repeated runs on the same uninjected Bash binary, under the same conditions.

7.1 Results

First, the fuzzer detection rates for the spatial and temporal memory errors are shown separately. Then, we analyse two runs (and therefore injections) for each category in order to gain understanding about why certain injections did or did not cause any additional crashes. Concluding, those results are compared and discussed, and we provide possible explanations for the difference. Considering that a maximum of 12 unique crashes were already found in an uninjected version of Bash, we take that number as the baseline. Crash results in the following tables are therefore calculated by `max(0, #crashes - 12)`, with the original number of unique crashes found shown in parentheses. What follows from this is a ground truth, by knowing exactly how many bugs can be found by the fuzzer with and without injecting vulnerabilities. Crashes on initial testcases are denoted by a `*`, followed by the name of the testcase it crashed on.

AFL reports both the number of unique crashes as well as total crashes, and we will be using the unique crashes statistic. AFL considers a crash unique when the execution path involved any transitions that were not previously seen in crashes [AFL]. Each run injects a different fault in the category it belongs to, based on a different seed. The reachability of an injection, whether or not it can be hit during execution, is not guaranteed by the framework in its current state.

7.1.1 Spatial memory errors

The fuzzing results for the off-by-n ($n = 2$) category are shown in table 7.2. We chose $n = 2$ because it represents the often occurring off-by-one error and, in order to make it easier for the fuzzer to detect errors, added 1 to it. We have done something similar for the allocation category, where an inexperienced programmer might allocate one integer less than desired (4 bytes), and then subtracted 1 from it. The results of the off-by-n category are followed by the results the allocation category in table 7.3.

The results in table 7.2 show that AFL is not particularly good at detecting the injected bugs. Only in 3 out of 10 cases AFL could generate input that crashed Bash, with `strncpy` in particular only crashing one out of 5 times. The loop subcategory injection caused crashes only 2 out of 5 times.

Run	Crashes	Subcategory
1.	3 (15)	loop
2.	1 (13)	loop
3.	0 (12)	loop
4.	0 (12)	loop
5.	0 (11)	loop
6.	0 (12)	strncpy
7.	0 (11)	strncpy
8.	0 (9)	strncpy
9.	2 (14)	strncpy
10.	0 (11)	strncpy

Table 7.2: Fuzzing results on the off-by-n category ($n = 2$)

Run	Crashes	Subcategory
1.	0 (9)	alloca
2.	2 (14)	alloca
3.	78 (90)	alloca
4.	0 (12)	alloca
5.	1 (13)	alloca
6.	106 (118)	malloc
7.	* array11.sub	malloc
8.	372 (384)	malloc
9.	12 (24)	malloc
10.	3 (15)	malloc

Table 7.3: Fuzzing results on allocation injection (subtracted 5 from size argument).

Compared to the off-by-n results in table 7.2, the allocation category in table 7.3 shows promising results. There does not seem to be an average case, with 8 out of 10 runs resulting in crashes. In one case, one of the testcases immediately crashed Bash. This could be explained by the fact that both `alloca` and `malloc` are both very common and crucial instructions (calls). If these instructions are then modified, it will naturally lead to many crashes. Comparing this to the strncpy subcategory, which caused crashes in only 1 out of 5 cases, it could be argued that the allocation category crashes more often due to its lower level nature.

7.1.2 Analysing spatial memory error injections

For run 10 an injection in the malloc subcategory has been performed. The injection was performed in the function `xrealloc`, used, like the GNU libc `realloc`, for reallocating memory that has been allocated using `malloc` before. This function is part of Bash its custom implementation of `realloc` and is not the same as the implementation found in GNU libc [Mal]. This version of `realloc` is faster but also allocates more space than it strictly needs, which could hamper fuzzer detection performance (as it would take more to trigger a spatial memory error, because more memory is allocated).

The injected intermediate code can be found in listing 7.1, where we see an additional subtraction of the first argument at line 8, and then a call to `malloc` with that new argument. This injection caused 3 additional crashes. This is a fundamental function for the internal workings of Bash, and the three crashes all have in common that it wants to expand a command that it cannot store in memory, such as `echo \{1..329467295\}` (taken from crash 15), resulting in a segmentation fault. A fuzzer can trigger a crash using this vulnerability

```
define i8* @xrealloc(i8*, i64) local_unnamed_addr #4 {
    ...
; <label>:11:                                   ; preds = %9
 %12 = call i8* @realloc(i8* nonnull %0, i64 %1) #13
 br label %16

; <label>:13:                                   ; preds = %9
 %14 = sub i64 %1, 5
 %15 = call i8* @malloc(i64 %14)
 br label %16
    ....
}
```

Listing 7.1: Snippet of injected IR in the malloc subcategory in the xrealloc function (run 10)

with relative ease because it is injected in the body of a function that is called often, as opposed to a single call that might only be run once. This could be a general explanation of the high detection rates in the allocation category, in addition to the the fact that it might be caused due to the lower level nature of the allocation functions.

For run number 5 in the allocation category, we have performed an injection in the alloca subcategory. We decreased the allocation size by 5, as has been explained in at the beginning of the previous section. The unique vulnerability injection that was picked by the framework ended up in Bash its `adjust_shell_level()` function, a function that is responsible for keeping track how many levels of recursion the current shell is in. This is relevant when `bash` is run within Bash, for example. The intermediate representation prior to injection can be found in listing 7.2, with the relevant allocation at line 2, of which its argument is changed to 1.

From the results in table 7.3, run 5, we can see that it caused one additional crash besides the 12 that were already found in the uninjected versions of Bash. The input generated by AFL that triggered the additional crash does not seem related directly to adjusting the shell level, as one might expect, but manually entering the input in the injected version of Bash did crash Bash.

```
1  define void @adjust_shell_level(i32) local_unnamed_addr #4 {
2   %2 = alloca [6 x i8], align 1
3   %3 = alloca i64, align 8
4   %4 = call i8* @get_string_value(i8* getelementptr inbounds ([6 x i8], [6 x i8]*
        @.str.39.620, i64 0, i64 0))
5   %5 = icmp eq i8* %4, null
6   br i1 %5, label %14, label %6
7
8  ; <label>:6:                                   ; preds = %1
9   %7 = load i8, i8* %4, align 1
10  %8 = icmp eq i8 %7, 0
11  br i1 %8, label %14, label %9
12 }
```

Listing 7.2: Injection in the alloca subcategory in the adjust_shell_level function (run 5)

```
 1 define internal fastcc void @dispcolumn(i32, i8*, i32, i32) unnamed_addr #4 {
 2     ...
 3  %16 = getelementptr inbounds i8, i8* %1, i64 1
 4  %17 = add nsw i32 %2, 2
 5  %18 = sext i32 %17 to i64
 6  %19 = call i8* @strncpy(i8* %16, i8* %9, i64 %18) #13
 7  %20 = getelementptr inbounds i8, i8* %1, i64 %18
 8     ...
 9  br i1 %27, label %28, label %31
10 }
```

Listing 7.3: Snippet of injected IR in the strncpy subcategory in the dispcolumn function (run 10)

Injections in the strncpy subcategory generally did not lead to any additional crashes, as is seen in table 7.2. The IR modified in run 10 of the off-by-n category is shown in listing 7.3. No additional crashes were found in this run.

Injection of `strncpy` would allow for copying a string longer than is actually desired, and therefore open up the possibility of a spatial memory error. The actual injection in this run was in the function `dispcolumn`. The injected instruction can be found at line 4, an addition of 2 to the second function parameter. This specific function is used for displaying columns when `help` is run in Bash. The fuzzer did not manage to crash Bash with this injection, and manually running `help` in the injected Bash did not crash either. Although this injection is reachable, it is not likely that it can trigger any crashes as it does not depend on the input given.

The injection in the loop category in run 5 in the off-by-n category showed no additional crashes, but did show interesting behaviour when manually run. The loop injection ended up in the IR shown in figure 7.1 (left), in the `spname` function. This function is responsible for correcting names that are not spelled exactly as they should. Its corresponding C source code is shown next to it.

Injected IR in loop subcategory (run 5)

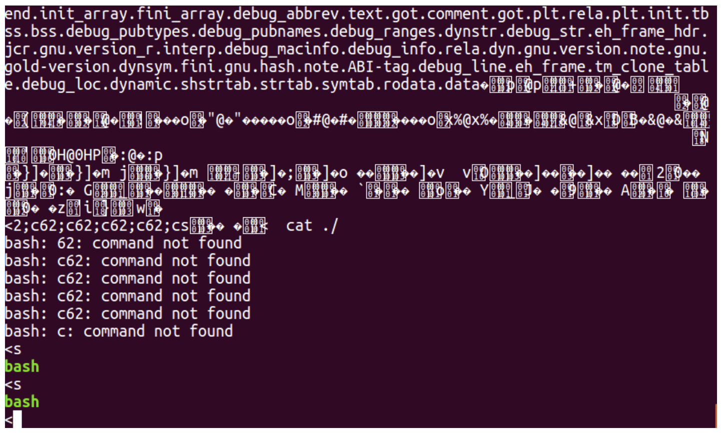
```
 1 define i32 @spname(i8*, i8*) {
 2     ...
 3 ; <label>:19:
 4   store i8 0, i8* %13, align 1
 5   %20 = load i8, i8* %12, align 1
 6   %21 = icmp eq i8 %20, 2
 7   br i1 %21, label %22, label %40
 8     ...
 9 ; <label>:40:
10   switch i8 %41, label %44 [
11     i8 47, label %52
12     i8 0, label %52
13   ]
14     ...
15   ret i32 %64
16 }
```

Corresponding C source code in Bash

```
 1 while (*op == '/')
 2   *np++ = *op++;
 3       *np = '\0';
 4
 5       if (*op == '\0')
 6   {
 7     for (p = guess; *op != '/' && *op !=
          '\0'; op++)
 8   if (p < guess + PATH_MAX)
 9     *p++ = *op;
10       *p = '\0';
11   }
```

Figure 7.1: Injected LLVM IR (left) with corresponding original C source code (right)

The injection added the instruction on line 6, and breaks proper functioning of the first loop in the C code. Although AFL was not able to trigger any additional crashes, using this injected version of Bash in real life is impossible because of the broken `spname` function, resulting in Bash returning nonsensical output and behaviour when entering commands. An example of this can be seen in figure 7.2, showing the output after entering `cat *`, where Bash attempts to run multiple commands (unintended) and improperly displays the input prompt.



```
end.init_array.fini_array.debug_abbrev.text.got.comment.got.plt.rela.plt.init.tb
ss.bss.debug_pubtypes.debug_pubnames.debug_ranges.dynstr.debug_str.eh_frame_hdr.
jcr.gnu.version_r.interp.debug_macinfo.debug_info.rela.dyn.gnu.version.note.gnu.
gold-version.dynsym.fini.gnu.hash.note.ABI-tag.debug_line.eh_frame.tm_clone_tabl
e.debug_loc.dynamic.shstrtab.strtab.symtab.rodata.data�[illegible]
                                                                            [illegible]
�[illegible]���o[illegible]"@�"�����o[illegible]#@�#�[illegible]����o[illegible]%@x%�[illegible]&@[illegible]&x[illegible]�&@�&[illegible]
                                                                              [illegible]
[illegible]H@0HP[illegible]:@�:p
[illegible]}]�[illegible]}]�m j[illegible]}]�m [illegible]]�;[illegible]]�o ��[illegible]�]�v  v[illegible]�]��[illegible]�]�� ��[illegible]2[illegible]��
j[illegible]:� G[illegible]�� �[illegible]C� M[illegible]� `[illegible]� [illegible]� Y[illegible]� �[illegible]� A[illegible]� [illegible]
[illegible]� �z[illegible]i[illegible]w[illegible]
<2;c62;c62;c62;c62;cs[illegible]� �[illegible]  cat ./
bash: 62: command not found
bash: c62: command not found
bash: c62: command not found
bash: c62: command not found
bash: c62: command not found
bash: c: command not found
<s
bash
<s
bash
<
```

Figure 7.2: Output of Bash after injection in the loop subcategory (run 5)

7.1.3 Temporal memory errors

The detection rates of fuzzing temporal memory errors, consisting of the use-after-free category, is shown in table 7.4

Run	Crashes	Subcategory
1.	* array11.sub	use-after-free
2.	* globstar1.sub	use-after-free
3.	0 (12)	use-after-free
4.	* arith5.sub	use-after-free
5.	* globstar1.sub	use-after-free
6.	0 (10)	use-after-free
7.	0 (12)	use-after-free
8.	0 (11)	use-after-free
9.	0 (12)	user-after-free
10.	* array11.sub	use-after-free

Table 7.4: Fuzzing results on use-after-free injection. A * denotes a crash, following by the testcase it crashed on.

The results for temporal memory error detection show that Bash either immediately crashes (1), or it never crashes (2). The following can be said about those 2 cases:

- The bug would almost certainly fixed before releasing the application. Real-world software is subject to testing tools like unit tests, which would trigger crashes such as the ones we observed while running the testcases.

- The bug would go unnoticed. In this case, it is extremely hard for a fuzzer to find any crashes.

This clearly contrasts the behaviour exhibited in the spacial memory error category, where injections on most subcategories resulted in an increase in crashes generated by fuzzers.

7.1.4 Analysing temporal memory error injections

In run 9 of table 7.4 we can see that it did not find any additional crashes. The intermediate representation of the section that was injected is shown in listing 7.4.

```
 1  %18 = call fastcc i64 @xdupmbstowcs2(i32** nonnull %0, i8* nonnull %2)
 2      ...
 3 ; <label>:24:                                    ; preds = %19
 4  %25 = call noalias i8* @malloc(i64 256) #13
 5  %26 = icmp eq i8* %25, null
 6  br i1 %26, label %27, label %28
 7
 8 ; <label>:27:                                    ; preds = %24
 9  call void @free(i8* %20) #13
10  store i32* null, i32** %0, align 8
11  store i8** null, i8*** %1, align 8
12  br label %98
13      ...
14  }
```

Listing 7.4: Snippet of injected IR in the temporal subcategory (run 9)

In the IR, the distance from the initial `malloc` (line 4) to the `free` (line 9) is relatively small. This could be an explanation why no additional crashes could be triggered by AFL, as the chance of allocating that same space of memory again are slim. This is supported by the fact that there are no additional allocations between the initial allocation and its corresponding `free`. After injection, the call to `free` is moved from line 9 to line 5 (per the algorithm explained in subsection 5.1.5).

The final run (run 10) in the temporal subcategory supports the claim that the distance between initial allocation and freeing plays a role in detecting temporal memory errors. The distance between the allocation and free in this run is relatively large, about 20 lines of code, as can be seen in listing 7.5. In addition to the larger distance, another allocation using `malloc` is performed in between as well. As a result, as we can see in the results in table 7.4, it crashed on one of the initial testcases provided to AFL.

```
define i8** @glob_filename(i8*, i32) local_unnamed_addr #4 {
 %3 = call noalias i8* @malloc(i64 8) #13
     ...
 %21 = add i64 %19, 2
 %22 = and i64 %21, 4294967295
 %23 = call noalias i8* @malloc(i64 %22) #13
     ...
; <label>:25:                                    ; preds = %15
 call void @sh_xfree(i8* nonnull %3, i8* getelementptr inbounds ([7 x i8], [7 x i8]i64
     0, i64 0), i32 1065)
}
```

Listing 7.5: Snippet of injected IR in the temporal subcategory (run 10)

This testcase (array11.sub) consists of numerous array declarations, something that would likely allocate memory. Therefore, it is likely to trigger a crash on a temporal memory error injection, given that the distance between allocation and free is large enough. This varying distance can explain the stark contrast observed in the temporal memory error detection rates in table 7.4

7.2 Evaluation summary

Using our framework we can establish a ground truth and easily inject common vulnerabilities into existing software. After evaluating the two different categories on Bash, it can be concluded that there is a difference in detection behaviour between spatial and temporal memory errors. Temporal memory errors were either never or instantaneously detected, while spatial memory errors were detected most of the time.

Chapter 8

Conclusions

In this text, we presented a design as well as an implementation of a framework that can automatically inject vulnerabilities in real-world applications. This framework can be used to establish a ground truth to accurately quantify performance of software testing tools, with fuzzers in particular. The framework has been built with extensibility in mind, and a wide range of vulnerabilities have been discussed and implemented as injections in the framework already.

Using the presented framework we injected two types of memory errors, spatial and temporal memory errors, in GNU Bash. We put the state-of-the-art fuzzer AFL to the test in order to quantify whether there is a difference in fuzzer performance of those two types of memory errors. From the obtained results, we can conclude that the two different types show different behaviour in detection. In the spatial memory error category, AFL was able to trigger additional crashes more often than not, with the allocation category in particular being relatively easy to detect. The temporal category exhibited all or nothing behaviour in our evaluations, by either crashing in the initial testcases, or not at all.

A limitation of this research is that both memory error categories are represented by a subset of the whole category, not including all types of possible common spatial memory errors. Implementing and evaluating all common vulnerabilities would support confidence in the claims made. Performing multiple runs on the same injection or simply performing more runs would improve confidence as well.

We believe we have built a solid foundation for further research. It would be beneficial to implement more injections, and using these injections, more research could be carried out. The results shown in the evaluation raised suspicion about whether detection rates are coupled to the level of abstraction of the instruction, with lower level instructions being easier to detect than higher level instructions. Additional research to either confirm or deny these suspicions can be performed. Lastly, besides continued development of this framework, more research using the framework in its current state is viable too. It would be interesting to investigate the effect of injecting multiple vulnerabilities on fuzzer performance. Investigating how much of an impact the distance between initial allocation and freeing of resources in temporal memory errors really has would be interesting as well.

Bibliography

[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.

[AFL] Afl readme. `http://lcamtuf.coredump.cx/afl/README.txt`. Accessed: 2018-08-13.

[Aho03] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[BAD$^+$10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

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

[Bas15] Integer overflow in braces. `https://lists.gnu.org/archive/html/bug-bash/2015-08/msg00093.html`, 2015. Accessed: 2018-07-14.

[BCSS90] James H. Barton, Edward W. Czeck, Zary Z Segall, and Daniel P. Siewiorek. Fault injection experiments using fiat. *IEEE Transactions on Computers*, 39(4):575–582, 1990.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[Bro] Andries E. Brouwer. Exploiting the heap. `https://www.win.tue.nl/~aeb/linux/hh/hh-11.html`. Accessed: 2018-07-04.

[CC18] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[CKMM13] Steve Christey, J Kenderdine, J Mazella, and B Miles. Common weakness enumeration. `http://cwe.mitre.org/documents/views/view-evolution.html`, 2013.

[CMS98] João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.

[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.

[HSR95] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213. IEEE, 1995.

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

[jpe] Cve-2004-0200: Buffer overflow in the jpeg parsing engine. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-02001`. Accessed: 2018-07-04.

[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.

[Kli18] Evan Klitzke. Beware of strncpy and strncat. `https://eklitzke.org/beware-of-strncpy-and-strncat`, 2018.

[KSP$^+$14] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *OSDI*, volume 14, page 00000, 2014.

[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.

[Mal] Optional features. `https://www.gnu.org/software/bash/manual/html_node/Optional-Features.html`.

[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.

[RNR$^+$15] Lionel Riviere, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High precision fault injections on the instruction cache of armv7-m architectures. *arXiv preprint arXiv:1510.01537*, 2015.

[sim] Simianarmy. `https://github.com/Netflix/SimianArmy`. Accessed: 2018-07-11.

[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.

[VdH] Todo: add proper reference to bachelor thesis vincent den hamer.

[Zal] Michal Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. Accessed: 2018-07-04.

Appendix A

Appendix

Table A.1: List of environment variables that are read

Environment variable	**Effect**
INTEGER_SEED	Sets the seed used in selecting the injection
JOBS	Sets the argument given to `make` -j flag
BUILD_BASH	If set, builds Bash
NO_PACKAGES	If set, does not build packages besides Bash
DO_NOT_MODIFY	If set, does not modify the target