Universiteit Leiden
The Netherlands

Opleiding Informatica

How example set size influences the effectiveness of the AFL fuzzer

W.A.M. Driessen

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

BACHELOR THESIS

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

09/10/2019

Abstract

For the infamous security breach Heartbleed, it has post factum been shown that the vulnerability could have been found by a fuzzer. A fuzzer is a helpful tool for the discovery of bugs in a program. A mutation-based fuzzer generates random input (commonly called fuzz) in an attempt to cause the program to crash. This thesis describes the design process of a framework that can insert a vulnerability into a C or C++ program during compilation. We use this framework to generate executables to target with a fuzzer. By fuzzing these executables we determine if and how the number of example files influences the effectiveness of the AFL fuzzer. The vulnerability injection framework is built upon the LLVM framework and provides two compiler passes, one to analyse the many possible injected errors and another to insert one of these errors. The results show that, in general for AFL, more example files do not necessarily lead to a higher code coverage. The composition of the example set – diversity of features within the example files – has a great influence on the fuzzing capabilities. The results suggest that a limited set of example files can lead to the same coverage as a larger set. This means that a potential balance can be found between the costs related to a higher number of example files and the quality of those example files versus the gain in the fuzzer's effectiveness.

Contents

Chapter 1

Introduction

Over the years, fuzzers have proven their worth in the field of software development. A fuzzer is a helpful tool for the discovery of bugs in a program. A fuzzer generates a lot of fuzz (input) in attempts to find a (set of) inputs that causes the program to crash.

The well-known fuzzer American Fuzzy Lop (AFL) has been used to improve several open-source projects, among others, Mozilla Firefox, OpenSSL, GnuPG and the iOS kernel [Lop]. For the infamous security breach Heartbleed, it has post factum been shown that the vulnerability could have been found by a fuzzer [Whe14]. From this we can take there is a good reason to use fuzzers to test software for (security-relevant) bugs. Background information on fuzzers is provided in Chapter 2.

An important factor in the effectiveness of a fuzzer is the way fuzz is generated. Mutation-based fuzzing is one of the most common ways to generate fuzz. It utilizes a set of example inputs, and alters them incrementally, thus staying close to a valid input-format without complicated (and thereby costly) alterations that aim to be close to a given structure. An unknown factor in mutation-based fuzzing is the influence the number of example inputs has on the process of generating fuzz.

It can be desirable to know the influence of the example set size on the fuzzing process, because fuzzing is an intensive operation which costs both time and resources and any optimizations could be beneficial. In this project we have created a tool that inserts a vulnerability during the compilation of a C or C++ program. This allows us to create "fuzzable" binaries which we know to contain a vulnerability. The tool we have built serves as a device that first and foremost simplifies the process of generating a fuzzable target and furthermore assists in setting up simultaneous fuzzing runs against different example sets. The guaranteed presence of a vulnerability is created by compiling a regular program, and inserting a vulnerability during compile time. The tool leaves it up to the user to analyse the fuzzers' outputs, and make the comparison of their performance. Our tool only supplies convenience to the process.

The vulnerability-insertion framework will be used to generate ample fuzzable binaries in order to find out if the example set size has any influence on the effectiveness of the AFL fuzzer. We measure the fuzzer's

effectiveness by its ability to find the vulnerability and by its line and path coverage.

1.1 Contributions

- A tool that can insert a vulnerability in C or C++ programs.
- Using this tool, we show the influence of example set size on the number of paths found, amount of code covered and ability to find a vulnerability with AFL.

1.2 Thesis Overview

This thesis starts with a discussion of related work, showcasing the necessity of this research, followed by some background definitions, recommended for the reader unfamiliar with the subjects of fuzzers or compilers. The approach, design and implementations will be discussed next, leaving the experiments and the conclusions as the closing chapters of this paper.

This bachelor thesis was written at the Leiden Institute of Advanced Computer Science (LIACS) under supervision of Dr. E. van der Kouwe and Dr. K. Rietveld. The compiler plugin was created in a team setting with V. den Hamer [Ham] and V. van Rijn [Rij]. The author of this thesis is mainly responsible for:

- Embedding the compilation of Bash into the framework, including the instrumentation for AFL
- Implementing the UnsafeC class of vulnerabilities
- Creating a 'sandbox' for Bash – this ended up unused
- Implementing the vulnerability selection algorithm, including seed parameter
- Bugfix: integer size in Off-by-N vulnerability class
- Bugfix: reimplementation of alloca instruction replacement
- Embedding the compilation of libpng into the framework, including the instrumentation for AFL
- Configuring AFL to run in parallel

Chapter 2

Background

This chapter discusses the subjects of compilers, vulnerabilities and fuzzing, in order to refresh the reader's knowledge on these subjects.

2.1 Compiler

A compiler is a tool that converts a high(er) level programming language into machine code. This process is usually divided into a few stages, which we can divide in three groups: the front end and the back end, which are optionally intermediated by a set of compiler passes (usually optimization). The front end usually consists of the lexical, syntax and semantic analyser, after which a code generator translates the analysed input to an intermediate representation (IR). We will not work within the earlier stages of compiling (front end), but rather work with the stages of optimization and assembly. Intermediate representation is a machine-independent format from which the compiler back end can generate machine-dependent code. As the IR is a standardized format, independent of the initial programming language, it is a good place to analyse for and implement potential optimizations. These optimizations are often implemented in a 'compiler pass'. In a compiler pass, the IR is processed in a single pass from beginning to end, rendering a new IR for an optional subsequent pass or the back end. A small sample of intermediate representation is given in Figure 2.1.

Many different compilers are freely available. For the C / C++ programming languages, the most well-known compilers are the GCC compiler [gnu] and the Clang compiler. The Clang compiler is part of the LLVM compiler framework [llva]. Within the Clang compiler it is possible to introduce your own compiler passes. In this research, we have taken advantage of this option in Clang to add a compiler pass containing our vulnerability insertion functionality.

```
%76 = phi i64 [ %93, %91 ], [ 0, %73 ]
%77 = icmp slt i64 %76, %74
br i1 %77, label %78, label %94
```

Figure 2.1: An example of LLVM intermediate representation

```
%76 = phi i64 [ %93, %91 ], [ 0, %73 ]
%77 = add i64 %74, 1
%78 = icmp slt i64 %76, %77
br i1 %78, label %78, label %94
```

Figure 2.2: The example code from Figure 2.1, modified to include an off-by-one error (additional instruction shown in red)

2.2 Vulnerabilities

During the development of a program, flaws in the design or formulation of the code can introduce unintended errors, commonly known as bugs. In an obvious case, a bug will cause a notable malfunction in the program and will be noticed by the developer. In some cases a bug will not always cause a huge transgression, but is only triggered under certain conditions. Vulnerabilities are those kinds of bugs that are present in software that can lead to misuse. Examples of misuse of a program are: access to data that should be restricted, the altering of the program's functionality, the denial of the program's use by others, or the altering of the data that should have been restricted. Some bugs are not to be considered a vulnerability: they may lead to (visual) non-conformities with respect to the program's requirements, but cannot be misused.

In C or C++, many different types of bugs can accidentally be created. Some of the most prevalent will be used in the work described in this thesis. An example of a common error – also applied in this research – is the so called 'off-by-n' error. An example of an off-by-n error is a mismatch between the intended number of iterations and the actual number of iterations of a loop, or when there is a difference between the size of allocated memory and necessary memory. When too little memory is allocated, a program will crash when attempting to write beyond the allocated memory area. When a loop iterates over an array, the program will crash when it attempts to read or write beyond the length of said array. Figure 2.2 shows the introduction of an off-by-one error in the IR shown in Figure 2.1. It increases the loop boundary by one, presumably causing the program to read or write beyond allocated space.

2.3 Fuzzing

A fuzzer is a powerful tool commonly used in the field of application security and software testing. The ultimate goal of a fuzzer is to find bugs in applications. It achieves this by 'fuzzing'. The process of fuzzing is repeatedly running said application and feeding it random, unexpected or invalid input (also called 'fuzz'). The fuzzer changes the fuzz every iteration, thereby hoping to find a class of inputs that causes the program to crash or fail built-in assertions. The fuzz created every new iteration is often not chosen randomly. The first iteration is usually based on one or more 'valid' input samples often called seeds. Iterations thereafter are based on the iteration before it, carefully edited in the right place in order to hit other parts of the target program, increasing the code coverage.

2.3.1 Grey Box Fuzzing

A naive implementation of a fuzzer would just generate random input and feed it to its target program, this is commonly called a black-box fuzzer. A black-box fuzzer does not employ methods to incrementally improve its fuzz. Grey box fuzzing is a common fuzzing strategy as it is relatively easy to set up and is quite an improvement performance-wise. Without analysing the entirety of the source code, a grey-box fuzzer tries to gather information on the impact a test case has on the targeted program. Metrics such as code coverage or execution time can be used to determine promising inputs. Gathering code coverage metrics can be done by for instance running the program within an emulator such as QEMU [QEM], or by inserting instrumentations during the compilation of the program. These instrumentations serve as markers from which, when triggered during the execution of the program, the fuzzer can determine the execution path the input has led to. When a new execution path is found, the input that generated this path is marked as 'interesting' and the fuzzer will be more likely to base new fuzz on that input.

2.3.2 Code Coverage

The percentage of a program's entire source code that has been executed in a collection of runs is commonly called 'Code Coverage'. Code coverage is an important measure for fuzzers. Since a vulnerability can be hidden anywhere, a fuzzer needs to execute as much as possible of the program's code. A fuzzer aims to have an aggregate code coverage of one hundred percent because every uncovered statement might still trigger undesired behavior. Having a code coverage of one hundred percent does not mean the fuzzer is 'finished' with fuzzing. This is because having all statements executed does not guarantee the program is bug-free. Take for example the `malloc` call, which requests the operating system that a certain amount of memory be allocated. If the amount of memory requested exceeds the limit, no memory is allocated and a null pointer is returned. If the returned null pointer is used, this results in a segmentation fault. In this example, if the amount of memory requested is in some way dependent on the input, the statement might be covered, but the vulnerability not found.

2.3.3 Path Coverage

There are many different ways to measure code coverage, each with another strategy. The AFL fuzzer, which is the fuzzer used in the research described in this thesis, gathers information from its instrumentation markers, from which path coverage can be deduced. Path coverage is a form of code coverage wherein the flow of the program is recorded. The flow is the order in which statements within the program are executed. An example would be a 'for' loop that is executed one, two and three times corresponding to three different paths. Information on how the AFL fuzzer gathers its path coverage with help of instrumentation can be found in Chapter 1 of the white paper on the fuzzer [lca].

2.3.4 Line Coverage

Another way of measuring code coverage is 'Line Coverage'. During the run of a program, the number of lines that are executed are kept track of. The line coverage is the percentage of the number of lines of the

original source code that are executed at least once during the run of the program. If all lines of the program are executed, 100% line coverage is reached.

Chapter 3

Related Work

The word 'fuzz' was coined first in 1988 by Prof. B. Miller, in a University of Wisconsin class project [Mil]. Fuzzing originally started without any example, simply by generating a never-ending stream of inputs. After thirty years of maturing, the field of fuzzing has gotten more sophisticated. Over the years several improvements have been introduced, among which fuzzers tracking code coverage [Lop], static analysis tools [SZ18] and machine learning based fuzz generators [GPS17].

A fuzzer, at its core, is a crude tool. As mentioned in the previous paragraph, many different enhancements have been created to optimize the process. A great example of an enhancement of the process is a project named CollAFL [GZQ+18]. Gan et al. investigated the way AFL tracks code coverage during fuzzing runs and enhanced that tracking by solving a hash collision issue and introducing three new seed selection policies. Seed selection policies try to determine which seeds or fuzz are the most promising to use as a base for the generation of further fuzz that with a higher chance of finding new paths. Gan et al. have devised three new seed selection policies for the AFL fuzzer namely: memory-access guided, untouched-branch guided and untouched-descendant guided. The hash collision part is a purely technical enhancement. By ensuring unique identifiers for all basic blocks, a more exact code coverage can be achieved. The seed selection policies on the other hand, are configurations to choose from by the person employing the fuzzer. The seed selection policy may be regarded as a parameter for the fuzzer, a way to influence the process.

Where the CollAFL project focused on an algorithm to select seeds (in combination with already-found paths that appear to be promising), other projects focused on the creation of seeds [GPS17] [GMH+18]. These seeds are example inputs from which subsequent fuzz is generated. The creation of effective fuzz is especially difficult for targets that have a complex input structure such as PNG or PDF files.

Cheng et al. describe a machine-learning tool that optimizes the covered basic blocks and execution paths by learning from the fuzz created by AFL [CZZ+19]. The evaluation of their framework uncovered a 23% increase in the number of basic blocks covered and a 32% increase of execution paths in comparison to the original seeds.

In 2018, R. Gopinath et al. devised a tool for the derivation of an input grammar for a given program [GMH+18]. The generated files were intended to be used as seeds for grey-box fuzzers. Gopinath et al. determined that it was "possible to determine an input language from a given program alone, without requiring input samples" [GMH+18]. They concluded that their precise targeting of feature and feature combinations performed much better than standard mutation based fuzzers.

The projects reported in these papers focused on the generation of high-quality seeds. What all these papers fail to consider is the effect the number of seeds has on the performance of a fuzzer. Therefore the research described in this thesis focuses on the effect the number of seeds has on the path and line coverage of the fuzzer. If this effect is known, a trade-off can be made between the efficiency and the accuracy of a fuzzer.

Just as in the papers by Gopinath et al. and Cheng et al., this research considers seeds as a parameters of the fuzzing project. We used code coverage as a measure for the effectiveness of the fuzzer as was done with the CollAFL project [GZQ+18]. However, in contrast to the research described above, the focus of the research in this thesis initially lies on the quantity of the input files, rather than the quality.

Chapter 4

Overview

In order to investigate the efficiency of a fuzzer, we wanted to generate target programs which are known to have a vulnerability. Creating such a target would previously require taking an existing program's source code, altering it by purposefully introducing a vulnerability, and compiling it. This process is an intensive task and if it were to be performed repeatedly, it would take a lot of time and effort to do this manually. Automating this process would enable a user to create many different 'fuzzable' executables in a short period of time with considerably less effort. We have chosen to implement this process within the existing LLVM compiler framework [llva]. This has led the authors to the following requirements: the vulnerability insertion framework must be able to take a program's source code, written in C or C++, then analyse for all possible places to make an alteration. When all possible modifications are found, one of them must be chosen and performed.

The vulnerability insertion framework is divided in roughly three sections, as depicted in the dashed box within Figure 4.1. Part one is the compiler front-end, this converts the source code to an intermediate representation (IR) format, as described in Section 2.1. This part of the framework is unaltered from the LLVM compiler framework, it is used as such in the research described in this thesis.

In part two, the compiler passes take the IR and work within this format, analysing and altering it. The four compiler passes we utilize are, in order, a 'dump' pass, an analysis pass, the modification pass and finally another dump pass. The purpose of the analysis pass is to gather information on the structure of the program. This information will later be used in the modification pass. The analysis pass leaves the IR

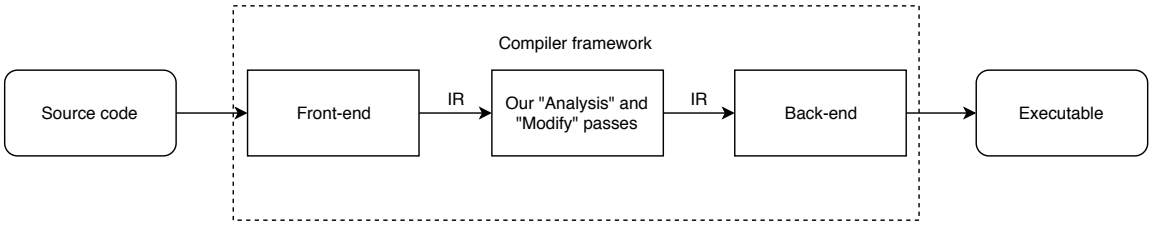

Figure 4.1: An overview of the vulnerability insertion framework.

unaltered. The modification pass runs through the source code, marking all possible locations for vulnerability insertions. Some vulnerabilities, such as 'use-after-free' and 'off-by-one' for loops are identified with help of the information gathered in the analysis pass. Passes one and four, the dump passes, were originally introduced for debugging purposes. In the final version, the last dump pass enables us to take the dumped IR and insert AFL instrumentation. More details concerning the specifics of the AFL instrumentation are described in Section 6.4.

The third part of the framework is the conversion from the IR format to an executable format. In a regular compiler this would just consist of the back-end but in the research described in this thesis, this is where we included the instrumentation necessary for the AFL fuzzer. As mentioned in the previous paragraph, the last compiler pass dumps all IR. In order to create an instrumented executable from this IR dump, the LLVM assembler turns it into bitcode and the AFL clang-wrapper adds instrumentation to this bitcode file and generates an executable.

4.1 Generating the Executable

The compiler front-end is responsible for converting the source code to intermediate representation. In the framework we built upon [VDKNG17] Clang 4 was used, we continued with this version. When invoked with the `-flto` flag, Clang employs link-time optimization. This ensures all IR will be accumulated before calling upon the compiler passes. This is intended, otherwise the compiler passes would run more than once. The modification pass introduced in this thesis marks and inserts the following types of vulnerabilities:

1. OffByN: loops, strncopy, fgets,strncat
2. Allocation: alloca, malloc
3. Temporal: use after free
4. UnsafeC: strncpy, strtol
5. FormatStrings: printf

The modification pass takes the list of possible modifications, selects one of them and then introduces the vulnerability in the selected location, taking care to update references from the original result to the new result wherever applicable. The vulnerability selected for modification is picked based on a random number generator. Without input, this results in a random vulnerability. A parameter is introduced called 'INTEGER_SEED' with which the user can replicate a given vulnerability.

When all compiler passes are finished, the LLVM assembler is fed with the IR from which it creates a bitcode file including the modifications by the passes. When working with AFL, instrumentation is added by supplying the bitcode file to the `afl-clang` wrapper. This renders the instrumented executable.

4.2 The Fuzzer

When the vulnerability insertion framework has done its job, the fuzzer takes the executable, takes an assortment of predefined inputs for the target and starts fuzzing. When the process is stopped by the user, the fuzzer has accumulated a file or folder containing the details of the run. Common details entail the number of crashes, the number of unique crashes and the number of paths discovered. We aimed to compare three different sizes of example sets on the number of paths found. Thus, employing the framework, we will generate three identical executables and fuzz them against different sizes of example sets. After we have set up the framework within three identical (virtual) machines, we let each of them generate the same executable by using the 'INTEGER_SEED' parameter. With the generated executables all three machines start fuzzing. The only difference being that every machine has a different size of example set.

Chapter 5

Design

While designing our framework various parameters were to be taken into consideration. We wanted to supply the AFL fuzzer with an executable of which we knew it contained an exploitable bug. This way we aimed to mimic a 'real life' scenario of a fuzzing project.

5.1 Compiler Passes

The first and foremost step in the project was to create a compiler that was able to analyse the program and insert a vulnerability. We chose to adopt the LLVM framework. The LLVM project is an open-source collection of compiler-related technologies. The framework also contains an interface that allows us to write our own compiler passes. During these passes we can add, alter, and/or remove code. The framework helps to determine what to do by providing various analytic functionalities. Examples of these helpful functionalities are the construction of Dominator Trees and the provision of Loop information.

We created an LLVM compiler pass we called 'Strategic Analysis', which iterates over all instructions found in the IR. When an instruction is an instance of a function call, the pass marks the instruction as 'modifiable' when it is one of the instructions for which we have implemented a vulnerability (see Section 5.2). Next to this, it also analyses the dominator tree for all functions in the module. When a loop is found in the dominator tree, it is added to the list of loops, which will later be useful in the analysis of possible off-by-n targets.

5.2 Considering Possible Vulnerabilities

The next step concerned research on which type of vulnerabilities we could insert. We wanted to generate vulnerabilities that were easy to construct from a 'real-life' program. We chose to do this because it would mimic the kind of bugs a human programmer could insert by accident. Based on the overview by Szekeres, Laszlo et al. [SPWS14] and on the possibilities within the LLVM framework, we decided to start with memory allocation errors and off-by-one errors. Memory allocation errors can occur when during memory allocation the allocated size is too small for its intended purpose. The error never occurs immediately, but only when

attempts are undertaken to use memory outside the allocated boundaries. To introduce such an error, our compiler pass purposely reduces the intended allocation size when it encounters an allocation instruction. Off-by-one errors happen whenever some loop iterates one (or more) times too few or too many. These errors can be easily introduced by increasing or decreasing the boundary check on some operations or loops. Examples of eligible operations are the `strncpy` function, or the iteration over array elements. In case the number of characters copied by `strncpy` were to be increased to a number larger than the length of the string, an attempted read at unallocated memory may trigger an error.

5.3 Fuzzing

After we implemented some vulnerability-inserting functionality into the compiler, we needed to create a script to feed the finished binary to the fuzzer. Since probably a large number of runs would be required, we automated the process of setting up the environment and preparing the fuzzing. We chose AFL as the first fuzzer to implement. This is mostly because AFL supplies different fuzzing options, including black-box, emulated and instrumented fuzzing. We decided to use instrumented fuzzing based on the relative simplicity to set it up. We could not instrument the already created executable. In order to add instrumentation we had to take a few steps back in the compilation process. Right after our modification compiler pass, we triggered a dump pass that writes all the IR into a file. When the compiler pipeline has completed successfully, we assembled the generated IR file into a bitcode file, which can be instrumented and turned into an executable by the `afl-clang` tool.

5.3.1 Fuzzing Target

The first target we implemented was GNU Bash. We considered Bash to be a good candidate because it is easy to compile and it basically revolves around input. Fuzzers often need an example input, based on which they will create subsequent fuzz. The GNU Bash project comes with a large number of tests. These tests are scripts, designed to test the correct operation of the shell. We intended to use (a subset of) these test cases as a basis for the example files we supply to a fuzzing run. During the fuzzing of Bash, we encountered problems, see Section 6.5.

The problems encountered with Bash made us switch to targeting the libpng library [lib]. The libpng library is the official PNG reference library. The libpng library enables users to read from and write to the PNG file format, and apply modifications to the image within. A great argument for using this library as a fuzzing target is the fact that the main element of the library is a parser, the eminent place for a program to digest its input. Libpng, just like GNU Bash, basically revolves around input.

Chapter 6

Implementation

The vulnerability injection framework described in this thesis was built upon an existing project which employed the LLVM framework. This existing project already provided a script for the download and compilation of all tools necessary to compile LLVM 4.0 and added an easily extendable way to add compiler passes. Another helpful aspect of the existing framework is that the list of 'targets', i.e. the programs that needed to be compiled, is easily extended with a new target. We received access to this base framework [HJP+16] [VDKNG17] from our supervisor Dr. E. van der Kouwe.

6.1 Compiler Pass

The compiler pass is built as a C++ class deriving from the `ModulePass` class, this is one of the ways the LLVM framework is open for extension. The LLVM documentation was a great help in the creation of the compiler passes [llvc]. The declaration of our 'Modify' pass can be found in Figure 6.1. By overriding the `getAnalysisUsage` method, we can add the constraint that our analysis pass, implemented in the `StrategicAnalysis` class, should have been called before this pass is run. The overridden method `runOnModule` is called with a `Module` parameter, this parameter contains the Intermediate Representation for the program currently being compiled in an object structure that we can work with.

```
struct Modify : public ModulePass {
    static char ID;
    Modify() : ModulePass(ID) {}
    virtual bool runOnModule(Module &M);
    virtual void getAnalysisUsage(AnalysisUsage &AU) const;
};
void Modify::getAnalysisUsage(AnalysisUsage &AU) const {
        AU.addRequired<StrategicAnalysis>();
}
```

Figure 6.1: An example of LLVM Intermediate Representation

```
std::vector<InjectableMemoryError*> InjectableMemoryErrors;

InjectableMemoryError* offByN = OffByN::analyseOffByN(targetInstruction, loops);
if(offByN != nullptr)
    InjectableMemoryErrors.emplace_back(offByN);
```

Figure 6.2: The analysis pass checks if an instruction could be manipulated to an off-by-n vulnerability

```
class InjectableMemoryError {
    public:
    //Injects the memory error
    virtual void modify() = 0;
    //Return a string how an instruction was modified
    //Return the main error type eg. offbyone, unsafec etc.
    //Used for logging to a file
    virtual std::string getMainErrorType() = 0;
    //Return the specific type of memory error eg. strncpy, printf etc.
    virtual std::string getSecondaryErrorType()= 0;
    //Returns the original instruction so that an user can check if
    // two or more memory errors point to the same instruction
    virtual llvm::Instruction* returnTargetInstruction() = 0;

    ...
```

Figure 6.3: The `InjectableMemoryError` class serves as an abstraction for modifiable instructions

6.1.1 Analysis Pass

The analysis pass' responsibility is the gathering of possible vulnerability insertions. First it gathers information on loops within the module and it creates a Dominator Tree (credits: V. van Rijn) [Rij]. Then, when the loop information and Dominator Trees are gathered, it starts looping over all instructions. For every implemented vulnerability class it checks if the instruction could be mutated to trigger that vulnerability and if so, it adds the instruction to the list of possible vulnerabilities. An example of this step can be seen for the `OffByN` class in Figure 6.2. When all instructions have been analysed, the pass finishes.

6.1.2 Modification Pass

All vulnerability-specific classes we have included in our framework derive from the abstract class `InjectableMemoryError` we have created, see Figure 6.3. All classes that derive from the `InjectableMemoryError` class, like `OffByN` and `UnsafeC`, implement the `modify` method. This method is specific to the different vulnerabilities. In Figure 6.4 it is shown how the `UnsafeC` class takes the original instruction, and replaces it with an older, deprecated version of that instruction. These older functions, from the `ato*` class (e.g. `atoi`, `atol`), may trigger undefined behavior in case of overflow. These can therefore be considered a vulnerability.

```
//Get the types of the dest and src.
ArrayRef<Type *> params = {originalFunctionCall->getOperand(0)->getType()};
//Create a functiontype using the types of dest, src en the returntype
FunctionType* funcType = FunctionType::get(returnType, params, false);
//find the function that replaces the target
targetName = replacementPair.first;
//Create the new function
Constant* castCall = M->getOrInsertFunction(
    StringRef(replacementPair.second),
    funcType,
    strCastFunction->getAttributes());

if(Function* newFunction = dyn_cast<Function>(castCall)){
    Value* str = originalFunctionCall->getOperand(0);

    ArrayRef<Value *> castOperands = {str};
    IRBuilder<> instrbuilder(originalFunctionCall);

    CallInst* castCall = instrbuilder.CreateCall(newFunction, castOperands);

    replaceAllUsages(originalFunctionCall,castCall);

    modifiedFunctionCall = castCall;

}
```

Figure 6.4: The unsafeC class takes a function call, and replaces it with another, transferring the arguments

6.1.3 Bug fixing Vulnerability Classes

As for any software development project, the vulnerability injection framework needed some inevitable bug fixing. As attributed in the contributions section, the author of this thesis was responsible for bug fixing the `alloca` injection and for fixing an assumption that all integer types were specific to a function. The original `alloca` modification implementation gave no problems while compiling Bash, but when compiling the new target libpng, the compilation crashed. During `alloca` modification, the original instruction needs to be replaced by the new version, using half the original allocation size. In order to replace such a function call, the framework checks all 'users' of the original instruction, and replaces their usage of the old instruction with the new instruction. The problem was that, when accessing a property within one of the the allocated objects through an array index, the LLVM framework uses a concept they call `GetElementPtrInst` or 'Get Element Pointer Instruction' in order to find the location of the property accessed [llvb]. Figure 6.5 shows how the `AllocationAlloca` class manages to replace these types of usages. Another problem that arose when compiling libpng was that some of our vulnerability classes used predefined integer sizes in their modifications. Libpng made use of different integer sizes, and so we were forced to revisit all modifications where integer sizes, mostly fixed on 32 bits, were assumed.

```
void AllocationAlloca::replaceAllUsagesForAllocA(
    Instruction* origInst, AllocaInst* newAlloca)
{
    std::vector<Use*> vec;
    for(auto& use : origInst->uses()){
        vec.emplace_back(&use);
    }

    for(Use* use : vec){
        User* user = use->getUser();
        user->dump();
        if(auto gep_inst = dyn_cast<GetElementPtrInst>(user))
        {
            gep_inst->setSourceElementType(newAlloca->getAllocatedType());
            user->setOperand(use->getOperandNo(), newAlloca);
        } else {
            user->replaceUsesOfWith(origInst, newAlloca);
        }
    }

    origInst->eraseFromParent();
}
```

Figure 6.5: The AllocationAlloca class replaces its original usages with the updated instruction.

6.2 Vulnerability Selection

At the start of the project, our goal was to create an option in the framework that would allow the user to direct the insertion of a given vulnerability at a specific location. Due to time constraints we were forced to lower the bar on this functionality. The modification pass now picks a vulnerability at random from the list of possible mutations that were gathered in the analysis pass we created. The process can be steered by the use of the 'INTEGER_SEED' environment variable, which simply seeds the random number generator. This allows for the reproducibility of vulnerability insertions.

6.3 Compiling Bash

Compiling Bash is usually as easy as running `./configure` followed by `make`. In our case, we needed to alter this process due to the necessity of adding AFL instrumentation. Before the compilation, `autoconf` performs checks on the correct behavior of the compiler. If we were to include our modification passes during this check, the assertions of the configure script would fail, and we would be unable to compile further. We solved this problem by creating a check on the presence of an environment variable, which we declared during the configuration stage. Another problem was the fact that we want our passes to run only during the final linking. Generally, programs are compiled in separate chunks called object files (simplified). These object files are then gathered in a linking stage, creating the final executable. During the creation of these object files we do not want to insert our vulnerability, since then overall, we would have run our passes more than once. To overcome this problem, we created a 'wrapper' around the compiler, a small script that calls the real compiler. If certain arguments are given to this wrapper, it adds the command-line arguments that trigger our passes.

```
export NEW_AFL_PATH=$(mktemp -d)
cp -r $AFL_PATH/* $NEW_AFL_PATH/.
cp -r $PATHROOT/testcases $NEW_AFL_PATH
mkdir -p $NEW_AFL_PATH/output-bash
cp bash $NEW_AFL_PATH/bash
sudo chown -R testuser:testuser $NEW_AFL_PATH
sudo -u testuser bash <<EOF
finish() {
    echo "Fuzz details in: $NEW_AFL_PATH/output-bash"
}
trap finish exit
cd
export AFL_PATH=$NEW_AFL_PATH
$NEW_AFL_PATH/afl-fuzz -i $NEW_AFL_PATH/testcases/bash -o \
    $NEW_AFL_PATH/output-bash\ $NEW_AFL_PATH/bash"
EOF
```

Figure 6.6: A shell script for the creation of a sandboxed environment to fuzz Bash in.

6.4 Instrumenting for AFL

AFL's instrumented mode requires an executable that has been instrumented during compile time. This proved to be a challenge, because the `afl-clang` compiler, although wrapping correctly around the compiler, calls its own assembler. This created the problem that the modification was not performed. We solved this by calling the `afl-clang` wrapper manually after the modified bitcode has been created.

6.5 Fuzzing Bash

Early tests with AFL and Bash have shown a slight complication while fuzzing Bash. Bash commands can be followed by the > sign, which should then be followed by a filename. This diverts the standard output of the command to the file given. This phenomenon led to the entire working directory of the fuzzer and/or home directory of the user being filled up with nonsense files. This led us to the solution to run the fuzzer as a dedicated user with close-to-none privileges in a temporary folder. Figure 6.6 shows the script that sets up this 'sandbox'. It starts by creating a temporary folder, after which it copies all necessary files to this folder. When all required files are present in this folder, all permissions of this folder are assigned to a new user called 'testuser'. A new shell is opened for this user in which the AFL fuzzing run is initiated.

6.6 Libpng

As libpng is a library, we needed to write a small 'entry program' or wrapper that we can supply input to. Since the source code of libpng also includes a small program called 'pngtest', we decided to use this tool as our fuzzing target. Pngtest uses libpng to read a PNG file and confirms or denies its correctness. The program reads a file called 'pngtest.png' from the current working directory. This did not work well together with our plan to fuzz in parallel, as is explained further in the next section. Luckily, the program also accepts a command-line argument for an alternative file path to read from.

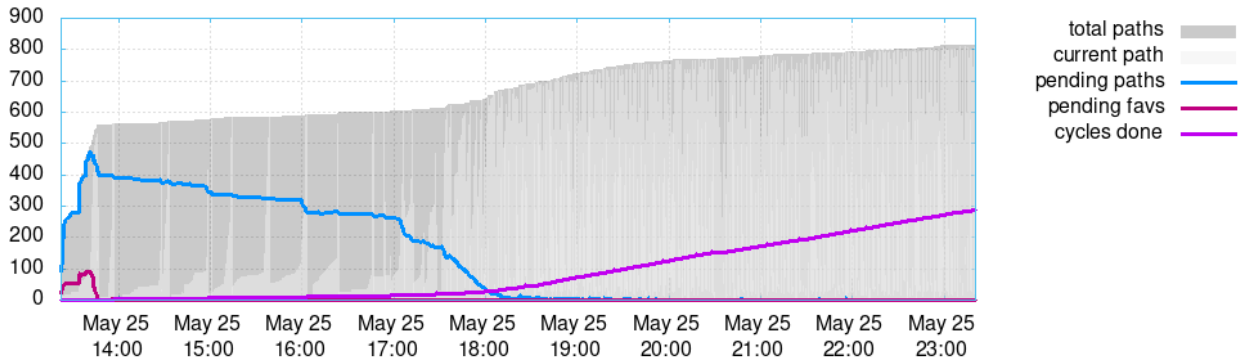

Figure 6.7: An `afl-plot` graph on the first fuzzer run with the largest set of example files

```
fuzzers/afl/afl-fuzz -i $testcasepath -o $outputpath -f pngtest1.png \
-M fuzzer1 -- $pngtestpath --strict @@ > tmpout1.afl &

sleep 60 # -- Allow the master instance to look through the given test cases

fuzzers/afl/afl-fuzz -i $testcasepath -o $outputpath -f pngtest2.png \
-S fuzzer2 -- $pngtestpath --strict @@ > tmpout2.afl &

fuzzers/afl/afl-fuzz -i $testcasepath -o $outputpath -f pngtest3.png \
-S fuzzer3 -- $pngtestpath --strict @@ > tmpout3.afl &
```

Figure 6.8: Bash script launching three instances of AFL.

During the implementation of Bash' AFL instrumentation and compilation, a number of challenges were encountered and overcome, see Section 6.3. This gave us a head start in the implementation of libpng's AFL instrumentation and compilation. As explained in Section 5.3, the dump file we created after our vulnerability insertion was used to feed the LLVM-assembler, which generated a bitcode file. Thereafter, the `afl-clang` wrapper transpiled the bitcode file to an instrumented executable.

6.7 Fuzzing Results

The AFL fuzzer keeps track of fuzzing metrics during runs and supplies an extra tool called `afl-plot`. After a fuzzer run, `afl-plot` can generate a graph including some statistics. Figure 6.7 is such a plot, created from the first fuzzer run with the largest set of example files. It shows a growth in the number of paths found during the fuzzing run of ten hours. AFL also shows the number of its 'pending paths' which are the paths it has discovered with which it wants to investigate further. Its 'pending favs' are the number of its favorite fuzz files, with which it hopes to discover more paths.

The `afl-plot` program creates its plot by reading the `plot_data` file in the fuzzer's output directory. The final data points in these files are used in Chapter 7 to describe the effect of different sizes of test sets on the number of paths found after 10 hours of fuzzing.

```
print("Randomising " + str(number_of_testfiles) + " files")

directory = os.fsencode(dir + "scriptie01")
all_testfiles = os.listdir(directory)
random_indexes = random.sample(range(0, len(all_testfiles)), number_of_testfiles)

for x in random_indexes:
    filename = os.fsdecode(all_testfiles[x])
    copy2(dir + "scriptie01/" + filename, dir + sys.argv[1] + "/" + filename)
```

Figure 6.9: Randomly selecting a number of test files from the set of all possible files.

6.8 Gathering line coverage

In order to gather line coverage on the fuzzer runs we adopted the use of `afl-cov` [mra], a freely available line coverage generator for the AFL fuzzer. After the fuzzer has been stopped, we start `afl-cov`, giving it a reference to the fuzzer's output folder. Then, `afl-cov` takes all generated fuzz and feeds it through an unedited version of `pngtest`. During these runs of `pngtest`, `afl-cov` accumulates all line coverage and then generates a report on the fuzzing run. This report is used in Chapter 7 to describe the results of our fuzzing experiments.

6.9 Parallel Fuzzing

In order to utilize the full potential of the machines running our fuzzer setup, we decided to employ AFL's parallel-fuzzing option. This option lets the users start several AFL processes, marking one of them as 'master'. The fuzzers communicate found paths and promising inputs through a shared directory. Figure 6.8 shows how AFL's parallel fuzzing routine is set up for the experiments done in this research. In Figure 6.8 three instances of `afl-fuzz` are started, their output directed to a temporary log file. The first instance receives the `-M` flag, indicating this is the master instance. It gets 60 seconds to read through and test all example files, a directory of which is indicated by the `$testcasepath` variable. After 60 seconds, two more instances are started with the flag `-S` indicating they will be run in slave mode. All three fuzzers share a common 'working directory', indicated by the `$outputpath` variable. The last variable, `$pngtestpath`, contains the location of our target program.

6.10 Randomizing Test Sets

In Section 7.4.2, another set of fuzzing runs is described. These runs were designed to contain a different set of files each. Twenty different sets, each containing ten example files had to be created at random given the set of 91 possible files, as gathered in Chapter 7. In order to randomize these subsets, the author wrote a small script, as can be seen in Figure 6.9. This same script has been used to randomly select twenty example files for individual testing.

Chapter 7

Experiments

Three different experiments will be performed in order to study the effect the example set size has on the effectiveness of the AFL fuzzer. First, we performed a series of fuzzer runs on different vulnerabilities, deliberately inserted by use of the vulnerability insertion framework described in previous chapters. Each vulnerability was fuzzed three times with example sets of different sizes. The different set sizes are: one PNG file, ten PNG files and one hundred PNG files. In order to create a significant distinction between groups, a factor 10 difference in size was chosen. Only three groups were considered, to represent a small, a medium and a large example set. The largest set is a strict super set of both the medium and small sets, and the medium set is a strict super set of the small set. We attempted to supply diversity in the medium and large example sets by employing files of different sizes, and files that utilize different features of the PNG standard. Secondly, we created twenty new vulnerabilities on which we each fuzzed with an example set size of 10 and once with a set size of one. In contrast to the first experiment, the sets we used were not fixed. Every vulnerability was fuzzed with a randomized set of example files pooled from the original set of the first experiment. This experiment was performed to see if the content of the example files would have an influence on the results of the fuzzing run. Finally, we repeated the first five successful fuzzing runs from Experiment 1, but this time we disabled the Cyclic Redundancy Check (CRC) contained in our subject under test. This experiment is carried out to study the influence of CRC on the results of the fuzzing runs.

The experiments will be performed on Digital Ocean droplets in their AMS3 data center. Each Virtual Machine has access to three virtual CPUs and one gigabyte of RAM. CPU specifications are 'Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz'. After installing the LLVM vulnerability insertion framework we worked on, a shell script will be started manually. This script will start the compilation of libpng with a provided seed. When finished it will start three parallel instances of the AFL fuzzer as explained in Section 6.9. The fuzzers are automatically stopped after fuzzing for 10 wall-clock hours and their results are saved.

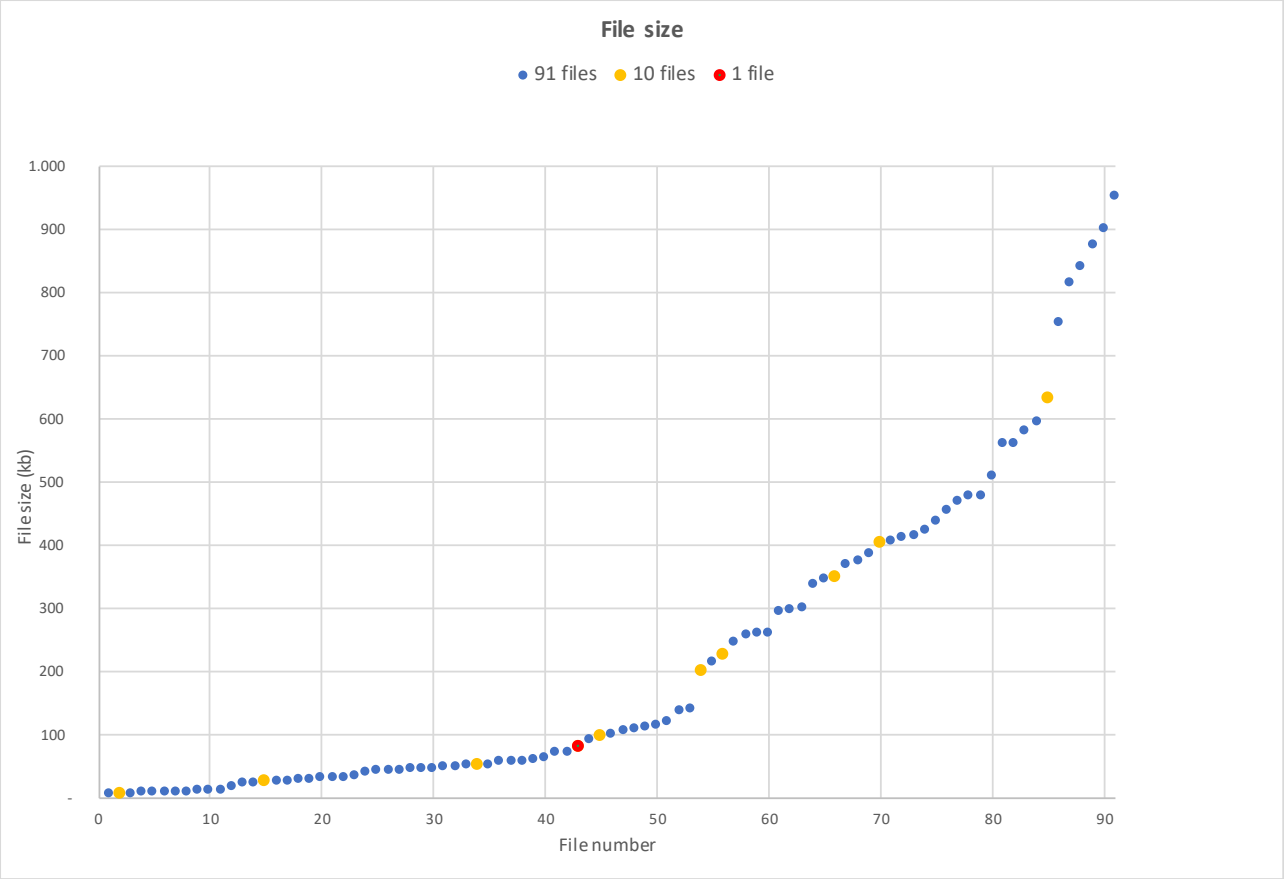

Figure 7.1: The different file sizes present in the example sets

7.1 Example Set Gathering and Diversity

In order to show the impact of the size of an example set on the fuzzing process, we needed to assemble test sets of different sizes. We employed an online search engine's filter ability to search for PNG files only. In order to introduce diversity into the different sets of example files, different PNG features were considered when gathering the files. We tried to gather PNG files of different size (resolution as well as file size), with or without transparency, with or without gamma correction, and with or without text. These are also features that correspond to some of the 18 different chunk types of the PNG standard such as `tRNS`, `gAMA` and `iTXt` [ABB].

The exact image resolution of the used files, along with an indication of their spread over the different example sets is provided in Appendix A. The different file sizes of the PNG files and their distribution over the three initial example sets of the first experiment is shown graphically in Figure 7.1. The presence or absence of textual, gamma and transparency chunks in the different PNG files within the example sets of experiment 1 is shown graphically in Figure 7.2, which gives an indication of the diversity of features within the example sets. These figures were created after the experiments, hence they count 91 files instead of 100, as is explained in Section 7.3.

In the second experiment, the example files used in each fuzzing run are a random subset of the large set of 91 files. These sets were generated as described in Section 6.10.

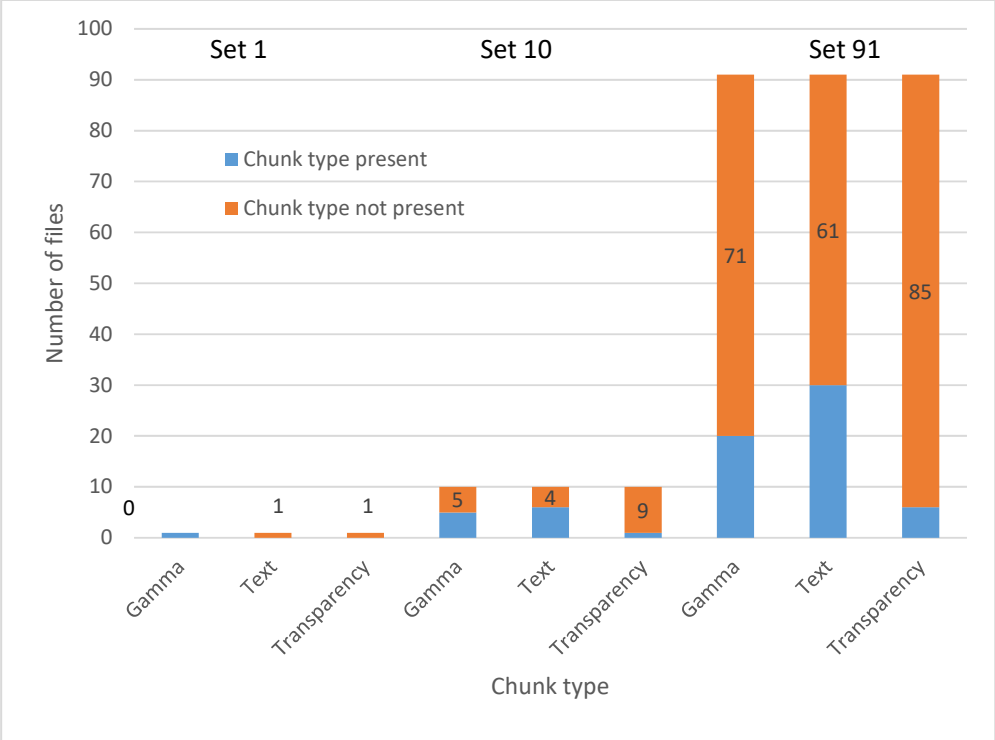

Figure 7.2: The number of PNG files that include a `gAMA`,`tEXt`,`zTXt`,`iTXt` or `tRNS` chunk within the example sets. The number shown in the graph indicates the number of files that do not have the specified feature.

7.2 Analysis approach

During each run, AFL keeps track of statistics. Most specifically, the number of crashes and the number of paths found are recorded. These two metrics can be gathered immediately after the fuzzer has been stopped. Afterwards, we employed 'afl-cov' (see Section 6.8) to gather specifics on line coverage, and whether the line containing the vulnerability has been executed or not. In order to assess the effectiveness of a fuzzing run, consideration of the line coverage percentage is of more importance than the number of paths covered. A larger number of paths found does not necessarily imply that a larger part of the program has been executed, whereas the line coverage percentage represents the actual share of the program that has been covered. The use of 'afl-cov' also enabled us to investigate the specific locations where the vulnerabilities were injected. From this investigation, the number of times the fuzzer executed the line containing the vulnerability is recorded (number of hits). The absolute number of hits is unimportant in most cases. It is more important to know if the line has been executed at all. Furthermore, a hit does not imply that the vulnerability has been found. If the injection always leads to a crash a hit will have triggered it, whereas for instance vulnerabilities that merely leak memory will need a more sophisticated setup to be found. In case of a memory leak, running the subject under test within 'Valgrind' [Val] would be a good way to actually find the vulnerability. Though the line coverage percentage may be more important than the number of paths explored, we still look into both metrics. Combining them gives a broader view on the performance of the fuzzer and since the number of paths found can directly be taken from AFL's output, these could quite easily be analysed. The line coverage serves both as a check to see if the conclusions drawn from the path coverage are valid and as a more precise

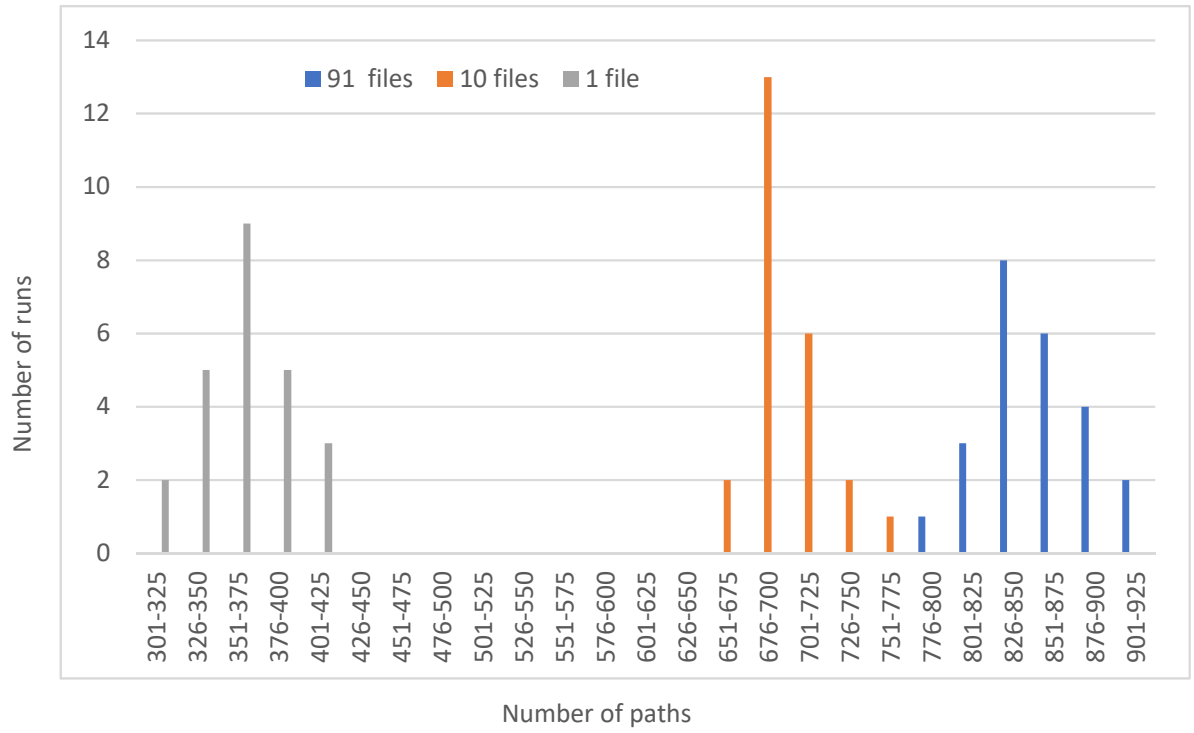

Figure 7.3: The number of runs that resulted in a number of paths grouped by the different test file sets

analysis of the performance of the fuzzer.

7.3 Run time problems

When the first three machines were ready to start the first fuzzer run, the machine with the largest example set could not start fuzzing because AFL claimed that the target program took too long (+100 ms) to process a given example file. Removing nine of the one hundred example files within the largest set seemed to resolve this problem. This led to a large example set of 91 files, with which we continued for the subsequent fuzzer runs.

7.4 Results

In the first experiment 26 fuzzing runs on different vulnerabilities have been performed on three example sets with different sizes. In a second set of runs, it was investigated whether the example set composition affected the outcome of the number of paths found. Finally, a third experiment was conducted, in which we optimized the conditions of the fuzzing runs.

7.4.1 Effect of example set size

The 26 binaries with different vulnerabilities that we created were each fuzzed for 10 wall-clock hours with the three different sets of example files. First we will discuss the results of the path coverage for these runs,

followed by the line coverage analysis. The number of paths found in each fuzzing experiment are collected in Appendix B, Table B.1. The vulnerabilities inserted in runs 3 and 7 were too disruptive in the process and triggered a crash for every one of the example files. The result was that AFL did not process these target programs, and we were forced to start the next fuzzer run with another vulnerability. The different number of paths found for the remaining fuzzer runs is also depicted in Figure 7.3. This figure clearly shows that the three sets do not overlap in the number of paths found.

To determine the significance of the difference in the number of paths found for the different example sets, we conducted a one-way Analysis of Variance (ANOVA). We gathered 24 data points for all three groups of example files, leading to a total of 72 data points considered in the ANOVA. The test shows a significant difference between the groups for the average number of paths discovered ($F = 2191.5, p =< .00001$). In order to check what groups showed a significant difference, an Ad-Hoc Least Significant Difference (LSD) test was performed. The LSD test shows that the average number of paths found for each group differs significantly. The group with 91 files has a higher average number of paths found than the group with 10 files, and the group with 10 files has a higher average number of paths found than the group with 1 file.

As described in Section 7.2, line coverage percentage is the more significant metric to consider. The line coverage percentages for the first experiment are shown in Appendix B, Table B.3. The example set of size one shows line coverage between 15.9% and 17.6% with an average of 16.1% and a standard deviation of 0.3 percentage point. The example set of ten files shows line coverages between 27.8% and 28.1% with an average of 28.0% and a standard deviation of 0.1 percentage point and the example set of 91 files shows line coverages between 29.7% and 30.0% with an average of 29.8% and a standard deviation of 0.1 percentage point. Another Analysis of Variance (ANOVA) has shown that the difference in the numbers of these groups are significant ($F = 32401.5, p =< .00001$).

For each fuzzing run the number of times the line containing the vulnerability was executed is tabulated in Appendix B, Table B.2. As can be seen from this table, only seven of the injected vulnerabilities have been hit by AFL: runs number 2, 6, 14, 15, 20, 22, 23. The set containing only one example file reached significantly fewer of the injected vulnerabilities. These results show that it might not be sufficient to only provide the fuzzer with one example file. However, only one example file has been used; the use of another file with different features can lead to totally different results. By using only one example file in this experiment, we actually tested the quality of that specific file as an example input. In Section 7.4.2 the use of different input files is discussed further. In our experiments, the example set sizes of 91 and 10 both hit the same vulnerabilities.

The results described in this section show that the number of example files has an effect on both the number of paths the AFL fuzzer discovers and the line coverage percentage. Specifically, our results show that a larger example file set leads to an increase in both the number of paths found and line coverage percentage. The different nature of path coverage and line coverage explains the larger growth in the number of paths found relative to the growth in line coverage percentage. Additional path coverage does not have to cause new lines of source code being covered. However, the growth in these metrics from the example set of size 10 to size 91 is rather small. Only seven of the fuzzer runs actually hit the line containing the inserted vulnerability.

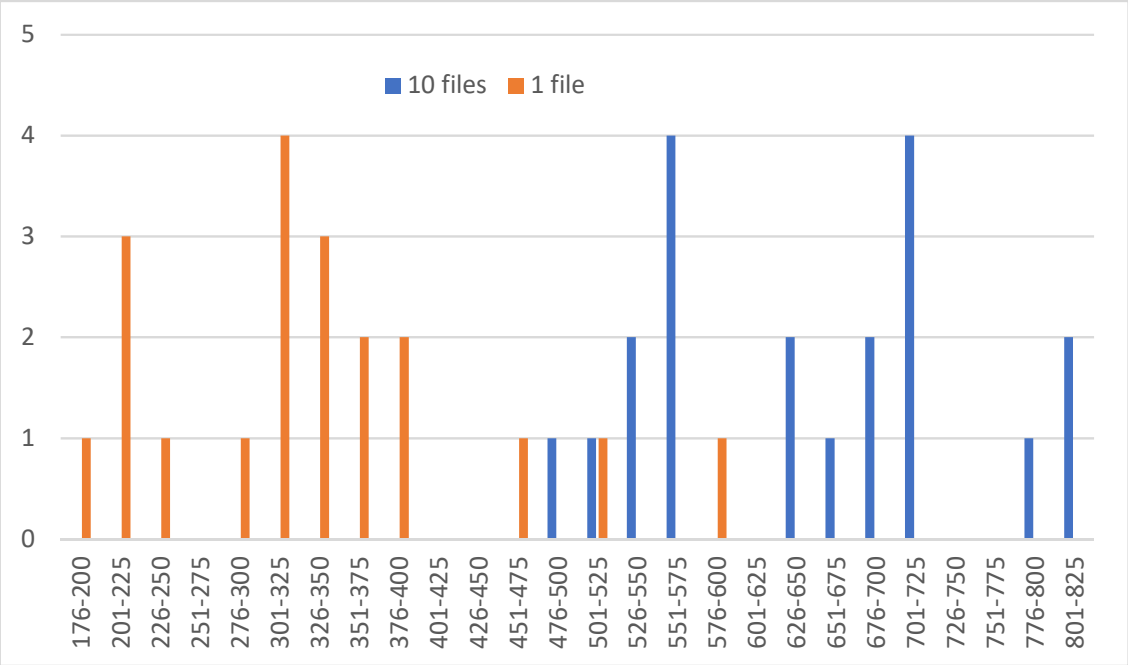

Figure 7.4: The number of runs that resulted in a number of paths grouped by the different test file sets. Please be aware that every run contained a different set of test files

The possible causes of the fact that certain vulnerabilities are not reached at all, as well as the possible reasons AFL did not mark the reached vulnerabilities as crashes is discussed in Section 7.5.

7.4.2 Effect of set composition

The results from the first experiment have been gathered using specific sets of 91, 10 and 1 example files. In order to investigate whether the results are biased due to the selection of files, forty additional fuzzing runs were performed using sets comprising of 1 or 10 example files that were randomly selected from the super set of 91 example files (see Section 6.10). Inadvertently, the runs in this second experiment were run against a different set of vulnerabilities when compared to the first set of fuzzing runs. This means that two parameters are changed with respect to the first experiment. The results of these runs can be found in Appendix C, Table C.1. These numbers are visualized in Figure 7.4.

With the original set of 1 example file – the results described in the previous section – the number of paths found ranges between 308 and 420 with an average of 365. Using twenty different single example files, these numbers range from 194 to 595 with an average of 335 paths found. Similarly, for the original set of 10 example files, the number of paths found ranges between 672 and 751 with an average of 700. Using the twenty randomly generated sets of 10 example files, these numbers range from 477 to 806, with an average of 641. The ranges of the numbers of paths found in these runs are visualized in Figures 7.3 and 7.4. These numbers suggest that the characteristics of files themselves indeed influence the number of paths the fuzzer can find. The line coverage percentages of these runs are shown graphically in Figure 7.5. This figure clearly shows the great inconsistency between example file sets with the same size, suggesting that the contents of these files

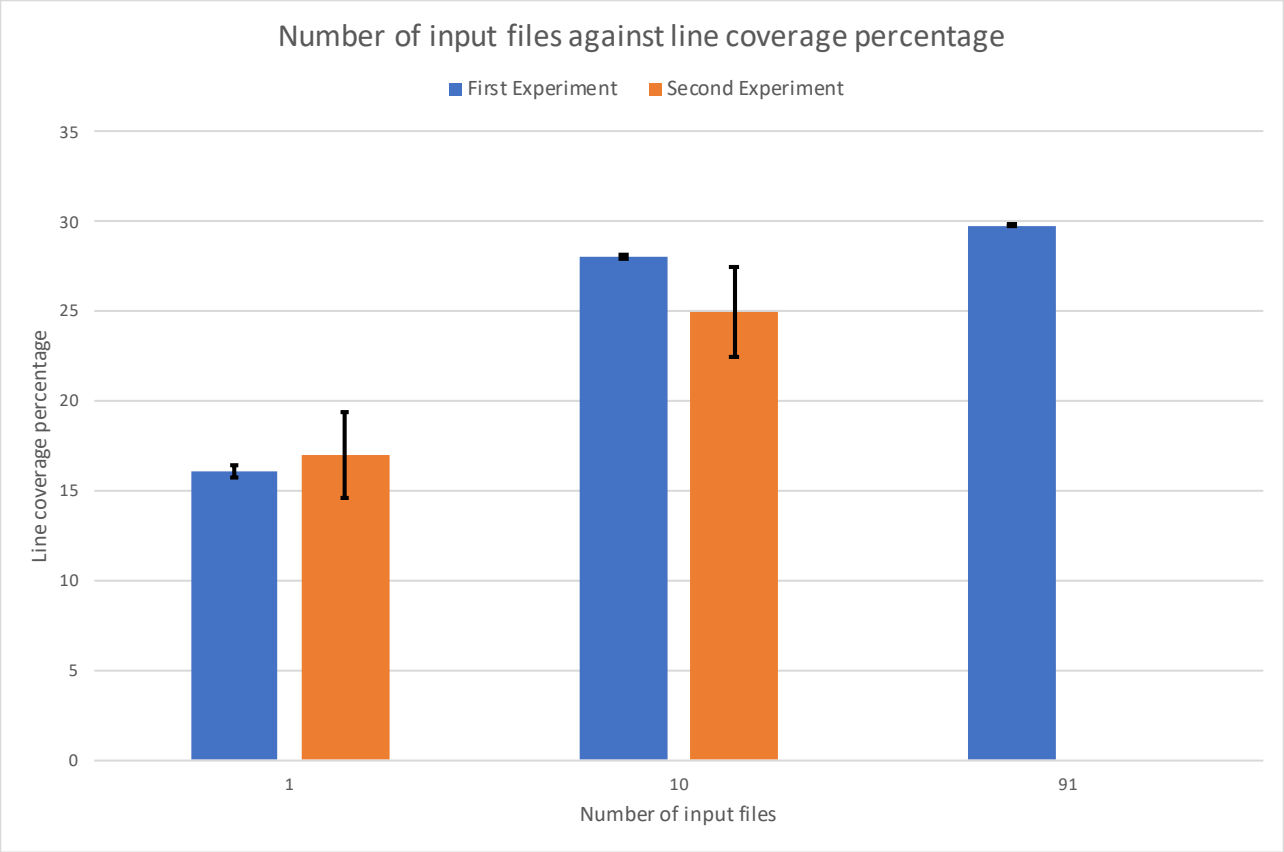

Figure 7.5: Average line coverage percentages and their standard deviations, of experiments one and two

matter more than their numbers.

The line coverage percentages for this second experiment are shown in Appendix C, Table C.3. The example set of size one shows line coverage between 14.1% and 20.5% with an average of 17.0% and a standard deviation of 2.4 percentage point. The example set of ten files shows line coverages between 22.0% and 28.6% with an average of 25.0% and a standard deviation of 2.5 percentage point. Another Analysis of Variance (ANOVA) has shown that the difference in the numbers of these groups are significant ($F = 106.8, p =< .00001$).

For each fuzzing run the number of times the line containing the vulnerability was executed is tabulated in Appendix C, Table C.2. As can be seen from this table, only five of the injected vulnerabilities have been hit by AFL: runs number 3, 8, 9, 18, 19. The set containing only one example file reached significantly fewer of the injected vulnerabilities.

The results described in this section show that the composition of the example file set indeed influences the effectiveness of the fuzzer. Clearly, a larger deviation of the line coverage percentages can be observed with different compositions of example sets.

7.4.3 Disabling the CRC

A Cyclic Redundancy Check (CRC) is a technique commonly used to assure the consistency of data in e.g. binary data files and digital network transmissions. The PNG standard includes this CRC to make sure that the

data saved in a PNG file has not been corrupted. A fuzzer creates fuzz by modifying, inserting, or removing data from an example file. Since a fuzzer is not aware of the presence of a CRC, the generated file will most likely seem corrupted. In the experiments described above, we fuzzed the `pngtest` program without removing this CRC functionality. This probably influenced the effectiveness of the fuzzer, and thereby the results of our experiments, because `pngtest` discards a PNG file that does not pass the CRC, whereas a valid PNG is processed with a wide range of library functions. The CRC could be an explanation for the low number of vulnerabilities that have been hit in the experiments described in previous sections. By removing the CRC, the files generated by the fuzzer will not be intercepted by the CRC upfront, meaning that a larger share of the generated files is run through the entirety of the test program, likely increasing code coverage and number of vulnerabilities that are hit.

In order to validate the results described in the previous section, the first five successful experiments on the three example sets with different sizes were repeated with the CRC disabled. The results are provided in Appendix D, tables D.1, D.2 and D.3.

Comparing the results of the runs with and without the CRC enabled, a number of differences can be observed. For one, the number of paths found increased significantly for all three set sizes. The line coverage also increased for all sets, with five to six percentage points. Most notably, without the CRC, runs four and five show that the fuzzer has passed the line containing the vulnerability, where this did not happen when the CRC was still active. Be aware that a firm conclusion cannot be drawn, since the limited number of five experiments might bias the outcome.

7.5 Discussion

The aim of the research described in this thesis was to find out whether the example set size would have an influence on the effectiveness of the AFL fuzzer. We measured the fuzzer's effectiveness by its ability to find the vulnerability and by its line and path coverage. From the results described in Section 7.4 some general observations can be described. The inserted vulnerability does not have an influence on the line and path coverage reached by AFL, because the vulnerability does not play a role in the execution path a certain input takes within the target program.

From the results of Section 7.4, it is apparent that many of the injected vulnerabilities were not reached for any of the given example sets, irrespective of the presence of CRC. This can be due to the fact that the injected function was not used by our target program. This could technically happen, since we injected a library (libpng) with a vulnerability, and if our target program (pngtest) happens to not use a function in this library, that function would never be reached in our fuzzing runs. A second (more likely) reason could be that the fuzzer has never generated the input that would lead the execution path to pass the injected vulnerability. For two of the three experiments, the reason our vulnerability was not hit could also be due to the CRC intercepting the generated fuzz before critical parts of the library would be executed.

According to the line coverage reports, some vulnerabilities were in fact reached by the fuzzer. However, with

```
while (iin < PNG_MAX_ERROR_TEXT-1 && error_message[iin] != '\0')
    buffer[iout++] = error_message[iin++];

/* iin < PNG_MAX_ERROR_TEXT, so the following is safe: */
buffer[iout] = '\0';
```

Figure 7.6: `libpng, pngerror.c: png_format_buffer()`

none of the runs the fuzzer encountered a crash of the target program, which could be due to two reasons. For one, not all types of vulnerabilities cause the program to crash. A memory leak – caused by any of the `alloca` injections from Section 7.4 – could simply result in some undefined behavior without a segmentation fault. In order to catch these types of bugs, we could have fuzzed our target program within e.g. Valgrind, which returns a positive exit status when it encounters a memory leak, causing the fuzzer to mark the execution as failed. A second reason that the fuzzer did not mark the injected change in the program can be because the alteration we made did not create an actual vulnerability. Two of the fuzzer runs in Section 7.4 reached vulnerabilities that were not of the `alloca` type. When we looked into these injected vulnerabilities, we noticed that one was actually impossible to trigger (Table D.2, run #4 – strncpy) and the other highly unlikely to be triggered (Table C.2, run #3 – `offbyn:loop`). This is due to the nature of the vulnerabilities injected in these runs. The `strncpy` injection swaps the usage of the `strncpy` function call with the `strcpy` call. In some cases this might lead to a buffer overflow but in this particular injection the source string was a constant and thus had a fixed length, which fitted the destination. As described in Section 2.2, the `offbyn:loop` injection increases the loop guard with one. In this particular case, this is still highly unlikely to trigger undefined behavior. Looking at the code in Figure 7.6, it is apparent that if we were to increase the guard in the first condition of the while loop, the final write to `buffer` would be out of bounds. However, the while loop contains a second condition which most likely causes the loop to break before the guard in the first condition is reached.

The question remains why some vulnerabilities were not reached by the fuzzer at all. Consider run #4 from Table B.2. We know this vulnerability can actually be reached, as demonstrated by run #4, Table D.2. Possible reasons why this vulnerability was not reached in experiment one are described above, although the fact that the same vulnerability actually was reached in the third experiment suggests that the CRC prevented the fuzzer from reaching the vulnerability in the first experiment. Runs #9, #13 and #21 fuzzed a version of `libpng` whereby a vulnerability was inserted within the `png_write_pCAL` method. As none of our example files contain the pCAL chunk type, it is clear that the fuzzer would not be able to locate this vulnerability in such short time, although theoretically it would be possible to reach the line containing the vulnerability.

Looking at the results from the second experiment, it is clear that a diverse composition of the example file set has a larger influence on the line coverage and path coverage than the size of the example file set. Whereas line coverage for the example file set of size one ranges from 14.1% to 22.2%, the example set containing 10 files results in a line coverage of 20.2% to 28.6%. This seams to indicate that the quality of the input file is more important than the number of input files.

Chapter 8

Conclusion

This thesis aimed to work out the effect the example set size has on the effectiveness of the AFL fuzzer, specifically whether it would benefit the total path and line coverage as well as its ability to locate an injected vulnerability. By creating two compiler passes within a framework around the Clang compiler, we were able to insert vulnerabilities into the existing libpng library. The vulnerable programs could then be used as targets in several series of fuzzer runs. In the first experiment 26 fuzzing runs were performed using three example sets, each with a different number of files. These experiments showed an increase in both path coverage and line coverage with increasing example set sizes. Based on initial results, it appeared that the use of an example set containing only one file is generally not sufficient to effectively locate the vulnerability. However, we later showed that the use of different single-file example sets provided different results, meaning that the quality of the file has a large influence on the effectiveness of the fuzzer.

As these initial experiments were all run with fixed sets of example files, a second set of experiments was carried out using randomized example sets, as to lower the probability that our previous results were biased by a specific file. The results from these runs indeed show that the composition of an example file set influences the effectiveness of a fuzzer; it appears that a diverse composition of the example file set has a larger influence on the line coverage and path coverage than the size of the example file set. Nevertheless, the example sets containing 10 files appear to have higher chances of reaching the vulnerability when compared to example sets containing one file.

Inadvertently, in the first two experiments the CRC was not disabled in our target program, leading to our results showing an underrepresentation of the fuzzer's capabilities. Thus, finally, we performed a set of runs – again, with three example set sizes – in which the CRC in the target program was disabled. This led to higher code coverage metrics for both line and path coverage. Whereas in the initial experiments, without the CRC only two of the five vulnerabilities were located, four out of five vulnerabilities were reached in the runs with CRC disabled.

A larger example set size generally leads to larger path and line coverage, although not linearly with the number of files: the use of example sets containing 91 or 10 files did not lead to large differences in line and

path coverage. This appears also the case when the CRC is disabled. Both sets (91 and 10) located the same vulnerabilities.

Based on the first experiment described in this thesis, it may be concluded that an example set with a limited set of files may be sufficient to locate vulnerabilities. However experiment 2 shows that the quality of the files in the example set have a more significant influence on the effectiveness of the fuzzer. Putting the costs of gathering, and fuzzing with, a large number of example files against the benefits of the higher code coverage reached by the fuzzer, one could deduce an optimal number of input files provided one takes into consideration the diversity within the input files.

8.1 Future Work

The research described in this thesis considers the quantity of seed files, whereas other papers investigate the optimal quality of seed files. The quality of seed files is indeed important, as can be seen from the higher outliers in the results displayed in Appendix C, table C.1 (e.g. entries 2 and 15). However, also the quantity of seed files in the example set can be optimized. One should try to find a good balance between the two, benefitting of both the low effort of inserting many seed files and the fuzzing capabilities of specially-crafted seed files.

The third experiment in this thesis showed that the original methodology was flawed. Fuzzing a program that includes a CRC is ineffective when compared to the same fuzzing run without this check. Future research should take care to disable CRC in their target program before fuzzing. Additionally, in order to register memory leaks within the fuzzing process, one could consider running the target program within Valgrind, although this might slow down the fuzzing process due to overhead.

The fuzzing runs in this thesis have taken exactly 10 hours each. The impact of this parameter on the results is not considered in this thesis. If all three sizes of example sets were for example given 100 hours of fuzzing time, would the average number of paths be closer together, would they be further apart? This question might be worth investigating in future research.

Bibliography

[ABB] Mark Adler, Thomas Boutell, and John Bowler. Portable network graphics (png) specification. `https://www.w3.org/TR/PNG/#4Concepts.FormatTypes`. Accessed: 2019-07-10.

[CZZ+19] Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. Optimizing seed inputs in fuzzing with machine learning. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, pages 244–245. IEEE Press, 2019.

[GMH+18] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *arXiv preprint arXiv:1810.08289*, 2018.

[gnu] The gnu compiler collection. `https://gcc.gnu.org/`. Accessed: 2019-07-07.

[GPS17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

[GZQ+18] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[Ham] V. den Hamer. Hiding in plain sight: How location affects memory error detectability by fuzzers. *Thesis Bachelor Informatica, Leiden University, 2019*.

[HJP+16] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517–528. ACM, 2016.

[lca] lcamtuf. Technical "whitepaper" for afl-fuzz. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed: 2019-07-07.

[lib] libpng. libpng, the official png reference library. `http://www.libpng.org/pub/png/libpng.html`. Accessed: 2019-06-25.

[llva] The llvm compiler infrastructure. `http://llvm.org/`. Accessed: 2019-07-07.

[llvb] The often misunderstood gep instruction. https://www.llvm.org/docs/GetElementPtr.html#introduction. Accessed: 2019-07-11.

[llvc] Writing an llvm pass. https://www.llvm.org/docs/WritingAnLLVMPass.html. Accessed: 2019-07-11.

[Lop] American Fuzzy Lop. Afl bug-o-rama trophy case. http://lcamtuf.coredump.cx/afl/#bugs. Accessed: 2019-05-25.

[Mil] Prof. Barton Miller. How the term 'fuzz' was coined. http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html. Accessed: 2019-05-25.

[mra] mrash. afl-cov - afl fuzzing code coverage. https://github.com/mrash/afl-cov. Accessed: 2019-09-08.

[QEM] QEMU. Qemu, the fast processor emulator. http://qemu.org/. Accessed: 2019-05-25.

[Rij] V. van Rijn. Quantifying fuzzer performance on spatial and temporal memory errors. *Thesis Bachelor Informatica, Leiden University, 2018*.

[SPWS14] Laszlo Szekeres, Mathias Payer, Lenx Tao Wei, and R Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014.

[SZ18] Maksim O. Shudrak and Vyacheslav Zolotarev. Improving fuzzing using software complexity metrics. *CoRR*, abs/1807.01838, 2018.

[Val] Valgrind. Valgrind - system for debugging and profiling linux programs. http://valgrind.org/. Accessed: 2019-09-14.

[VDKNG17] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419. ACM, 2017.

[Whe14] David A Wheeler. Preventing heartbleed. *IEEE Computer*, 47(8):80–83, 2014.

Appendix A

Resolutions of Used PNG Files

Table A.1: The resolutions for every image in the test sets.

File nr.	Resolution	File nr.	Resolution	File nr.	Resolution
1	105x105	32	512x512	62	917x612
2	194x260	33	550x580	63	924x720
3	200x200	34	563x329	64	960x371
4	218x231	35	586x300	65	960x380
5	220x220	36	596x411	66	960x422
6	245x205	37	600x320	67	960x480
7	250x250	38	600x700	68	992x992
8	255x221	39	613x399	69	1000x350
9	256x256	40	619x420	70	1024x576
10	256x256	41	620x221	71	1024x768
11	259x195	42	625x420	72	1024x1193
12	260x600	43	638x359	73	1024x1240
13	266x279	44	640x360	74	1026x913
14	299x600	45	640x480	75	1142x704
15	300x210	46	678x600	76	1191x460
16	300x300	47	725x457	77	1200x800
17	300x300	48	727x436	78	1240x1600
18	302x128	49	728x380	79	1280x820
19	302x512	50	730x779	80	1359x1379
20	325x359	51	750x1112	81	1452x900
21	380x500	52	768x430	82	1479x783
22	400x215	53	799x531	83	1493x1046
23	400x360	54	800x565	84	1500x1500
24	400x387	55	800x798	85	1587x907
25	439x294	56	811x340	86	1600x980
26	475x594	57	816x980	87	1920x1080
27	480x480	58	900x648	88	1920x1080
28	499x355	59	900x675	89	1920x1080
29	500x375	60	900x900	90	1955x600
30	512x387	61	902x569	91	2400x3100
31	512x480				

- included in 1 & 10 files set
- included in 10 files set

Appendix B

Effect of example set size

Table B.1: The number of different paths found for the given example set sizes per iteration

#	Vulnerability type	Injected Function	Paths 91	Paths 10	Paths 1
1	allocation:alloca	png_do_unshift	856	729	341
2	allocation:alloca	png_handle_pHYs	813	718	383
3	offbyn:loop	png_write_info			
4	unsafec:strncpy	test_one_file	878	704	338
5	allocation:alloca	png_write_sPLT	840	696	308
6	allocation:alloca	png_chunk_unknown_handling	851	688	339
7	allocation:alloca	png_write_IHDR			
8	offbyn:loop	png_init_read_transformations	840	680	342
9	offbyn:loop	png_write_pCAL	833	698	400
10	offbyn:loop	png_set_hIST	791	709	410
11	allocation:alloca	png_do_shift	825	721	366
12	offbyn:loop	png_init_palette_transformations	852	692	366
13	allocation:alloca	png_write_pCAL	841	700	356
14	allocation:alloca	png_formatted_warning	831	686	364
15	allocation:alloca	png_write_bKGD	822	702	373
16	offbyn:loop	png_destroy_gamma_table	924	688	379
17	offbyn:loop	png_destroy_gamma_table	858	699	352
18	offbyn:loop	png_destroy_gamma_table	851	700	308
19	allocation:alloca	png_handle_oFFs	846	751	373
20	allocation:alloca	png_handle_tIME	847	694	389
21	allocation:alloca	png_write_pCAL	877	688	360
22	allocation:alloca	write_sTER_chunk	842	679	379
23	allocation:alloca	png_read_IDAT_data	872	708	364
24	offbyn:loop	png_write_info	879	672	340
25	allocation:alloca	png_user_version_check	876	732	420
26	allocation:alloca	png_do_shift	901	675	410

Table B.2: The number of times the line containing the vulnerability is executed for the given example set sizes per iteration

#	Vulnerability type	Injected function	Hits 91	Hits 10	Hits 1
1	allocation:alloca	png_do_unshift	0	0	0
2	allocation:alloca	png_handle_pHYs	620	249	0
3	offbyn:loop	png_write_info			
4	unsafec:strncpy	test_one_file	0	0	0
5	allocation:alloca	png_write_sPLT	0	0	0
6	allocation:alloca	png_chunk_unknown_handling	7240	11076	9328
7	allocation:alloca	png_write_IHDR			
8	offbyn:loop	png_init_read_transformations	0	0	0
9	offbyn:loop	png_write_pCAL	0	0	0
10	offbyn:loop	png_set_hIST	0	0	0
11	allocation:alloca	png_do_shift	0	0	0
12	offbyn:loop	png_init_palette_transformations	0	0	0
13	allocation:alloca	png_write_pCAL	0	0	0
14	allocation:alloca	png_formatted_warning	44	62	0
15	allocation:alloca	png_write_bKGD	224	226	0
16	offbyn:loop	png_destroy_gamma_table	0	0	0
17	offbyn:loop	png_destroy_gamma_table	0	0	0
18	offbyn:loop	png_destroy_gamma_table	0	0	0
19	allocation:alloca	png_handle_oFFs	0	0	0
20	allocation:alloca	png_handle_tIME	159	215	0
21	allocation:alloca	png_write_pCAL	0	0	0
22	allocation:alloca	write_sTER_chunk	3	6	4
23	allocation:alloca	png_read_IDAT_data	342724	141670	41203
24	offbyn:loop	png_write_info	0	0	0
25	allocation:alloca	png_user_version_check	0	0	0
26	allocation:alloca	png_do_shift	0	0	0

Table B.3: The percentage of lines covered for the given example set sizes per iteration

#	Vulnerability type	Injected function	L.cov % 91	L.cov % 10	L.cov % 1
1	allocation:alloca	png_do_unshift	29.9	28.1	17.6
2	allocation:alloca	png_handle_pHYs	29.7	28.1	16.1
3	offbyn:loop	png_write_info			
4	unsafec:strncpy	test_one_file	29.8	28.1	16.1
5	allocation:alloca	png_write_sPLT	29.7	28.0	15.9
6	allocation:alloca	png_chunk_unknown_handling	29.8	28.1	16.0
7	allocation:alloca	png_write_IHDR			
8	offbyn:loop	png_init_read_transformations	29.9	28.0	16.2
9	offbyn:loop	png_write_pCAL	29.8	27.8	16.0
10	offbyn:loop	png_set_hIST	29.8	28.0	16.1
11	allocation:alloca	png_do_shift	29.7	28.1	16.2
12	offbyn:loop	png_init_palette_transformations	29.8	27.9	16.0
13	allocation:alloca	png_write_pCAL	29.9	28.1	16.0
14	allocation:alloca	png_formatted_warning	29.8	28.1	16.0
15	allocation:alloca	png_write_bKGD	29.7	28.1	16.0
16	offbyn:loop	png_destroy_gamma_table	29.9	28.1	16.1
17	offbyn:loop	png_destroy_gamma_table	29.8	28.0	15.9
18	offbyn:loop	png_destroy_gamma_table	29.8	28.0	16.1
19	allocation:alloca	png_handle_oFFs	29.8	28.2	16.1
20	allocation:alloca	png_handle_tIME	29.9	27.9	16.3
21	allocation:alloca	png_write_pCAL	29.9	28.1	16.1
22	allocation:alloca	write_sTER_chunk	29.8	28.0	16.1
23	allocation:alloca	png_read_IDAT_data	30.0	28.0	16.0
24	offbyn:loop	png_write_info	29.9	28.1	16.1
25	allocation:alloca	png_user_version_check	29.9	28.0	16.1
26	allocation:alloca	png_do_shift	29.9	28.1	16.4

Appendix C

Effect of set composition

Table C.1: The number of different paths found for randomized subsets of example input

#	Vulnerability type	Injected function	Paths 10	Paths 1
301	offbyn:loop	png_do_unshift	530	217
302	offbyn:loop	png_set_pCAL	803	595
303	offbyn:loop	png_format_buffer	709	394
304	allocation:alloca	png_write_hIST	561	315
305	offbyn:loop	png_build_8bit_table	806	375
306	offbyn:loop	png_set_eXIf_1	677	326
307	allocation:alloca	png_handle_tRNS	707	301
308	allocation:alloca	write_vpAg_chunk	629	313
309	allocation:alloca	png_write_pHYs	712	348
310	allocation:alloca	png_handle_hIST	569	203
401	offbyn:loop	png_do_unshift	523	332
402	offbyn:loop	png_set_pCAL	560	459
403	offbyn:strncopy	test_one_file	722	370
404	allocation:alloca	png_write_hIST	529	234
405	offbyn:loop	png_build_8bit_table	784	507
406	allocation:alloca	png_handle_iCCP	639	194
407	allocation:alloca	png_handle_tRNS	677	298
408	allocation:alloca	write_vpAg_chunk	551	392
409	allocation:alloca	png_write_pHYs	477	220
410	allocation:alloca	png_handle_hIST	665	306

Table C.2: The number of times the vulnerability was reached for randomized subsets of example input

#	Vulnerability type	Injected function	Hits 10	Hits 1
301	offbyn:loop	png_do_unshift	0	0
302	offbyn:loop	png_set_pCAL	0	0
303	offbyn:loop	png_format_buffer	6653	8031
304	allocation:alloca	png_write_hIST	0	0
305	offbyn:loop	png_build_8bit_table	0	0
306	offbyn:loop	png_set_eXIf_1	0	0
307	allocation:alloca	png_handle_tRNS	0	0
308	allocation:alloca	write_vpAg_chunk	24	0
309	allocation:alloca	png_write_pHYs	388	0
310	allocation:alloca	png_handle_hIST	0	0
401	offbyn:loop	png_do_unshift	0	0
402	offbyn:loop	png_set_pCAL	0	0
403	offbyn:strncopy	test_one_file	0	0
404	allocation:alloca	png_write_hIST	0	0
405	offbyn:loop	png_build_8bit_table	0	0
406	allocation:alloca	png_handle_iCCP	0	0
407	allocation:alloca	png_handle_tRNS	0	0
408	allocation:alloca	write_vpAg_chunk	26	4
409	allocation:alloca	png_write_pHYs	55	0
410	allocation:alloca	png_handle_hIST	0	0

Table C.3: The line coverage percentages found for randomized subsets of example input

#	Vulnerability type	Injected function	L.cov % 10	L.cov % 1
301	offbyn:loop	png_do_unshift	22.0	15.0
302	offbyn:loop	png_set_pCAL	28.6	22.2
303	offbyn:loop	png_format_buffer	25.3	19.2
304	allocation:alloca	png_write_hIST	20.6	15.1
305	offbyn:loop	png_build_8bit_table	27.8	19.2
306	offbyn:loop	png_set_eXIf_1	23.8	15.1
307	allocation:alloca	png_handle_tRNS	27.2	14.2
308	allocation:alloca	write_vpAg_chunk	27.9	18.8
309	allocation:alloca	png_write_pHYs	28.1	15.3
310	allocation:alloca	png_handle_hIST	25.1	15.1
401	offbyn:loop	png_do_unshift	23.3	17.4
402	offbyn:loop	png_set_pCAL	23.0	20.5
403	offbyn:strncopy	test_one_file	26.2	17.5
404	allocation:alloca	png_write_hIST	22.0	14.9
405	offbyn:loop	png_build_8bit_table	26.2	20.5
406	allocation:alloca	png_handle_iCCP	24.1	14.1
407	allocation:alloca	png_handle_tRNS	26.4	14.8
408	allocation:alloca	write_vpAg_chunk	25.8	17.6
409	allocation:alloca	png_write_pHYs	20.2	16.0
410	allocation:alloca	png_handle_hIST	26.3	16.6

Appendix D

CRC-less fuzzing runs

Table D.1: The number of different paths found for the given example set sizes per iteration

#	Vulnerability type	Injected function	Paths 91	Paths 10	Paths 1
1	allocation:alloca	png_do_unshift	1738	1890	1004
2	allocation:alloca	png_handle_pHYs	1723	1928	954
3	offbyn:loop	png_write_info			
4	unsafec:strncpy	test_one_file	1760	1799	890
5	allocation:alloca	png_write_IHDR	1663	1830	933
6	allocation:alloca	png_chuck_unknown_handling	1609	1764	941

Table D.2: The number of times the line containing the vulnerability is executed for the given example set sizes per iteration

#	Vulnerability type	Injected function	Hits 91	Hits 10	Hits 1
1	allocation:alloca	png_do_unshift	0	0	0
2	allocation:alloca	png_handle_pHYs	911	317	0
3	offbyn:loop	png_write_info			
4	unsafec:strncpy	test_one_file	8	2	0
5	allocation:alloca	png_write_IHDR	0	34	0
6	allocation:alloca	png_chuck_unknown_handling	11989	21199	5893

Table D.3: The percentage of lines covered for the given example set sizes per iteration

#	Vulnerability type	Injected function	L.cov % 91	L.cov % 10	L.cov % 1
1	allocation:alloca	png_do_unshift	35.1	35.2	21.1
2	allocation:alloca	png_handle_pHYs	34.9	35.2	20.9
3	offbyn:loop	png_write_info			
4	unsafec:strncpy	test_one_file	35.8	35.3	21.0
5	allocation:alloca	png_write_IHDR	35.4	34.7	20.9
6	allocation:alloca	png_chuck_unknown_handling	35.3	34.7	21.0