Universiteit Leiden
The Netherlands

Opleiding Informatica

Quantifying the performance of fuzzers

in the detection of security threats

Levi Vos

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

BACHELOR THESIS

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

12/08/2018

Abstract

Programmers are prone to introduce bugs when creating C programs. Some of these bugs are security threats. Fuzzing is a technique to detect these security threats. This thesis sets out to quantify the performance of fuzzers in the detection of security threats. This thesis has shown complex programs take more time to fuzz in order to detect security threats. As well as, how various code constructs affect the effectiveness of a fuzzer. This has been done by writing a random C program generator, which generates programs with varying complexities and one security threat. This thesis provides a metric for determining the minimal amount of time needed for effective fuzzing. As well as, how to generate random, succesfully terminating and satisfiable C programs.

Contents

Chapter 1

Introduction

A programmer is during the creation of a C program, due to various reasons, prone to introduce bugs [1] and errors. Some examples why this might occur include among other reasons: lack of knowledge by the programmer; carelessness by the programmer; insufficient testing; and, complexity of the program. Usually the compiler performs some type of error checking, and will not produce a working executable upon the detection of errors. Bugs, on the other hand, go unnoticed by the compiler, and only identify themselves in unexpected or undefined behaviour during the execution of the program. Usually, such a bug is relatively harmless. Sometimes however, such a bug leads to a security threat with unforeseen consequenses.

Notable techniques for detecting bugs in a computer program are formal verification [2] and testing. Verification of a program determines whether a computer program is correct, and thus free of bugs. However it is a costly process, proving that a program satisfies a formal specification of its behaviour. Testing, on the other hand, is faster, but imperfect and can not guarantee the absence of bugs. In practice, mostly testing is used, since it does not impose such a big time constraint on the development of a program as formal verification does.

Fuzzing [3], or fuzz testing, is an automated software testing technique that covers numerous boundary cases using invalid, unexpected or random data as input, and then systematically identifying the failures, such as crashes or security vulnerabilities, that arise. Fuzzing provides a helping hand in the testing of software, since a programmer is prone to test only the use cases for which the program is designed. An effective fuzzer generates semi-valid inputs that are "valid enough" in that they are not directly rejected by the program, but do create unexpected behaviours deeper in the program.

A fuzzer might be very valuable to the programmer in the detection of security threats, since these threats usually occur in unconsidered use-cases. *The goal of this research is therefore to quantify how well fuzzers perform in the detection of security threats.* This research will mainly focus on the context in which a security threat is placed, and how differences in context result in differences in detection rates.

Because formal verification imposes such a big time constraint on the development of a computer program, testing, and more specifically, fuzzing, is mostly used for the detection of security threats in computer programs.

Therefore, it is of great importance to have a metric by which various fuzzers can be compared to eachother. This gives a programmer insight into which fuzzer is the best for detecting security threats in programs. More complex programs usually take more time to fuzz in order to detect security threats, since these take up more time than a less complex program. Time is of great importance, since being able to do more test runs in the same timeframe has a higher probability of detecting security threats.

1.1 Contributions

This thesis provides the following contributions:

- how to generate satisfiable C programs using the C grammar;
- how various code constructs affect the effectiveness of a fuzzer;
- how the amount of time spent fuzzing affect the effectiveness of a fuzzer;
- and, a metric by which various fuzzers can be compared to eachother.

1.2 Thesis Overview

This chapter contains the introduction; Chapter 2 contains some background information needed for reading this thesis; Chapter 3 discusses related work; Chapter 4 discusses how this experiment is set up; Chapter 5 evaluates the results of the experiment; Chapter 6 concludes this thesis. The chapters about the design and evaluation are particularly important, because these explain the generation of the C programs, and most importantly, how to guarantee the security threat can be reached by the fuzzer.

This is a bachelor thesis, by Levi Vos for the Leiden Institute of Advanced Computer Science, supervised by Dr E. van der Kouwe and Dr. K.F.D. Rietveld.

Chapter 2

Background

2.1 Software Security Threats

There are a multitude of security threats possible in software. One of the most common security threats discovered and exploited, is the stack-based buffer overflow [4]. A buffer is a piece of allocated computer memory also known as an array.

A stack-based buffer is allocated at run-time on the stack, a last in, first out datastructure which consists of information about the active subroutines of a computer program. Upon calling a subroutine a new stack frame is pushed onto the stack and the program continues execution in the subroutine.

A stack frame consists of the parameters used by the subroutine; some control information, such as the *return address*, to which control flow must be reverted upon termination of the subroutine; and, local variables of the subroutine, such as buffers. The layout of a stack frame for a call to a simple subroutine, in Figure 2.1, can be seen in Figure 2.2.

```
void function(int a, int b, int c) {
     char buffer1[5];
     char buffer2[10];
}


void main() {
    function(1,2,3);
}
```

Figure 2.1: A simple C program used for demonstrating the resulting stack layout in Figure 2.2.

To overflow means to fill something with more content than it can handle. Imagine filling a bucket with water until it pours over the edge. Thus, a buffer overflow, is defined as writing data past the bounds allocated for the buffer, and thus overwriting important control data on the stack. A buffer overflow vulner-

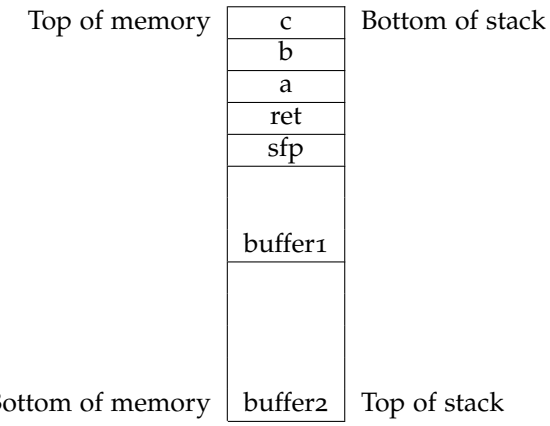

Figure 2.2: Resulting stack layout of the example in Figure 2.1.

ability can be exploited by an attacker by writing more data to a buffer than is allocated for it on the stack. Choosing the data with which to overflow, such that the return address gets overwritten with an address pointing to the attackers own code. Then upon termination of the subroutine, control flow gets reverted to the address contained in the *return address* of the stack frame. However, by choosing the overflow data wisely, the attacker can point this *return address* to their own code, and thus taking over the control flow of the program.

Buffer overflows are possible because popular programming languages, such as C or C++ do not check by design whether or not a buffer gets overflowed. This responsibility is entrusted onto the programmer, however humans are not perfect and make mistakes. Defences are made against buffer overflows, however new ways to get around these defences are being made by attackers as well [5] [6].

Aside from buffer overflows, there are a lot of different types of software security vulnerabilities [7]. Some examples include, among others: off-by-one overflows; non-terminated string overflows; integer overflows; dangling pointers; uninitialized variables; and, heap overflows.

2.1.1 Example of a Security Threat

An example of a security threat can be seen in Figure 2.3. This security threat has the potential to be exploited via a buffer overflow. This is possible, because the function *scanf* copies a string from the *stdin* of the program into *BUFFER*. However *scanf* does not check whether or not the copied string fits in *BUFFER*, thus resulting in an overflow when the copied string is larger than the allocated space.

```
void vuln(){
    char BUFFER[32];
    scanf("%s", BUFFER);
}
```

Figure 2.3: A simple security threat.

2.2 Software Security Assessment methods

Looking for bugs in software, and especially bugs with security implications, is essential for different reasons, namely: quality assurance of software; finding flaws so that they can be patched, making the software more secure; and, for finding flaws such that they can be exploited for fun, profit or another reason for exploiting software vulnerabilities.

As is described in [8], there are three approaches to testing software for security threats, namely: black box; white box; and, grey box. These approaches differ, most notably, in the amount of resources they have to their disposal. As well as, their cost (in both time and economical sense).

Black box testing is the simplest approach to testing. In this approach, some input is given to a program without knowledge of its inner working, the black box, and consecutively observing the output that emerges out of the box, such as a successful execution or a crash. This approach only gives insight into what kind of inputs trigger a certain output, however not exactly why this output was triggered. Black box testing is always applicable and is very simple in practice, however the amount of code that is covered in the test can be very low. Fuzzing is a type of testing that falls under this category.

White box testing is the complete opposite of black box testing, because this approach has access to all the resources available, including the source code. For this approach to work, source code has to be available. However, with source code available this approach gives complete coverage of the tested code, but is costly as well. Formal verification is an example of white box testing.

The grey box approach floats somewhere inbetween black box and white box approaches. Grey box testing is defined as black box testing with additional insights available from white box testing. Grey box does not make use of the source code, but analysis of the compiled assembly instructions can help unfold a similar story. Some fuzzers fall under this category as well.

Chapter 3

Related Work

This research makes use of a random C program generator, in order to generate testsuites on which the performance of fuzzers can be quantified. There is some related work available regarding the generation of C programs needed for this research. Code generators are used for testing compilers [9] as well. Testsuites containing a large number of programs are generated for the testing of a compiler. A lot of inspiration about how to expand on grammar productions in order to generate a program that is random and able to compile came from [10]. This paper provided insights how to generate random C programs using the C grammar.

Currently, there is more research being done to quantify the performance of fuzzers. For example, the same research is being conducted [11] [12] [13], but from a different angle. Namely, taking an existing program, injecting some security vulnerabilities in it, and then running a fuzzer on it. This approach has the advantage of testing real world computer program instead of generated programs, which lack real world functionality. When injecting faults into real world computer programs they get the advantage of realism. However, what they gain in realism, they lack in flexibility. The code generator can be modified to attend to a lot of different needs such as compiler testing or testing the performance of a fuzzer, for example.

The fault injection technique, used by [11] [12] [13], works by injecting a software security threat in the intermediate representation of the source code during the compilation process. The fault injection works by somewhat altering the real intermediate representation to an intermediate representation containing a threat. In contrast with this thesis, where the security threat is placed within the source code and the program is compiled as normal. The fault injection technique has the advantage of being able to research real world code constructs such as memory allocations and recursion. Whereas, for code generation this is very hard to do. This is where the two approaches differ in their ways to quantifying the performance of fuzzers in the detection of security threats.

Chapter 4

Design

We quantify the performance of fuzzers in the detection of security threats by creating a tool which generates simple, pseudo-random C programs, containing a security threat. These vulnerable programs together, form a test suite on which various fuzzers can perform their security vulnerability analysis tests.

Comparing the results of the various fuzzers, given the testsuite of programs consisting of a security threat, gives a metric for determining the performance of various fuzzers. As well as, giving insight in the failings of various fuzzers in the detection of security threats. An overview of the design can be seen in Figure 4.1. The generator generates a C program. Such a program consists of: variables; loops; functions; and, a security threat placed within a number of *if*-statements, the depth. A testsuite to quantify the performance of the fuzzer, consists of multiple programs with the same characteristics. Such as, the same number of functions across all the programs forming the testsuite. Running a fuzzer on a particular testsuite gives insight in what kind of characteristics of a C program will affect the performance of the fuzzer in the detection of security threats.

4.1 Code generation

Requirements

The generated C programs need to satisfy some requirements in order for this research to work. Firstly, the generated C programs need to be able to compile succesfully, in order to obtain an executable on which the fuzzer can run. Secondly, the generated C programs must not contain infinite loops. Otherwise, the fuzzer would be running its tests, but it would be testing nothing, since the program is in a never ending loop and the vulnerability can possibly never be reached. Lastly, the vulnerability must be placed within a piece of code which can be covered during normal execution. For example, the vulnerability must not be placed within an *if*-statement of which the branch can never be taken.

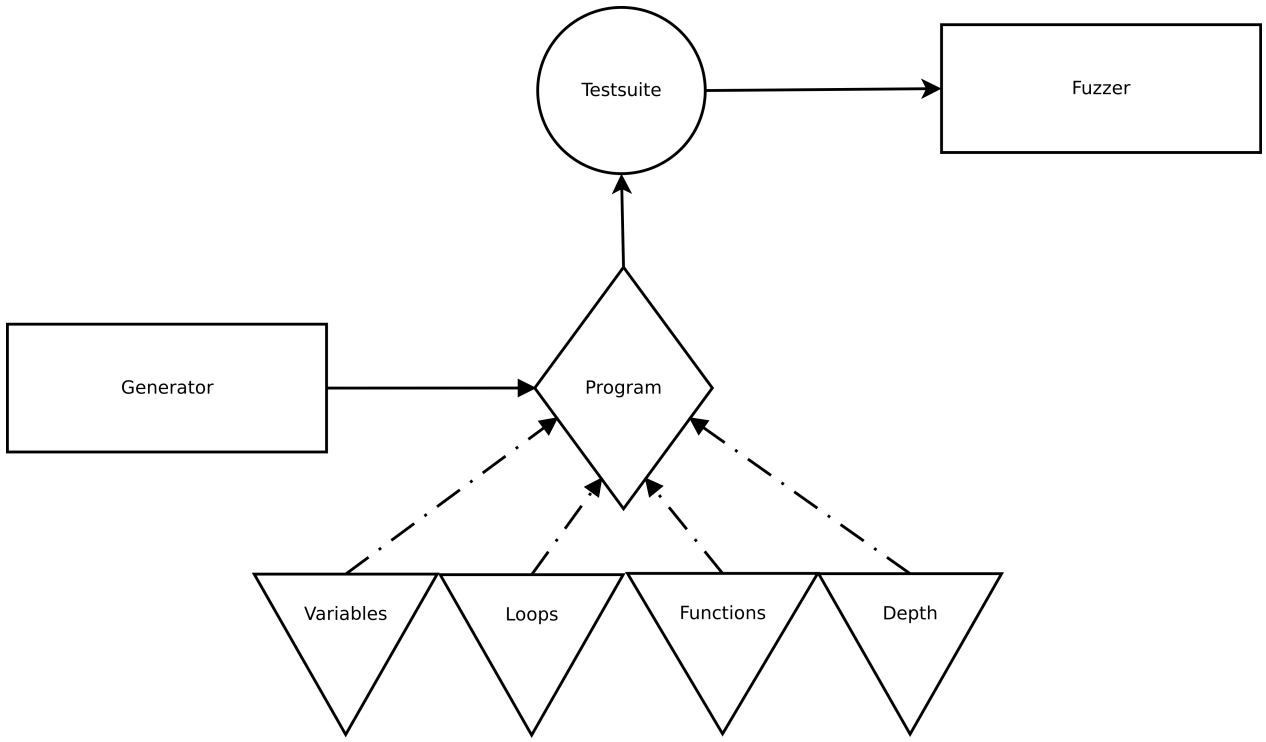

Figure 4.1: An overview of the design.

Constraints

Besides these requirements, there are also some constraints to limit the complexity of the generated programs. These constraints are:

- the only type which is used is *int*;
- a limited number of functions, with at least two and at most four parameters;
- a limited number of loops;
- and, a limited number of variables which are initialized by input from *stdin*.

4.1.1 Stepwise expansion C grammar productions

The first requirement states that it should be able to compile the generated C programs. This requirement is met by generating the programs by making use of stepwise expansion [10] on the C grammar productions [14].

One of the first stages of compiling a C program [15] is checking whether or not the program conforms to the rules set by the C grammar. The compiler does this by checking if it is possible to derive the program using productions from the C grammar. This process is called parsing. Generating programs is like parsing in reverse. Instead of a program going in and checking if it conforms to the rules of the C grammar, a program goes out. This program is generated by expanding on C grammar productions and succesful compilation can be guaranteed, because it has already been derived using productions from the C grammar. However, parsing

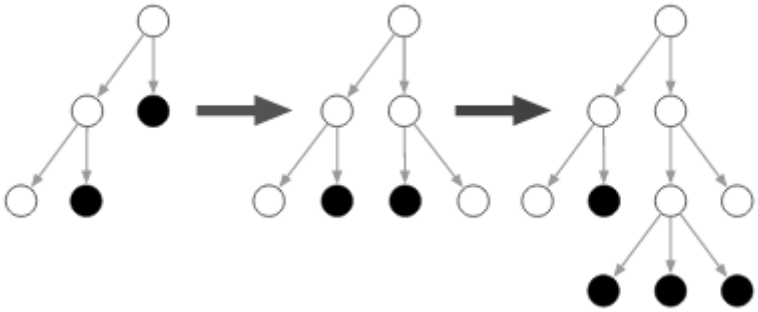

Figure 4.2: An example of a stepwise expansion. Dark nodes are unexpanded nonterminals (can be expanded) while the other nodes have already been expanded before.

is a deterministic process. Thus, generating should be probabilistic, because otherwise the same program is generated everytime. By assigning a weight to each grammar production, it is possible to probabilistically decide on which production to expand. The generation of C programs can be visualised as a random tree walk, as can be seen in Figure 4.2, which came from [10].

Changes to C grammar

In order to meet some of the constraints described above, some changes are made to the C grammar. For example, all of the productions for declaring functions, parameters and variables are omitted, to guarantee there is only a limited number of those. Productions for (de-)referencing pointers are omitted, since these bring an enormous amount of complexity to the generation process. As well as, all division operators, such as /, %, etc. . . , in order to prevent division by zero.

4.1.2 Resolving infinite loops

```
while ( x == x ) {

        .
        .
        .

    /* Code here will endlessly be looped,
            since x == x can never evaluate to false */
}
```

Figure 4.3: An example of an infinite loop.

The second requirement states there are no infinite loops allowed in the code. Infinite loops in the code are eliminated by adding to each loop a maximum upper bound as to ensure termination. An example of an infinite loop can be seen in Figure 4.3, and an example of eliminating the infinite loop with this construct, can be seen in Figure 4.4

```
#define LOOP_BOUND 128

.
.
.

int BND = 0;
while ( BND < LOOP_BOUND && x == x ) {

    /* Code here will loop LOOP_BOUND amount of times,
            since x == x can never evaluate to false,
            and terminate once BND == LOOP_BOUND */

    BND++;
}
```

Figure 4.4: An example of eliminating an infinite loop.

4.1.3 Satisfiability expressions

The third requirement states that it should be possible to reach the vulnerability. The need for satisfiable expressions as conditions for *if* and *while*-statements can be seen in Figure 4.5, whenever the security threat is placed within the branch of the *if*-statement, then there must be a guarantee that there is some value for which the branch can be taken. In other words, the expression in the branch condition must be satisiable, meaning there is at least one value for which the expression evaluates to true. Otherwise the security threat can never be reached during execution. Therefore, the fuzzer cannot reach the security threat, and thus not report it, which will in turn distort the results of the research.

```
if ( x != x) {

        .
        .
        .

    /* Code here can never be reached,
            since x != x can never evaluate to true */
}
```

Figure 4.5: Some source code showing the need for satisfiable expressions.

To guarantee there is at least one possible combination of inputs for the expression within the branch condition such that it evaluates to true, all variables which are normally initialized by user or fuzzer input, are now initialized by some random value, since it could be possible for the user to choose this exact combination of inputs. Whenever an expression needs to be generated for use in a branch or loop condition, then the expression must evaluate to true for these input variables. A new expression is generated, when this is not the case, for as long as the expression keeps evaluating to false.

4.1.4 Functions

The generation of functions is done as follows: generate a name for the function; randomly select a number of parameters to be used between two and four; generate the function body by probabilistically expanding on the *statement-list* production; and, generating a return expression.

Determining function return

A function with no parameters, will always return the same value if there are no local variables initialised with random values. A function with no parameters was deemed useless for this research, because these can be just as well replaced by an integer constant, since it will always return the same value.

Therefore, only functions with parameters are used. However, this does create a problem. Function calls can be part of expressions, and in order to guarantee the satisfiability of expressions, the return value of functions must be known during generation. However, the return value is possibly dependent on the input parameters. In order to determine the function return the function is compiled as a standalone program. This program is then executed in a seperate process with the parameters as input. The function return is then the result of the execution of this program. This does come at a cost, if the function contains a lot of loops, it is possible this will slow down the generation of C programs.

4.1.5 Vulnerability placement

```
int main(void){
    int var0;
    int var1;
    int var2;
    int var3;
    int var4;

    scanf("%d", &var0);
    scanf("%d", &var1);
    scanf("%d", &var2);
    scanf("%d", &var3);
    scanf("%d", &var4);

    if(var0 >= ((var3 * var0 * var2)) * var1){
        if(var3 + var4 & var3 <= (var3 * (var2))){
            if(func1(var4, var4, var4, 176) == 128){
                vuln();
            }
        }
    }
    return EXIT_SUCCESS;
}
```

Figure 4.6: An example of the placing of the vulnerability, within three nested *if*-statments (depth 3).

The vulnerability is always placed in a function called *vuln()*. This function consists of a very simple buffer overflow, like the one in Figure 2.3. The call to the function is placed in the *main*-function of the program, within a number of nested *if*-statements, as can be seen in Figure 4.6. This number of *if*-statements, the depth, can be specified in the generator.

4.1.6 Putting it all together

The following characteristics can be specified in the generator: the number of variables; the maximum number of loops per function; the number of functions; and, the depth of the security threat placement. The generator will produce a satisfiable program, because of the design elements and code constructs described in this chapter. An example of a complete, generated C program can be seen in Figure 4.7.

```c
/*
 * This is a generated C program
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define LOOP_BOUND 64

void vuln(){
    char BUFFER[32];
    scanf("%s", BUFFER);
}

int func0(int par0, int par1, int par2, int par3){
    int BND0 = 0;
    while(BND0 < LOOP_BOUND && (par2 && (par2) <= par0 || 16 & (par2) + par2 | (par0)
        && par0)){
        if(par2 <= (par0 + (par0)) && (par0 * par0) || par3){
            par3 = par3;
        }
        BND0++;
    }
    return par3 || par2;
}

int func1(int par0, int par1, int par2, int par3){
    par0 = par0 * par1;
    return par1;
}

int func2(int par0, int par1){
    par0 = par0;
    return par0 * (64) + par0 & par1 ^ par0 && par1 & par1;
}

int main(int argc, char * argv[]){
    int var0;
    int var1;
    int var2;
    int var3;
    int var4;

    scanf("%d", &var0);
    scanf("%d", &var1);
    scanf("%d", &var2);
    scanf("%d", &var3);
    scanf("%d", &var4);

    if(var0 >= ((var3 * var0 * var2)) * var1){
        if(var3 + var4 & var3 <= (var3 * (var2))){
            if(func1(var4, var4, var4, 176) == 128){
                vuln();
            }
            else{
                var2 = var0;
            }
        }
    }
    else{
        var4 = var4;
    }
    return EXIT_SUCCESS;
}
```


Figure 4.7: An example of a generated program.

Chapter 5

Evaluation

The goal of this research is to quantify the performance of fuzzers in the detection of security threats. Initially, we set out to quantify how various fuzzers would perform given some testsuite. Thus, determining which fuzzer is the best all-round, or on programs containing some particular characteristic. However, fuzzing is a very time intensive process. Therefore, this research shifted towards determining how various code constructs affect the detection rate of a fuzzer.

A number of testsuites were generated for each test performed in this chapter. Each testsuite consists of 25 programs which were fuzzed for 10 and 20 minutes. The machine on which these tests were performed consists of an Intel Core i5-4670K CPU with four cores at 3.40 GHz. Thus, at max load four testsuites can be fuzzed in parallel. Testing four testsuites at a time takes up: $\frac{(10+20)\times 25}{60} = 12.5$ hours. This chapter evaluates mainly the effect of the characteristics of C programs, which can be specified in the generator, on one particular fuzzer, called American Fuzzy Lop [16], or more commonly known as AFL. Due to a time constraint, this makes for a lot of opportunities for future work to expand and improve on.

This fuzzer, which can be seen in Figure 5.1, has an option for grey box and black box fuzzing. Differences between grey box and black box can still be researched as future work, as well as different fuzzers altogether. However, only using one fuzzer offers a clear limitation to this research.

Our hypothesis for this research is: *fuzzers detect on average fewer security threats when fuzzing complex programs*. Since, a complex program consist of more loops and function calls, which extend the execution time of a program, because the same code is repeatedly executed. Therefore, the fuzzer can perform less tests within the same timeframe compared to a less complex program.

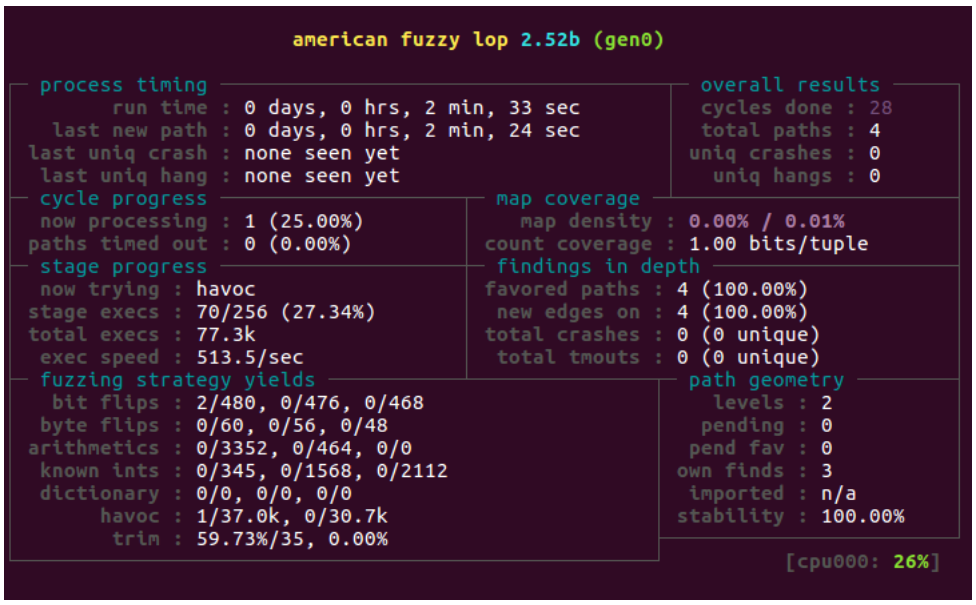

Figure 5.1: An example of a fuzzer at work.

5.1 Variables

The following test was conducted to understand how the number of variables influenced the detection rate. Each program consisted of two functions and a *main*-function. All functions, except *main*, consist of one loop. The *main*-function consists of exactly three function calls. The vulnerability was placed at depth three. For each possible number of variables, ranging from 1; 3; 5; to 7, 25 programs were generated. This makes for 100 testcases which were each fuzzed for a timeperiod of 10 and 20 minutes. The results of this test can be seen in Figure 5.2.

As can be seen in Figure 5.2, there is not a clear relation between the number of input variables a program has and the percentage of security threats detected. The effect of variables on the detection rate of the fuzzer may be so small that this test does not show it. In order to conclude no relation between the number of variables and the detection rate, this test needs to be redone with a greater number of variables.

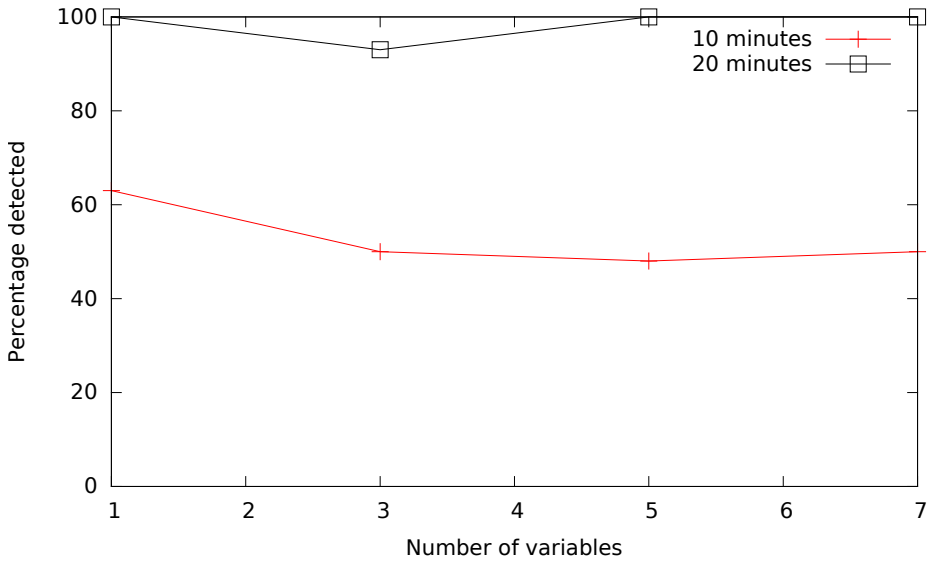

Figure 5.2: The effect of variables on the detection of security threats.

5.2 Loops

In order to understand how the number of loops per function influenced the detection rate, the following test was conducted. Five variables, two functions, three function calls and three nested *if*-statements were placed in each program. For each possible number of loops per function, ranging from 1 to 4, 25 programs were generated. This makes for 100 testcases which were each fuzzed for a timeperiod of 10 and 20 minutes. The results of this test can be seen in Figure 5.3.

As can be seen in Figure 5.3, there is not a clear relation between the number of loops per function and the percentage of security threats detected. However, a relation was expected by our hypothesis.

Inspection of the number of executions, as can be seen in Figure 5.4, confirms our hypothesis. However, the number of loops does not have such a big effect on the average number of executions as was expected. This can be possibly explained, because satisfiability of the loop condition is guaranteed, this does not necessarily mean the fuzzer guesses the correct combination of input variables in order to evaluate the loop condition to true. Meaning, the loop is possibly not being executed that much by the fuzzer. This has an even bigger effect on nested loops, since the inner loop does not get reached until the outer loop condition evaluates to true.

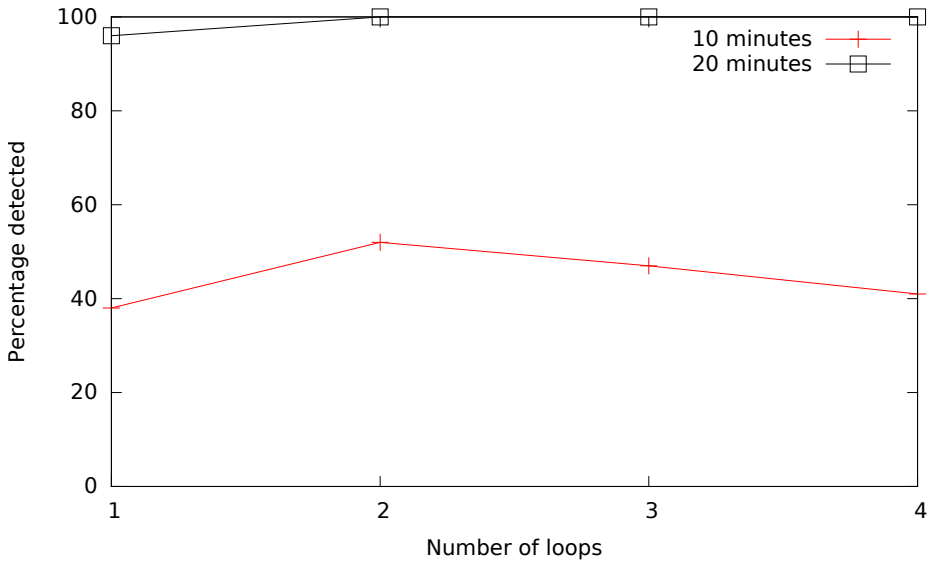

Figure 5.3: The effect of loops on the detection of security threats.

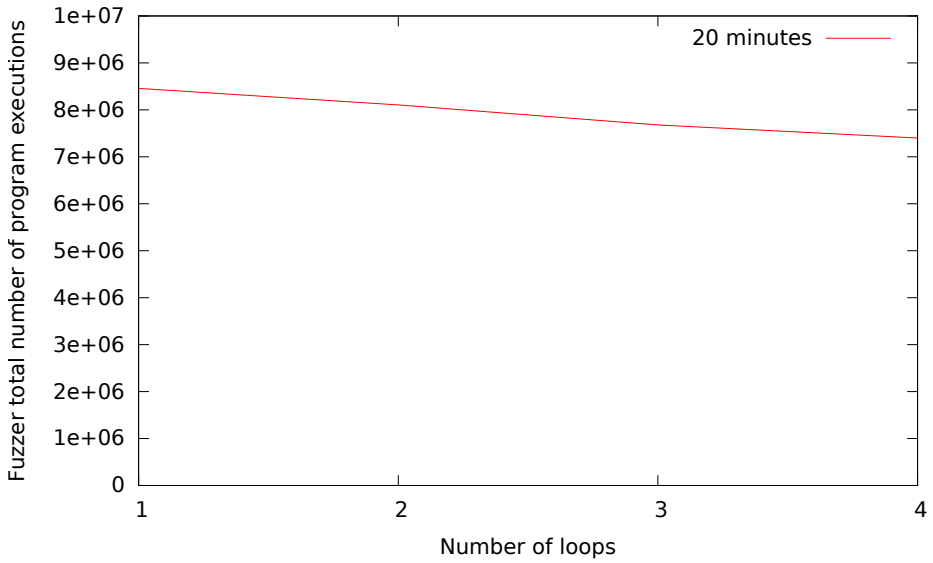

Figure 5.4: The effect of loops on the number of executions performed by the fuzzer.

5.3 Function calls

To understand how the number of function calls influenced the detection rate, the following test was conducted. Five variables, one loop per function, two functions and three nested *if*-statements were placed in each program. For each possible number of function calls, ranging from 1 to 4, 25 programs were generated. This makes for 100 testcases which were each fuzzed for a timeperiod of 10 and 20 minutes. The results of this test can be seen in Figure 5.5.

Again, as with the loops, there is not a clear relation between the number of function calls and the detection rate of the security threat. Inspection of the number of executions, as can be seen in Figure 5.6, shows the

number of function calls does not have such a big effect on the number of executions as was expected. Probably, because the loop in the function does not perform that much iterations in the tests by the fuzzer. In order to conclude no relation between the number of function calls and the detection rate, this test needs to be redone with: a larger number of function calls; a larger number of loops per function; and, a bigger LOOP_BOUND, such that the loops have a much bigger upper bound.

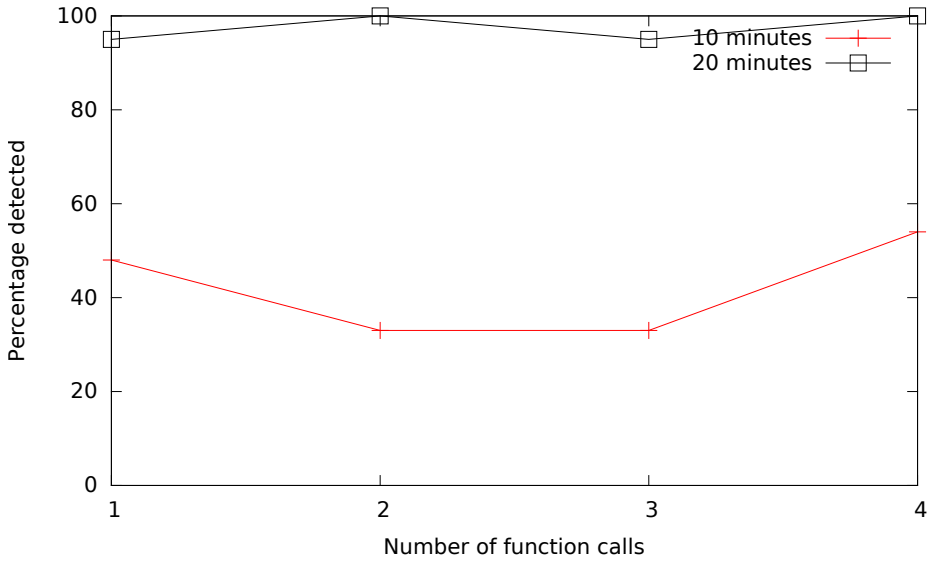

Figure 5.5: The effect of function calls on the detection of security threats.

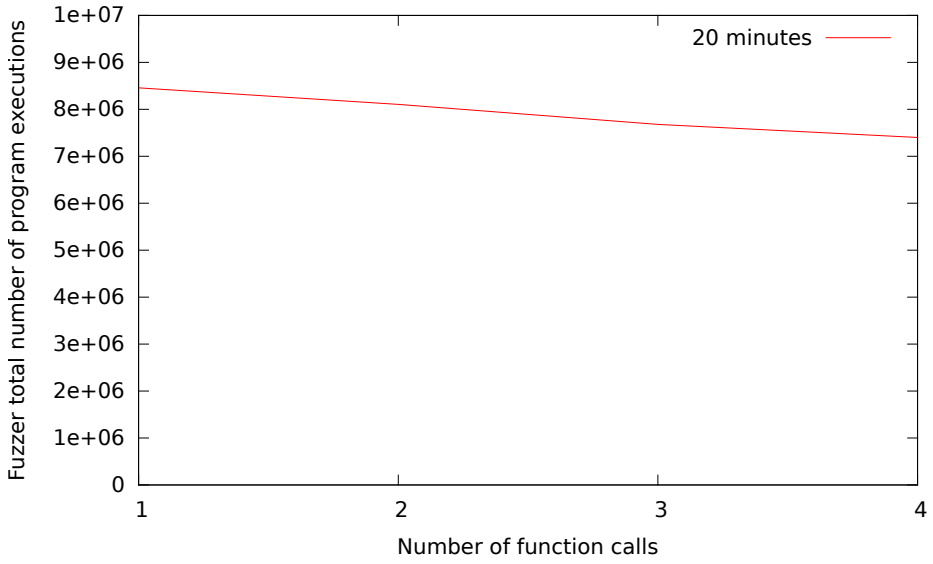

Figure 5.6: The effect of function calls on the number of executions performed by the fuzzer.

5.4 Depth

To understand how the depth of the vulnerability placement influenced the detection rate, the following test was conducted. Five variables, one loop per function, two functions, three function calls and a number of nested *if*-statements, ranging from 1 to 4, were placed in each program. For each possible number of *if*-statements, 25 programs were generated. This makes for 100 testcases which were each fuzzed for a timeperiod of 10 and 20 minutes. The results of this test can be seen in Figure 5.7.

As can be seen in Figure 5.7, there is a clear relation between the depth of the vulnerability placement and the percentage of security threats detected, at 10 minutes of fuzzing. The deeper the placement, the less security threats detected, since all the expressions in the *if*-statements need to evaluate to true in order to reach the security threat. There is not a clear relation at 20 minutes of fuzzing. Probably because again, the programs were not complex enough.

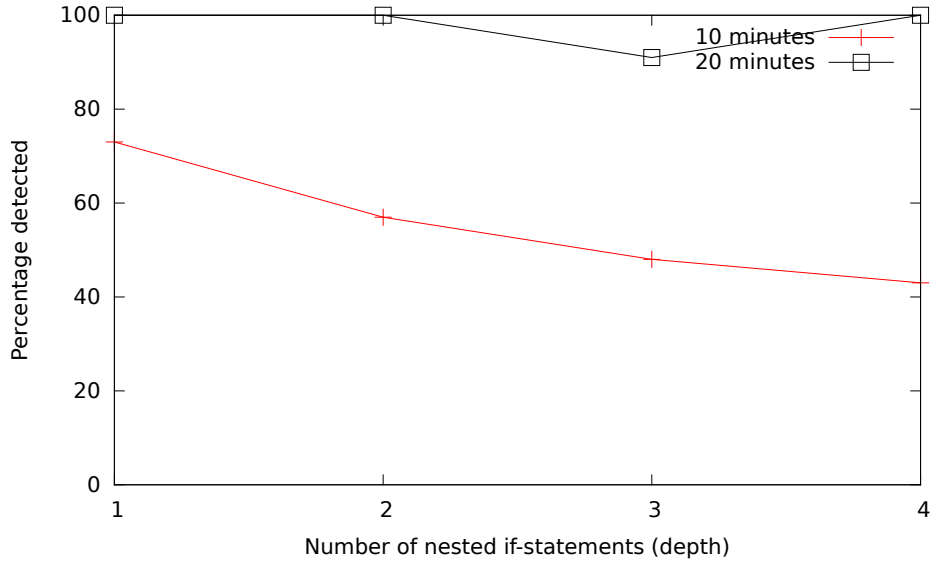

Figure 5.7: The effect of the placement of the vulnerability (depth) on the detection of security threats.

5.5 Concluding remarks

The tests performed provide insight to what extent time influences the detection rate during fuzzing. Logically, the more time spent on fuzzing the higher the percentage of security threats were detected, since the fuzzer can execute more tests. This chapter has shown that code constructs which take longer to execute such as loops and functions (specifically calls to the function), do not necessarily influence the detection rate of security threat that much. The depth of the placement of the security threat does have an effect on the detection of security threats, since the fuzzer must execute a greater number of tests before finding the right combination of inputs.

However, the programs which are generated are not necessarily realistic programs. The programs can be

compiled and executed, but they lack real functionality. This might offer a limitation to this research, since real world software makes use of a much bigger number of variables, loops (more importantly loops with loop conditions which probably are true more often, altough this cannot be said for sure) and functions. As well as, more complex code constructs such as pointers and memory allocations. This is probably the biggest limitation to this research, since sloppy memory operations in C are one of the biggest security threats.

Chapter 6

Conclusions

This thesis is about quantifying the performance of a fuzzer in the detection of security threats. As well as, how various factors, like the number of functions or the amount of time set for the fuzzer, influence the detection rate of these security threats. This thesis has shown that a fuzzer has a greater probability of not finding the security threat when it is placed within a greater number of nested *if*-statements.

In order to conduct this research, a random C program generator was written. This generator can generate guaranteed satisfiable programs containing exactly one security threat. Characteristics, such as the number of functions, can be set in the generator. This generator was used to generate testsuites which were used to quantify the performance of fuzzers.

A lot of research can still be done regarding this subject. It is still possible to quantify the performance of various fuzzers on the same testsuite of generated programs in order to determine which fuzzer performs the best.

There is also a lot of research possible regarding the generation of programs. Since, the generator used for this thesis omitted a lot of real world code constructs such as pointers and memory allocations. Being able to generate these kind of programs is difficult, but would provide a much more realistic test suite on which more experiments can be performed.

Bibliography

[1] LINFO. Bug definition. *The Linux Information Project*, 2005.

[2] Krzysztof R. Apt, Frank de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[3] Ari Takanen, Jared DeMott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance.* Artech House, Inc, 2008.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1998.

[5] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4), 2004.

[6] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[7] Chris Anley, Jack Koziol, Felix Linder, and Gerardo Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, Inc., 2007.

[8] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[9] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[10] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[11] Wampie Driessen. Benchmarking a fuzzer. *Leiden Universiteit, LIACS*, 2018.

[12] Vincent van Rijn. Quantifying to what extent software vulnerabilities can be detected using fuzzers. *Leiden Universiteit, LIACS*, 2018.

[13] Vincent den Hamer. Effectiveness of fuzzers in detecting memory errors. *Leiden Universiteit, LIACS*, 2018.

[14] ISO. Iso 9899:201x — programming languages — c. Standard, International Organization for Standardization, Geneva, CH, April 2011.

[15] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[16] Michael Zalewski. American fuzzy lop. *http://lcamtuf.coredump.cx/afl/*, 2017.