



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Impact factors for severity assessment of
bugs discovered via compositional
symbolic execution**

Ricardo Nales Amato





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Impact factors for severity assessment of
bugs discovered via compositional
symbolic execution**

**Einflussfaktoren zur Bewertung der
Severity von Programmfehlern, welche mit
Compositional Symbolic Execution
entdeckt wurden**

Author:	Ricardo Nales Amato
Supervisor:	Saahil Ognawala
Advisor:	Profr. Dr. Alexander Pretschner
Submission Date:	23.10.2017

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 23.10.2017

Ricardo Nales Amato

Acknowledgments

To my supervisor, for always guiding me, and pointing me in the right direction. To my friends, for always helping, believing and encouraging me. And to my family, without them none of this would have been possible. My most heartfelt thanks to all of you.

Abstract

Software projects are big undertakements, usually developed by several teams. All of them will be supporting the same codebase, which will grow exponentially week by week, and surely gathering a good amount of bugs along the way. For this matter, testing has been widely adopted to find bugs and resolve them. But the question that arises when a bug is found more times than not is: how do we know which bug(s) we should prioritize? how can we measure the impact of a given bug? how much resources are required to solve this bug?

In this Master Thesis we set out to find if there was a correlation between the graph features of a program's callgraph, and its vulnerabilities found through a compositional symbolic execution, using MACKE. The graph features are to be generated by an internally developed tool, which in turn would allow us to automate the process of assessing the vulnerability.

We then follow to test our automatic assessment model with real professional feedback by using a GUI with which users can interact and see the results for themselves.

Contents

Acknowledgments	iii
Abstract	iv
1 Introduction	1
2 Motivation	4
2.1 The severity of vulnerabilities	4
2.1.1 How are vulnerabilities found	4
2.1.2 The severity of a vulnerability	5
2.1.3 Common Vulnerability Scoring System	5
2.2 Symbolic execution	5
2.2.1 Modular and Compositional Analysis with KLEE Engine	6
2.2.2 Callgraphs	6
2.3 Putting it all together	7
3 Methodology and implementation	8
3.1 Research phase	8
3.1.1 Understanding LLVM	8
3.1.2 Getting to know KLEE	9
3.1.3 Compositional symbolic execution through MACKE	10
3.1.4 Call graphs	12
3.1.5 Directed graphs and Node Attributes	14
3.2 Node attributes found through MACKE	17
3.2.1 Exploring CVSS base scores, and their meaning	17
3.3 Implementation phase	21
3.3.1 Finding a reliable Vulnerabilities Database and suitable programs to analyze	21
3.3.2 Running all programs through MACKE	25
3.3.3 Analisis of the vulnerabilities found	26
3.3.4 Calculating node attributes	28
3.3.5 Correlation found	29
3.3.6 Learning CVSS base score values from node attributes	30

3.3.7	Framework implementation	31
4	Evaluation	33
4.1	Experiments	34
4.1.1	Docker container setup	34
4.1.2	Pre-processing the data	34
4.1.3	CVSS score assignment	34
4.1.4	Runnig MACKE on programs	34
4.1.5	Getting Call Graphs and Node Attributes	34
4.1.6	Learning Setup	34
4.1.7	Front-end implementation	34
4.1.8	Survey	34
4.1.9	Including Feedback	34
4.2	Results	34
4.2.1	CVSS Scores found	34
4.2.2	MACKE vulnerabilities found	34
4.2.3	Learning results	34
4.2.4	Survey Results	34
4.2.5	Future work	34
5	Related Work	35
5.1	MACKE's ranking of vulnerabilities	35
5.1.1	MACKE's severity assessment	36
5.2	A Novel Automatic Severity Vulnerability Assessment Framework . . .	39
5.2.1	How to improve upon current QVAS?	40
5.2.2	Pipeline of the ASVA framework	40
5.2.3	Is ASVA an improvement over our current framework	42
6	Conclusion	43
	List of Figures	44
	List of Tables	45
	Bibliography	46

1 Introduction

In today's world, where software applications are ever growing, with features being added at a much quicker pace as before has lead to an overflow of vulnerabilities, that also grow ever so complex. These vulnerabilities need to be taken care of, but what if our resources are scarce? if we cannot fix every single vulnerability? how do we know objectively which one should take priority over the rest?

This can be done manually of course if the developers have knowledge in security and how a specific vulnerability could be exploited, and while this works, it also could fall into the realm of the subjectivity. Fortunately, there is a standard that is being adopted rapidly by the IT industry called CVSS or Common Vulnerability Scoring System [7] which allows us to rate the severity of a vulnerability by extracting some of the properties of the vulnerabilities. This rating is a numerical value from 0 to 10, which could allow us to finally know which vulnerabilities should be prioritized.

The CVSS has been developed by the National Infrastructure Advisory Council (NIAC) in 2003/2004. It's first iteration showed promise in objectively assessing a vulnerability, and gained momentum across the IT industry with its second iteration. In late 2015 CVSS 3.0 – or CVSS3 – was released, and has now become a staple in security, when it comes to assessing how severe a vulnerability is, across projects. This in turn has also led to all obscure assessments done by development teams, which were for the most part subjective, to banish in favor of using a standardized metric.

In order to get the CVSS3 score of a vulnerability we must know certain attributes from it, which CVSS3 refers to as "Base Scores"[7]. These scores are:

- Attack Vector (AV) - This metric reflects the context by which vulnerability exploitation is possible. The Base Score increases the more remote (logically, and physically) an attacker can be in order to exploit the vulnerable component.
- Attack Complexity (AC) - This metric describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Such conditions may require the collection of more information about the target, the presence of certain system configuration settings, or computational exceptions.

- Privileges-Required (PR) - This metric describes the level of privileges an attacker must possess before successfully exploiting the vulnerability. This Base Score increases as fewer privileges are required.
- User Interaction(UI) - This metric captures the requirement for a user, other than the attacker, to participate in the successful compromise the vulnerable component. This metric determines whether the vulnerability can be exploited solely at the will of the attacker, or whether a separate user (or user-initiated process) must participate in some manner. The Base Score is highest when no user interaction is required.
- Scope (S) - Does a successful attack impact a component other than the vulnerable component? If so, the Base Score increases and the Confidentiality, Integrity and Authentication metrics should be scored relative to the impacted component.
- Confidentiality (C) - This metric measures the impact to the confidentiality of the information resources managed by a software component due to a successfully exploited vulnerability. Confidentiality refers to limiting information access and disclosure to only authorized users, as well as preventing access by, or disclosure to, unauthorized ones.
- Integrity (I) - This metric measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of information.
- Availability (A) - This metric measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. It refers to the loss of availability of the impacted component itself, such as a networked service (e.g., web, database, email). Since availability refers to the accessibility of information resources, attacks that consume network bandwidth, processor cycles, or disk space all impact the availability of an impacted component.

By finding these values, one can obtain the aforementioned value between 0 and 10, the former being vulnerability that requires no attention and the latter one that needs immediate attention.

Finding said vulnerabilities can prove itself difficult, and sometimes costly if it is not fixed before it gets in the hands of externals. Fortunately as well, there are tools that allow us to find vulnerabilities in an automatic fashion, with very high source code coverage by using compositional symbolic execution, such as MACKE [15]. The problem with this tool is that, though it can find vulnerabilities, we are left with the initially presented issue, how do we know how severe each vulnerability found is? would fixing it make sense, and be worth the investment?

The main drive of this thesis emerged from placing those two scenarios together. What if we could use compositional symbolic execution, and get vulnerabilities automatically, that have already been assessed following the CVSS3 standard? This would not only allow teams to find very obscure errors that might be hard to find during Q&A but also know if they need immediate attention or not.

To do this, we would need something that can be analyzed and matched against each of the aforementioned base scores from CVSS3. Luckily, a static call graph can be extracted from all programs that can be run by MACKE, which can be completely analyzed through the use of graph logic [1]. The call graph extracted would be a directed graph, that is nothing more than a collection of nodes and edges, the latter having a source and a target. Their relations have specific attributes and can be thoroughly analyzed and quantified by using graph theory.

To do all of this we would need a reliable point of reference, in our case it being vulnerabilities that have already been assessed with CVSS3 and their source code. The code will be passed through MACKE, which in turn will find vulnerabilities through compositional symbolic execution. If it is able to find the same vulnerability documented in the database, we would then be in good standing to analyze by taking a look at the attributes of the call graph.

With the attributes calculated from the vulnerable nodes, and the CVSS3 scores from the database – which have been manually calculated – we would be able to correlate them. Furthermore, through the use of machine learning, we expect that if we use the attributes found and the CVSS3 scores for learning, we would be able to get a strong model that could predict the CVSS3 base scores.

Our hypothesis is that by developing a framework, which uses all of the aforementioned procedures, empowers developers and testers, so that they can find vulnerabilities, assess them and correct them as fast as possible, reducing the investment in both testing – using symbolic execution – and assessment – using our framework – leading to a solution to the initially proposed problem.

2 Motivation

In today's world, all software that has been developed by a team of programmers is bound to have errors, and the amount of them you might have in a project will only keep incrementing if they are not dealt with in a timely manner. Of all these errors that are found, they usually lead to vulnerabilities in our system, which can be exploited in several ways – ways which will be discussed at a later point in this thesis – and sometimes the time to fix one vulnerability completely outweighs the benefit provided. That is why knowing the severity of a given vulnerability is usually crucial to assess correctly the amount of resources we need to assign for its correction.

2.1 The severity of vulnerabilities

Since the inception of informatics, errors in code, be it semantic or syntactic, have existed. These errors, as mentioned previously, usually lead to security issues in our application, called software vulnerabilities.

Most errors arise from simple programming mistakes, and vary from one programming language to another, as well as the field where these programs are being put to use. Regardless of how a vulnerability comes to be, a solution needs to be found for it, which usually includes its development, testing, and a deployment.

But in projects where there are more vulnerabilities than resources to assign to them, or when fixing one vulnerability would yield a benefit that is simply not worth the investment? What if there was a way to include a numerical approximation that assesses its severity? We go into detail throughout this chapter, covering all of the reasons that lead us to undertake this project.

2.1.1 How are vulnerabilities found

Vulnerabilities can be found in several ways, and all of them have different characteristics,

A vulnerability can be found in a plethora of ways: it can be a simple semicolon missing from the code, it could be a typo in one variable, these are now-a-days easy to find since there are IDEs (Integrated Development Environment); or it could be that someone overlooked sanitizing an input, that could lead to buffer-overflow errors

static
analysis,
dynamic
and sym-
bolic

during runtime. Whichever the source of a given vulnerability, the approach to find it is always the same: reproduce this error by supplying the inputs that got the software to an unstable state, or in most cases, crashed it.

2.1.2 The severity of a vulnerability

A vulnerability has the power to bring a whole system down. They can be a force to be reckoned with, when it has spawned inside a codebase that has built up to millions of lines of code. After a vulnerability has been found, we must know how severe its impact on the system will be, and act accordingly. If a severity has the power of bringing the system down, we know it is quite severe.

But what happens in the normal occasion when a vulnerability is found and there is no way to know how severe it is at first glance? when there are several vulnerabilities that subjectively speaking, all seem to affect the system equally? When resources are scarce and we need to prioritize which vulnerability to focus on next so that our system may continue working as intended? When there are extremely huge time constraints and we can only budget to fix one at a time, because our workforce is limited?

2.1.3 Common Vulnerability Scoring System

The Research by the National Infrastructure Advisory Council (NIAC) in 2003/2004 led to the launch of CVSS version 1 in February 2005, with the goal of being "designed to provide open and universally standard severity ratings of software vulnerabilities". This initial draft had not been subject to peer review or review by other organizations. In April 2005, NIAC selected the Forum of Incident Response and Security Teams (FIRST) to become the custodian of CVSS for future development.

FIRST.Org, Inc. (FIRST), a US-based non-profit organization, whose mission is to help computer security incident response teams across the world[7] asked themselves these same questions back in 2005, when they decided to introduce CVSS, or Common Vulnerability Scoring System[7].

Throughout this thesis we introduce ways to find the severity of a specific vulnerability. Depending on several values, and using the aforementioned metric CVSS3 as our basis to rate them, we will give a vulnerability a numeric score, which could allow us to know what vulnerability should be given priority, and which one might be harmless.

2.2 Symbolic execution

Symbolic execution is a way of testing a program to determine what inputs allow the software to execute. Instead of using normal variables for testing, an interpreter

follows the execution assuming symbolic values as the inputs instead of using concrete values – as per in normal execution – which can take any value for the given type of the input, thus covering the complete range of possible branches the program could take, therefore discovering vulnerabilities along its way.

2.2.1 Modular and Compositional Analysis with KLEE Engine

Throughout this thesis MACKE (Modular and Compositional Analysis with KLEE Engine)[15] will be used to find vulnerabilities, and callgraphs of programs to fully analyze the source of the vulnerability, and the possible severity of it.

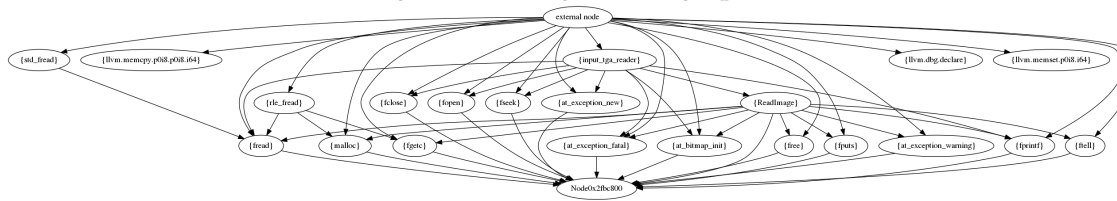
This tool makes use of symbolic execution techniques at the level of C functions, and then combines the results using static code information and inter-procedural path feasibility[15].

The use of this tool allows us to find an extensive amount of undocumented and documented bugs. The documented bugs found will be analyzed in the initial phases, since they have already been assessed manually with the CVSS3 framework.

2.2.2 Callgraphs

A callgraph[16] is the relationship between functions in a computer program, where several functions are present. It is, for the most part, user readable, and allows for an overall view of a program. It helps to pinpoint functions that might be unused, and if there are a few functions that are too intertwined. In general, a callgraph allows users to find if the intended purpose of the program has been met, and if the means to achieve the result was correct.

Figure 2.1: Program callgraph



Call graphs can be dynamic or static. A dynamic call graph is a record of an execution of the program, for example as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is an undecidable problem, so static call graph algorithms are generally overapproximations.

That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

Call graphs can be defined to represent varying degrees of precision. A more precise call graph more precisely approximates the behavior of the real program, at the cost of taking longer to compute and more memory to store. The most precise call graph is fully context-sensitive, which means that for each procedure, the graph contains a separate node for each call stack that procedure can be activated with. A fully context-sensitive call graph is called a calling context tree. This can be computed dynamically easily, although it may take up a large amount of memory. Calling context trees are usually not computed statically, because it would take too long for a large program. The least precise call graph is context-insensitive, which means that there is only one node for each procedure.

2.3 Putting it all together

By using compositional symbolic execution we would be able to find some vulnerabilities – as many as MACKE can find in its scope of testing – and also generate the programs callgraph. By having these two components, and by measuring its vulnerability – by following the CVSS3 guidelines – we could correlate the graph attributes to each and every one of the base scores, to see if there is a pattern, which could later then be used to generate a supervised learning algorithm that could automatically give us the CVSS3 scores of bugs found by MACKE.

The main drive behind doing this is the belief that the relations among functions, and their location in the graph is deeply intertwined with the severity of a vulnerability. A callgraph can tell us if the program is accessible directly, or not, for instance. It can also tell us if a vulnerability will propagate or not, allowing us to pinpoint the starting and end points of a potential chain of errors.

3 Methodology and implementation

In order to find the CVSS3 score of a set of bugs found through compositional symbolic execution, we needed to go through a set of very rigorous steps, that had to be sequential since all of them relied on the one that came just before it.

3.1 Research phase

With our motivation in mind, and a specific goal set, we set out to research all the components that would be needed. All successes and failures will be noted, so that future work on this area may use this work to avoid some pitfalls.

It is also worth noting that we will touch on all the main technologies involved in the entirety of the process, though not going to specifics when the involvement of the technology does not require much user intervention, such as in the case of LLVM for instance.

3.1.1 Understanding LLVM

Analysing source code can prove difficult, specially programs that need to be compiled, such being the case of our thesis, where we will be working with C and C++ source code. The reason why it could prove difficult is that the relationships between functions, and classes – in the case of C++ – can be lost when for example, cleansing the data for text mining. It could also be possible to analyze binaries, but this would prove difficult since it would only allow us to see the results of the compilation, and again could lead to losing some accuracy and important data during compilation.

Symbolic execution tools (e.g KLEE, or MACKE) require the input to be bitcode, specifically compiled with LLVM. The bitcode is an intermediate point where it can still be altered with, to some extent (since the source code is completely tokenized), while keeping its structure.

The LLVM compiler infrastructure project (formerly Low Level Virtual Machine) is a "collection of modular and reusable compiler and toolchain technologies" used to develop compiler front ends and back ends.

LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages.

Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include ActionScript, Ada, C Common Lisp, Crystal, D, Delphi, Fortran, OpenGL Shading Language, Halide, Haskell, Java bytecode, Julia, Lua, Objective-C, Pony, Python, R, Ruby, Rust, CUDA, Scala, Swift, and Xojo.

The LLVM project started in 2000 at the University of Illinois at Urbana–Champaign, under the direction of Vikram Adve and Chris Lattner. LLVM was originally developed as a research infrastructure to investigate dynamic compilation techniques for static and dynamic programming languages.

The name LLVM was originally an initialism for Low Level Virtual Machine, but this became increasingly less apt as LLVM became an "umbrella project" that included a variety of other compiler and low-level tool technologies, so the project abandoned the initialism. Now, LLVM is a brand that applies to the LLVM umbrella project, the LLVM intermediate representation (IR), the LLVM debugger, the LLVM C++ Standard Library (with full support of C++11 and C++14), etc. LLVM is administered by the LLVM Foundation. [10]

3.1.2 Getting to know KLEE

The program now known as KLEE[5] stemmed from EXE [2] which was a program that allowed for symbolic execution, or code that can generate its own test cases as they mention. To better exemplify how EXE worked, which is still the core of KLEE, we have the following code:

It shows how different errors may come to be, and some of them could lead to an exploitable vulnerability. This code also has several branches that can be taken, which are shown in the following figure:

And as shown, EXE had as one of its limitations, that it could only deal with branches where numeric values were used, as it happens in the example showcased in Figure 2.2 . KLEE has built upon this work, and now is what their developers call

`"A hybrid between an operating system for symbolic processes and an interpreter. Each symbolic process has a register file, stack, heap, program counter, and path condition" [5]`

All programs must be compiled to the LLVM assembly language. KLEE directly interprets this instruction set, and maps instructions to constraints without approximation, since it uses bit-level accuracy.[5]

KLEE requires no modification of the source file, only that it has been compiled with LLVM, and run the bitcode directly through KLEE. What KLEE will do, is the same thing that EXE previously did, which is add symbolic parts in the code – when possible –, but introducing all the aforementioned benefits.

Figure 3.1: C program analyzed by EXE

```

1 : #include <assert.h>
2 : int main(void) {
3 :     unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :     make_symbolic(&i);
5 :     if(i >= 4)
6 :         exit(0);
7 :     // cast + symbolic offset + symbolic mutation
8 :     char *p = (char *)a + i * 4;
9 :     *p = *p - 1; // Just modifies one byte!
10:
11:     // ERROR: EXE catches potential overflow i=2
12:     t = a[*p];
13:     // At this point i != 2.
14:
15:     // ERROR: EXE catches div by 0 when i = 0.
16:     t = t / a[i];
17:     // At this point: i != 0 && i != 2.
18:
19:     // EXE determines that neither assert fires.
20:     if(t == 2)
21:         assert(i == 1);
22:     else
23:         assert(i == 3);
24: }
```

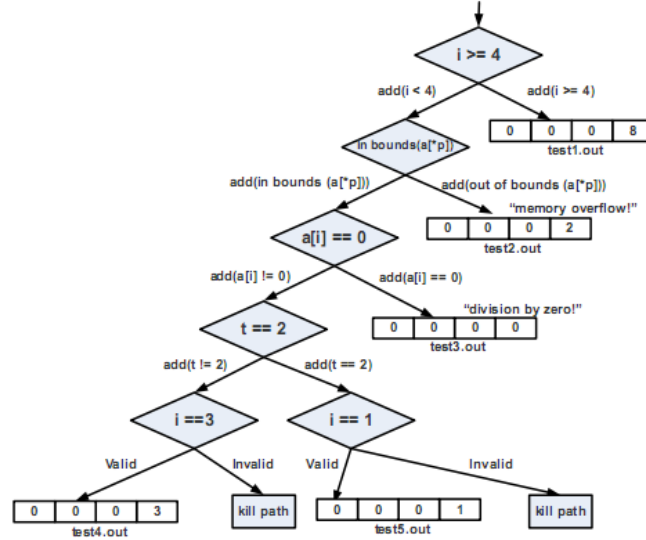
KLEE’s results are quite thorough, though for our research we only need the main JSON file that it generates with: functions where vulnerabilities have been found, and their respective JSON file that contains all the details of how this vulnerability came to be / was found.

The problem that KLEE has is that generates test cases by starting at the entry point of the program (forward symbolic execution), resulting in insufficient code coverage, which will leave many vulnerabilities unfound. Though there is a tool that allows for compositional symbolic execution, which allows for a much better code coverage, thus granting us a much higher chance of finding vulnerabilities across a given program. Its name is MACKE (Compositional Analysis of Low-Level Vulnerabilities with Symbolic Execution). [15]

3.1.3 Compositional symbolic execution through MACKE

The program MACKE, was designed to solve the issue that forward symbolic execution had, by using a three-step approach. Its developer describes the tool’s approach in the following manner:

Figure 3.2: EXE symbolic execution



"MACKE is based on the modular interactions inferred by static code analysis, which is combined with symbolic execution and directed inter-procedural path exploration. This provides an advantage in terms of statement coverage and ability to uncover more vulnerabilities."[15]

How MACKE goes about doing this is by splitting what would normally just be a forward symbolic execution of a program, like KLEE would – from which MACKE originated – into three steps.

"Firstly, MACKE performs symbolic execution on the individual components of a program, in isolation. This has the advantage of higher code coverage and ability to uncover many lowlevel vulnerabilities in all program components. Secondly, MACKE uses results of the first step to reason about (and, therefore, reduce the number of) reported vulnerabilities from a compositional perspective, i.e. by finding feasible inter-procedural paths for those vulnerabilities to be exploited. Thirdly, MACKE assigns severity scores to reported vulnerabilities by considering several characteristic features and provides the result in an interactive visual format."[15]

What this provides is a much higher chance of finding vulnerabilities in the code, because of a much more fine grain code coverage.

The usage of MACKE is quite similar to that of KLEE; the program must be compiled with LLVM, so that a bitcode file is available, which is the used as the input of MACKE.

There is no need to modify the source code in order for it to work.

MACKE's results are very robust, and showcases the three-step execution it goes through. The main resulting file is "klee.json" which includes the function names of only those where vulnerabilities were found. It shows whether the vulnerability was found in phase 1, which means if the vulnerability was found when each of the components of the program were isolated and executed. If it shows phase 2, it means that the vulnerability was found at a lower level, and found by using the results of phase 1.

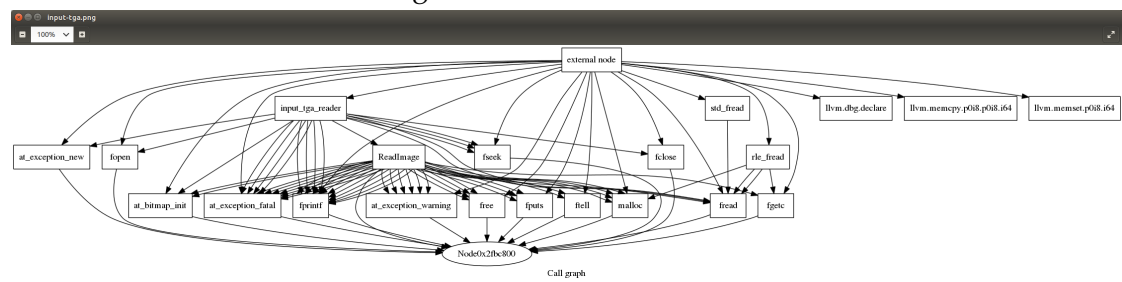
3.1.4 Call graphs

By compiling our programs with LLVM we are able to get a bitcode file, which includes all interactions between the program's components (i.e functions, interfaces..). This file will also allow us to generate a call graph of such interactions.

A call graph is a flow graph, that showcases in a form of a set of nodes connected to each other, how all functions from a given program interact between each other, in other words, their relationships. Callgraphs are directed graphs, which entails that all nodes will have edges (f,g) indicating that function f calls function g. For instance if a node X has an edge with (X,X) it means that it is calling itself, making it a recursive function.

The purpose of call graphs is to display in a human readable way, how a program interacts internally, which can allow programmers to quickly pinpoint where a bottleneck, or security flaw may be. It also serves as a starting point when analyzing the design of an application. If there are disconnected nodes, or redundant nodes, it is quickly visible at first glance, so that proper action can be taken.

Figure 3.3: AutoTrace 0.31.1



There are two types of callgraphs. The first one is dynamic, which displays how the relationships during runtime look. This callgraph can vary depending on what commands were given to a program upon its start-up. The second one is static, and

as the name implies, it never changes. A static call graph shows every single call, regardless of whether it may or not be called during the program's runtime.[8]

In this case we extract a static callgraph, as seen in Figure 2.3 it is clearly visible how a static callgraph overapproximates the amount of calls of a given function to another. Using "ReadImage" as an example, it calls "fprintf" several times in this callgraph. The code looks as follows:

```
static at_bitmap_type ReadImage (FILE *fp,
struct tga_header *hdr,
at_exception_type * exp);
at_bitmap_type
input_tga_reader (at_string filename,
at_input_opts_type * opts,
at_msg_func msg_func,
at_address msg_data)
{
    FILE *fp;
    struct tga_header hdr;

    at_bitmap_type image = at_bitmap_init(0, 0, 0, 1);
    at_exception_type exp = at_exception_new(msg_func, msg_data);

    fp = fopen (filename, "rb");
    if (!fp)
    {
        LOG1 ("TGA: can't open \"%s\"\n", filename);
        at_exception_fatal(&exp, "Cannot open input tga file");
    }

    /* Check the footer. */
    if (fseek (fp, 0L - (sizeof (tga_footer)), SEEK_END)
    || fread (&tga_footer, sizeof (tga_footer), 1, fp) != 1)
    {
        LOG1 ("TGA: Cannot read footer from \"%s\"\n", filename);
        at_exception_fatal(&exp, "TGA: Cannot read footer");
        goto cleanup;
    }

    /* Check the signature. */
```

```

if (fseek (fp, 0, SEEK_SET) ||
fread (&hdr, sizeof (hdr), 1, fp) != 1)
{
    LOG1 ("TGA: Cannot read header from \"%s\"\\n", filename);
    at_exception_fatal(&exp, "TGA: Cannot read header");
    goto cleanup;
}

/* Skip the image ID field. */
if (hdr.idLength && fseek (fp, hdr.idLength, SEEK_CUR))
{
    LOG1 ("TGA: Cannot skip ID field in \"%s\"\\n", filename);
    at_exception_fatal(&exp, "TGA: Cannot skip ID field");
    goto cleanup;
}

image = ReadImage (fp, &hdr, &exp);
cleanup:
fclose (fp);
return image;
}

```

It is not so clear to see, but the C library function `int fprintf(FILE *stream, const char *format, ...)` sends formatted output to a stream. It is being used on every line where a logentry is made. It clearly shows how only one of these log entries will be made per function call, and not all of them, so it is redundant, yet that is how static call graphs over aproximate calls (f,g).

The advantage of having a static call graph is that we can see every single path possible, and the redundancies – function X with repeated outbound edges(X,Y) or inbound edges(Y,X) – can be removed.

What is important about the call graphs is that they can be analyzed by using graph logic.

3.1.5 Directed graphs and Node Attributes

Graph theory [1] is the mathematical area that focuses on the analysis of the structure of graphs. A graph can be directed or undirected, the former having vertices that do not depict a source and a target, but only a relation between the two, while the latter

does have a source and target, and as the name implies, having a direction, as it was explained in 2.1.4 .

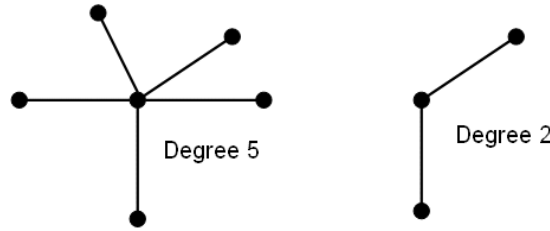
As it also has been explained, we are dealing with static callgraphs which are always directed graphs. Therefore, we set out to explore the different attributes that could be extracted. To this end, we explored graph attributes for classification from [13] which we found concise and well documented, as well as using generic formulas from [1].

We will be exploring, and explaining to detail what the considered graph attributes were, and which ones worked for our thesis.

Node degree

Every node has a degree which is the amount of its neighboring edges. So if a node has 2 outgoing edges and 3 incoming, its degree is 5. This works both for directed and undirected graphs.

Figure 3.4: Node degree



We thought of this value as extremely important, because it shows how intertwined a function is. What we mean by this, is that if it is very centric, the probability of a vulnerability spreading from this point or falling into the vulnerability is likely to be higher. Needless to say, this attribute was chosen to be implemented in our framework.

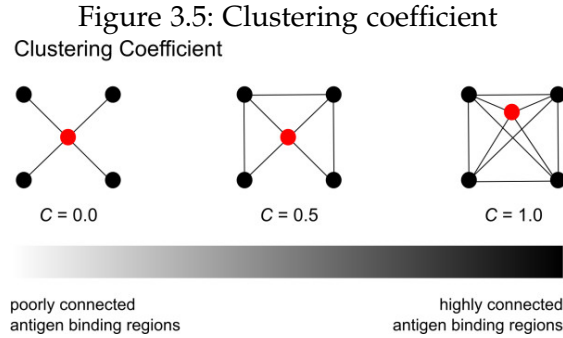
Clustering coefficient

The definition of this attribute is :

“For node u , the clustering coefficient $c(u)$ represents the likelihood that any two neighbors of u are connected.”[13]

In other words, the amount of triangles of node u , divided by the number of triples a node u has.

We also found this attribute to be relevant to our analysis, since it also shows the relations of a node to its surrounding nodes, but also how much it relies on other functions, as well as how much other functions rely on it.



Effective eccentricity

The definition of this attribute is :

“For effective eccentricity we take the maximum length of the shortest path from u , so that u can reach at least 90 percent of nodes in the graph. Effectiveness is a more robust measure if we take noise into consideration. The average effective eccentricity is the average of effective eccentricities of all nodes in the graph. ”[13]

This attribute would not be of good use in our case, since the likelihood of a node reaching 90% of the graph in a directed graph is very low, and this attribute would most of the time be extremely low, unless it was an interface, which would only be one node. Needless to say, this was discarded and completely disregarded.

Path length

The definition of this attribute is :

“Average path length (closeness centrality): The closeness centrality of a node u is defined as the reciprocal of the averaged total path length between node u and every other node that is reachable from node u . ”[13]

In other words, if we have the root be the node, and average the distance to its children nodes (every single one, regardless of their level), this will give us the closeness centrality of that given node.

This attribute complements the degree and the clustering coefficient nicely, and adds another angle of analysis. This was selected as another attribute to be used in our framework.

It is worth noting that the description we have given, and used only applies to directed graphs, where sources and targets are available, thus explicitly showing the relationships between nodes (e.g parent, child, etc...)

3.2 Node attributes found through MACKE

Getting outside of the realm of graph attributes, and into the results found by MACKE, we were able to get two very valuable node attributes, that we could use for every single function.

Vulnerabilities found by Macke

This node attribute depicts how many vulnerabilities MACKE was able to find, like shown in Figure 3.6 . For every vulnerability found, a counter would go up for the function affected. Therefore, if a function had 3 vulnerabilities found, that is the value this node attribute would hold for that specific function.

Macke bug chain length

This node attribute tells us how long a chain of calls was before the vulnerable function was called. To do this, we checked the phase in which each function was called. If a bug was found during phase two, we checked the caller and the callee of each function, until we reached the vulnerability. If a chain was, for instance, 5 functions long, that would be the value represented by this node attribute.

3.2.1 Exploring CVSS base scores, and their meaning

In our introduction to this thesis, we introduced CVSS – which we will be naming CVSS3, to explicitly specify that we use version 3.0 throughout this project – and its importance towards achieving our goal of assessing vulnerabilities as objectively as possible. We also introduced the base scores, what they mean, and how they impact the assessment. Now we will describe in detail what each base score can have as its value, and what each one means.

All of the following information has been taken from [7] as our source, to stay true and consistent.

Attack Vector (AV)

- Network (N) - A vulnerability exploitable with network access means the vulnerable component is bound to the network stack and the attacker's path is through OSI layer 3 (the network layer). Such a vulnerability is often termed "remotely exploitable" and can be thought of as an attack being exploitable one or more network hops away.

- Adjacent (A) - A vulnerability exploitable with adjacent network access means the vulnerable component is bound to the network stack, however the attack is limited to the same shared physical (e.g. Bluetooth, IEEE 802.11), or logical (e.g. local IP subnet) network, and cannot be performed across an OSI layer 3 boundary (e.g. a router).
- Local (L) - A vulnerability exploitable with local access means that the vulnerable component is not bound to the network stack, and the attacker's path is via read/write/execute capabilities. In some cases, the attacker may be logged in locally in order to exploit the vulnerability, otherwise, she may rely on User Interaction to execute a malicious file.
- Physical (P) - A vulnerability exploitable with physical access requires the attacker to physically touch or manipulate the vulnerable component. Physical interaction may be brief or persistent.

To summarize this base score, we can say that the more distance there can be between the attacker and the system, the more vulnerable the system is.

Attack Complexity (AC)

- Low (L) - Specialized access conditions or extenuating circumstances do not exist. An attacker can expect repeatable success against the vulnerable component.
- High (H) - A successful attack depends on conditions beyond the attacker's control. That is, a successful attack cannot be accomplished at will, but requires the attacker to invest in some measurable amount of effort in preparation or execution against the vulnerable component before a successful attack can be expected. For example, a successful attack may require the attacker: to perform target-specific reconnaissance; to prepare the target environment to improve exploit reliability; or to inject herself into the logical network path between the target and the resource requested by the victim in order to read and/or modify network communications (e.g. a man in the middle attack).

To summarize this base score, we can say that the more distance there can be between the attacker and the system, the more vulnerable the system is.

Privileges Required (PR)

- None (N) - The attacker is unauthorized prior to attack, and therefore does not require any access to settings or files to carry out an attack.

- Low (L) - The attacker is authorized with (i.e. requires) privileges that provide basic user capabilities that could normally affect only settings and files owned by a user. Alternatively, an attacker with Low privileges may have the ability to cause an impact only to non-sensitive resources.
- High (H) - The attacker is authorized with (i.e. requires) privileges that provide significant (e.g. administrative) control over the vulnerable component that could affect component-wide settings and files.

The more requirements the attacker requires, the less severe the vulnerability is. When it is regarded as "High" the CVSS3 score drops quite drastically, because it means that there are measures in place to keep attackers at bay.

Privileges Required (PR)

- None (N) - The attacker is unauthorized prior to attack, and therefore does not require any access to settings or files to carry out an attack.
- Low (L) - The attacker is authorized with (i.e. requires) privileges that provide basic user capabilities that could normally affect only settings and files owned by a user. Alternatively, an attacker with Low privileges may have the ability to cause an impact only to non-sensitive resources.
- High (H) - The attacker is authorized with (i.e. requires) privileges that provide significant (e.g. administrative) control over the vulnerable component that could affect component-wide settings and files.

The more requirements the attacker requires, the less severe the vulnerability is. When it is regarded as "High" the CVSS3 score drops quite drastically, because it means that there are measures in place to keep attackers at bay.

User Interaction (UI)

- None (N) -The vulnerable system can be exploited without any interaction from any user.
- Required (R) - Successful exploitation of this vulnerability requires a user to take some action before the vulnerability can be exploited.

If the attacker requires a user of the system to interact with, for example a malicious script, the severity would drop quite drastically.

Scope (S)

- Unchanged (U) - An exploited vulnerability can only affect resources managed by the same authority. In this case the vulnerable component and the impacted component are the same.
- Changed (C) - An exploited vulnerability can affect resources beyond the authorization privileges intended by the vulnerable component. In this case the vulnerable component and the impacted component are different.

This base score refers to the state of the system when attacked, and how many privileges the attacker will have after the attack in comparison to its prior unaffected state.

Confidentiality (C)

- None (N) - There is no loss of confidentiality within the impacted component.
- Low (L) - There is some loss of confidentiality. Access to some restricted information is obtained, but the attacker does not have control over what information is obtained, or the amount or kind of loss is constrained. The information disclosure does not cause a direct, serious loss to the impacted component.
- High (H) - There is total loss of confidentiality, resulting in all resources within the impacted component being divulged to the attacker. Alternatively, access to only some restricted information is obtained, but the disclosed information presents a direct, serious impact.

This base score refers to the amount of data accessible to the attacker after the attack. If this base score is high, the vulnerability will be affected severely.

Integrity (I)

- None (N) - There is no loss of integrity within the impacted component.
- Low (L) - Modification of data is possible, but the attacker does not have control over the consequence of a modification, or the amount of modification is constrained. The data modification does not have a direct, serious impact on the impacted component.
- High (H) - There is a total loss of integrity, or a complete loss of protection. For example, the attacker is able to modify any/all files protected by the impacted

component. Alternatively, only some files can be modified, but malicious modification would present a direct, serious consequence to the impacted component.

The more data is modifiable after the attack, the more severe the vulnerability is. This most of the time is linked to confidentiality since it handles privileges.

Availability (A)

- None (N) - There is no impact to availability within the impacted component.
- Low (L) - There is reduced performance or interruptions in resource availability. Even if repeated exploitation of the vulnerability is possible, the attacker does not have the ability to completely deny service to legitimate users. The resources in the impacted component are either partially available all of the time, or fully available only some of the time, but overall there is no direct, serious consequence to the impacted component.
- High (H) - There is total loss of availability, resulting in the attacker being able to fully deny access to resources in the impacted component; this loss is either sustained (while the attacker continues to deliver the attack) or persistent (the condition persists even after the attack has completed). Alternatively, the attacker has the ability to deny some availability, but the loss of availability presents a direct, serious consequence to the impacted component (e.g., the attacker cannot disrupt existing connections, but can prevent new connections; the attacker can repeatedly exploit a vulnerability that, in each instance of a successful attack, leaks a only small amount of memory, but after repeated exploitation causes a service to become completely unavailable).

3.3 Implementation phase

Now that we have a clearer picture of all the pieces of the puzzle, we need to make it all come together. In order for us to be able to do this, we had to make a very specific plan, and do all of the steps shown in this chapter sequentially, since each one depended on to one that came right before it.

3.3.1 Finding a reliable Vulnerabilities Database and suitable programs to analyze

The first thing we had to do was find a database of vulnerabilities, that was not only reliable, but also provided precise CVSS3 scores – or at least as detailed, and up-to-date as possible – so that we could use this information as a starting point.

The main idea behind getting vulnerabilities that have already been found and assessed manually, is that we could try to find the same vulnerability by different means, in this case, through composite symbolic execution using MACKE. By having done that, means that we would have a bitcode file, which could be then used to get a callgraph of the program. Here is where the CVSS3 scores come into play.

The CVSS3 scores from the database must be very accurate, reliable and thorough, since they would be our basis of analysis when comparing against the call graph. The main idea is that through the analysis of the callgraph and the attributes mentioned in 2.1.5 we could generate an algorithm that could assess these scores as precisely as possible.

Bugzilla database

First we started by taking a look at Bugzilla[3], since it is widely adapted in the industry and has a large developer base. They also can rate the severity of a bug – though manually done – which they call Severity as well, and describe as:

“How severe the bug is, or whether it’s an enhancement.”[3]

It’s worth noting that their use of the word “bug” is very broad in the Bugzilla project, and covers ultimately every type of vulnerability found.

They also divide severity into three four categories depending on their impact which are as follows:

Table 3.1: Bugzilla’s Severity Categories

Category	Description
Critical	crashes, memory leaks and similar problems on code that is written in a common enough style to affect a significant fraction of users
Normal	major loss of functionality
Minor	minor loss of functionality, misspelled word, or other problem where an easy workaround exists
Enhancement	Request for enhancement

Also, they do not use the severity of a given vulnerability only, they also use the Priority which they describe as:

“Engineers prioritize their bugs using this field.”[3]

The description of Priority can be quite obscure, and hard to understand, since it is not described what it means, and could change from one user of Bugzilla to the next. By combining Severity and Priority they get their Importance attribute which is their final assessment of a vulnerability.

While their assessment of the Importance is good, and could work, it was of no use to us since it relied on a subjective part, which they call Priority. Furthermore, they do not use CVSS, neither 2.0 nor 3.0, therefore any Bugzilla database would be of no use. Therefore, we moved on to NVD.

NVD - National Vulnerability Database

By looking at governmental vulnerability databases, we came across a USA based database, which not only is it up to date, but it also is funded by the government of said country. Their own description can be found below:

“The NVD is the U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP). This data enables automation of vulnerability management, security measurement, and compliance. The NVD includes databases of security checklist references, security related software flaws, misconfigurations, product names, and impact metrics.

Originally created in 2000 (called Internet - Categorization of Attacks Toolkit or ICAT), the NVD has undergone multiple iterations and improvements and will continue to do so to deliver its services. The NVD is a product of the NIST Computer Security Division, Information Technology Laboratory and is sponsored by the Department of Homeland Security’s National Cyber Security Division. The NVD performs analysis on CVEs that have been published to the CVE Dictionary. NVD staff are tasked with analysis of CVEs by aggregating data points from the description, references supplied and any supplemental data that can be found publicly at the time. This analysis results in association impact metrics (Common Vulnerability Scoring System - CVSS), vulnerability types (Common Weakness Enumeration - CWE), and applicability statements (Common Platform Enumeration - CPE), as well as other pertinent metadata. The NVD does not actively perform vulnerability testing, relying on vendors, third party security researchers and vulnerability coordinators to provide information that is then used to assign these attributes. As additional information becomes available CVSS scores, CWEs, and applicability statements are subject to change. The NVD endeavors to re-analyze CVEs that have been amended as time and resources allow to ensure that the information offered is up to date.”[4]

This database completely aligns with our needs, as it provides extremely reliable content – approved by the government of the USA, and backed up by the NIST (National Institute of Standards and Technology) – and has up to date CVSS3 scores, as

well as having a broad range of programming languages, including C/C++.

Getting vulnerabilities and their source code

Once we settled on the database, we knew we had to find vulnerabilities that met the following criteria:

1. Open source program
2. C or C++ code
3. Function where the vulnerability lies is documented
4. CVSS3 scores available

Knowing that we started going through the database, starting from the year CVSS3 started being used by NVD, which was 2016. After going through the entirety of the database, it was hard to find programs that met the aforementioned criteria, yet we did find them.

The programs found are:

- BlueZ 5.42
- AutoTrace 0.31.1
- AutoTrace 0.31.1
- GraphicsMagic 1.3.25
- Icoutils 0.31.1
- ImageMagic 6.0.4-8
- Jasper 1.900.27
- Jasper 2.0.10
- Libarchive 3.2.1
- Libass 0.13.3
- Libmad 0.15.1
- Libplist 1.12
- Libsndfile 1.0.28

- Libxml2 2.9.4
- Lrzip 0.631
- Openslp 2.0.0
- Potrace 1.12
- Rzip 2.1
- Tcpdump 4.9.0
- Tif Dirread
- Virglrenderer 0.5.0
- Ytnef 1.9.2

All of these programs met all of our requirements. We were able to find their source code, all of which were C and C++, and they all had extremely detailed CVSS3 scores.

3.3.2 Running all programs through MACKE

Now that we had the programs we could test if MACKE would be able to find the vulnerabilities noted in the NVD database. What had to be done first is run all of the make files in order to get all the source code needed for our specific system, and then run LLVM over them.

We took advantage of a tool developed internally at TUM called "make+llvm"[11] that does just that in one simple step, the only thing that is needed to be done is select the make file, and it takes care of the rest.

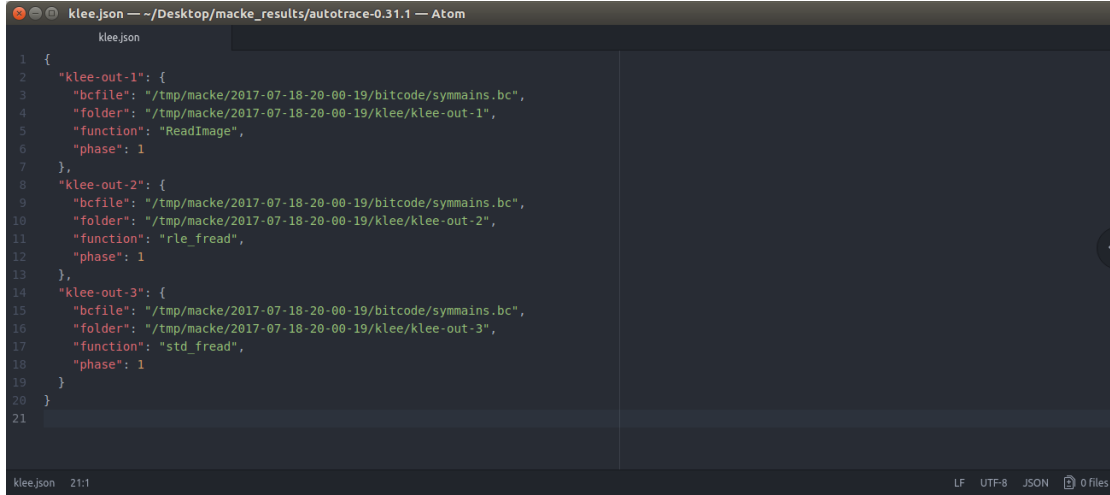
By the end of the make+llvm run, we are greeted with all of the bitcode files for all .c or .cpp files that were encountered in the program (and that were needed upon compilation). These files are crucial to our research because:

1. We can generate a callgraph of the program / function
2. We can run MACKE on them

With all of the files we followed to run MACKE on exactly the bitcode generated for the specific .c where the vulnerability was found and documented in the NVD database.

The results confirmed that compositional symbolic execution has a high source code coverage, as well as how reliable these findings were.

Figure 3.6: MACKE vulnerabilities found for AutoTrace 0.31.1



3.3.3 Analysis of the vulnerabilities found

The results gotten from MACKE were nothing short of outstanding. As seen on figure 3.6, MACKE found 3 vulnerabilities, two of which were documented on the NVD database, as shown in Figure 3.7 and 3.8:

As shown in the aforementioned figures, MACKE found a 100% of the vulnerabilities documented, on top of one more, which will come in handy in the future.

What we can do now by doing the analysis of this documented vulnerabilities is the following:

1. Store the CVSS3 for the functions found in the database AND in the MACKE results
2. Calculate the node attributes of all nodes (both where vulnerabilities have been found, and where none have been found)
3. Do these two previous steps for all programs mentioned previously
4. Analyze the correlation and covariance between the node attributes, and their CVSS3 scores.

After these steps are done, we would be in good standing to develop a learning algorithm that can predict the CVSS3 scores based on the node attributes.

Figure 3.7: NVD documented error for ReadImage

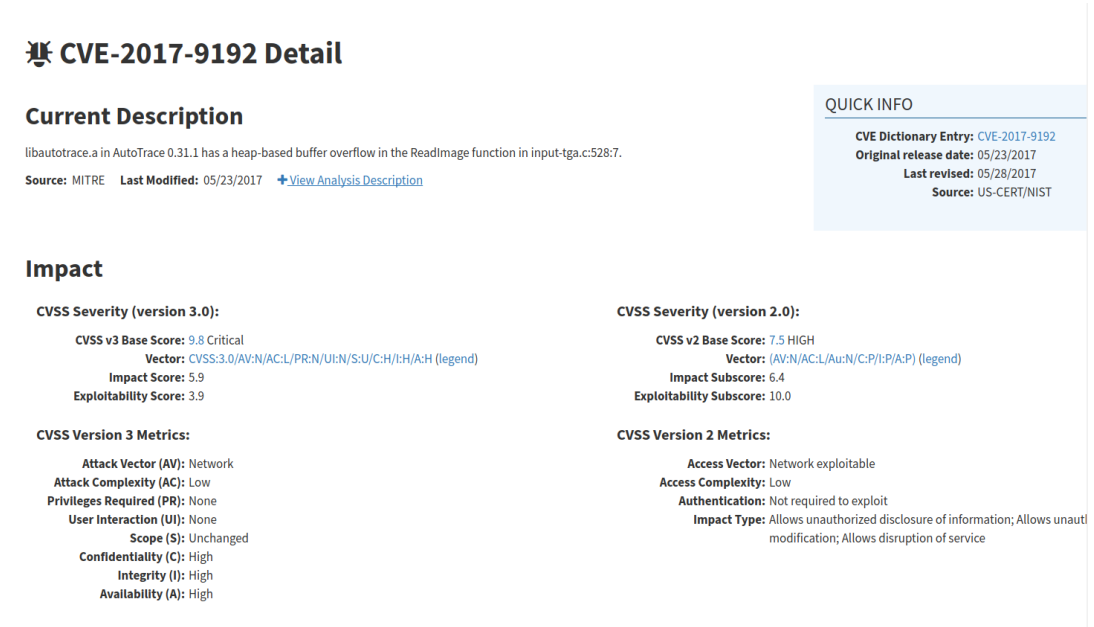
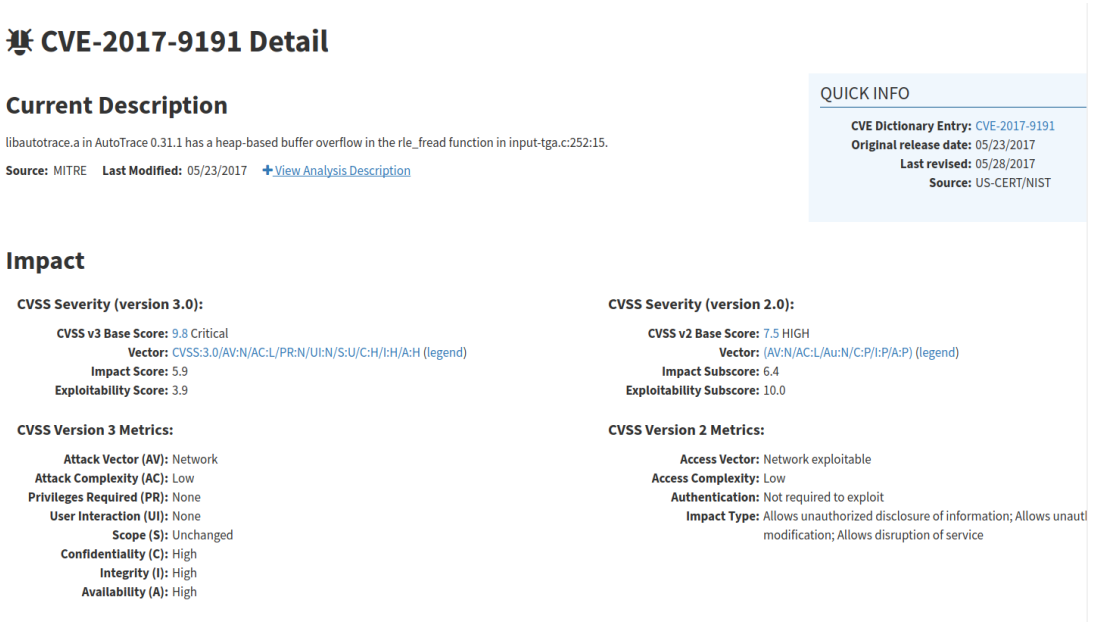


Figure 3.8: NVD documented error for rlefred



3.3.4 Calculating node attributes

A very crucial part of our thesis is the extraction of some node attributes on a per node basis, and to do this, as mentioned previously, we need to generate a callgraph of each program. In order to do this we require "llvm-opt" which is part of the LLVM [10] set of tools.

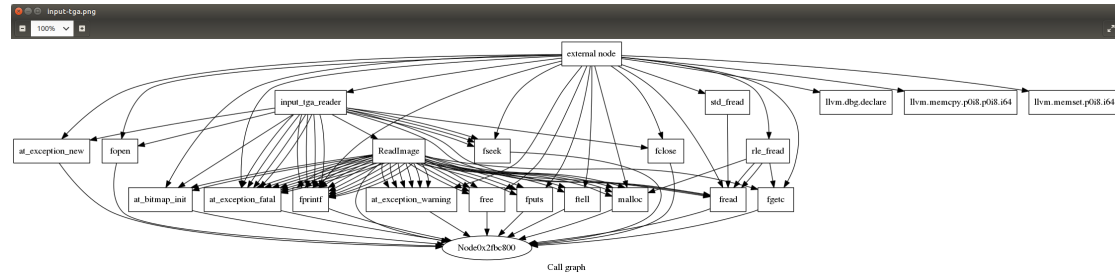
To extract the node attributes we need to run the following command:

```
$ opt -analyze -dot-callgraph BITCODE_FILE.bc
```

This command will generate a call graph in .dot file format, which is a graph description language [9] which is a static callgraph, which means it will display all relationships found in the code, yet not only the ones shown during runtime, and precisely what we need to analyze the graph features of it.

The callgraph for AutoTrace 0.31.1 looks as follows:

Figure 3.9: AutoTrace 0.31.1 callgraph with redundant edges

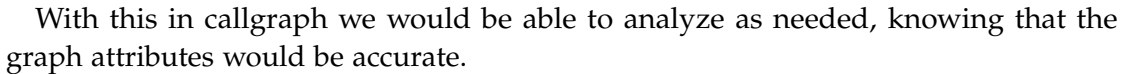


At first glance it is easy to see one of the underlying problems of static call graphs, which is having several edges that have the same source and target. The reason of this issue appearing in static call graphs is that, if for instance function *a* calls function *b* in different locations depending on its received parameters (e.g a branch of the function), all of these calls will be extracted and none will be disregarded. This not only would render the analysis of the graph useless, since all the values would be completely off.

Seeing this we set out to clean the graph, and remove all redundant edges. To do this we used the .dot file as our input in our node attributes program, which has been developed using Python as the programming language. A snippet of the part where we remove this redundancies looks as follows:

```
def generate_tree(entry_node, root, visited=None):
    if visited is None:
        visited = set()
    for link in data["links"]:
        if entry_node in link["source"]:
```

By doing this we did two things: we were able to create a tree representation of the callgraph as a structure in Python for easy manipulation and analysis, as well as the complete removal of all the redundant edges. The resulting callgraph is shown in Figure 3.10.

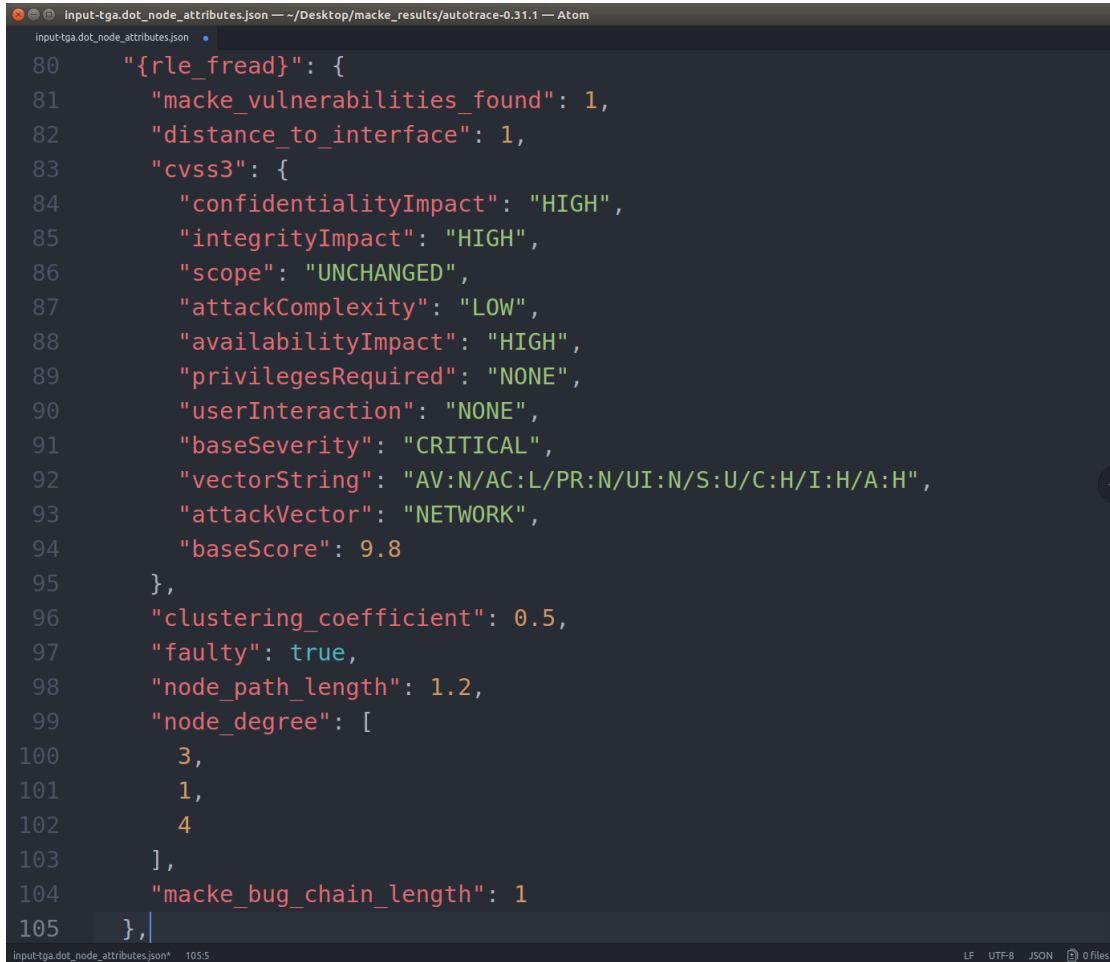


The node attributes resulting file encoded in JSON as shown in Figure 3.11 .

Right from the beginning, we set out to develop a way to test the correlation and covariance found between the node attributes, and the CVSS3 scores found in the NVD database. In order to do so, we developed a function, with the help of the library numpy [6] that would allow us to do so. The part of the entire codebase developed for this can be found at [14] .

Out of these results we were able to tell that the correlation for some node attributes

Figure 3.11: AutoTrace 0.31.1 node attributes JSON file



```
80  "{rle_fread}": {
81    "macke_vulnerabilities_found": 1,
82    "distance_to_interface": 1,
83    "cvss3": {
84      "confidentialityImpact": "HIGH",
85      "integrityImpact": "HIGH",
86      "scope": "UNCHANGED",
87      "attackComplexity": "LOW",
88      "availabilityImpact": "HIGH",
89      "privilegesRequired": "NONE",
90      "userInteraction": "NONE",
91      "baseSeverity": "CRITICAL",
92      "vectorString": "AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H",
93      "attackVector": "NETWORK",
94      "baseScore": 9.8
95    },
96    "clustering_coefficient": 0.5,
97    "faulty": true,
98    "node_path_length": 1.2,
99    "node_degree": [
100      3,
101      1,
102      4
103    ],
104    "macke_bug_chain_length": 1
105  },
```

was really small, which allowed us to have an idea how important, or irrelevant some node attributes were regarding some CVSS3 base scores. Furthermore, we also thought that while the results were not so encouraging, we needed to use this data for learning, and see how accurate it could actually be.

3.3.6 Learning CVSS base score values from node attributes

Once we had a ll of the previous parts developed, we set out to develop a learning function tha would:

- Have its X , or constant value be the node attributes.

- Have a different y value for each CVSS3 base attribute.

Therefore, we would be able to generate a model that based on the node attributes, and all of the CVSS3 scores at hand, could predict new base scores solely based on node attributes given to it.

In order to do this we used one of the biggest machine learning libraries for Python, called scikit [12], which allowed us to use several different machine learning algorithms to find whether or not some would fit our needs. The results will be discussed in detail in Chapter 4.

3.3.7 Framework implementation

test

Figure 3.12: Correlation found of node attributes to CVSS3 scores

```

ricardo@RicardoPC: ~
Correlation of CVSS3 Scores:

==== attackVector ====

clustering_coefficient : 0.477768443588
distance_to_interface : -0.0576490178928
macke_bug_chain_length : 0.355110412114
macke_vulnerabilities_found : 0.153846153846
node_degree : -0.271894680478
node_path_length : 0.096897032246
==== attackComplexity ====

clustering_coefficient : -0.285119738647
distance_to_interface : -0.181675477622
macke_bug_chain_length : 0.169376871474
macke_vulnerabilities_found : -0.0524142418361
node_degree : 0.255820461273
node_path_length : -0.0421430326747
==== privilegesRequired ====

clustering_coefficient : -0.0433219733755
distance_to_interface : 0.145966572245
macke_bug_chain_length : -0.419596332701
macke_vulnerabilities_found : -0.0649227473161
node_degree : 0.072451881988
node_path_length : -0.196789522791
==== userInteraction ====

clustering_coefficient : 0.292867470045
distance_to_interface : -0.187047417334
macke_bug_chain_length : -0.142771076228
macke_vulnerabilities_found : 0.146494858678
node_degree : 0.0595129698856
node_path_length : -0.0380182092101
==== scope ====

clustering_coefficient : 0.381411830069
distance_to_interface : -0.143266254496
macke_bug_chain_length : -0.00490278552445
macke_vulnerabilities_found : 0.169924546358
node_degree : 0.00959221997559
node_path_length : -0.0417378910141
==== confidentialityImpact ====

clustering_coefficient : -0.181084899345
distance_to_interface : 0.390285539961
macke_bug_chain_length : -0.204147709347
macke_vulnerabilities_found : -0.256021735264
node_degree : -0.134215256269
node_path_length : -0.0904024761156
==== integrityImpact ====

clustering_coefficient : -0.204247598173
distance_to_interface : 0.335776386335
macke_bug_chain_length : -0.204147709347
macke_vulnerabilities_found : -0.256021735264
node_degree : 0.0173833012638
node_path_length : 0.238114753557
==== availabilityImpact ====

clustering_coefficient : -0.204887109681
distance_to_interface : -0.180077722665
macke_bug_chain_length : 0.151986351258
macke_vulnerabilities_found : -0.0106202841474
node_degree : 0.246354034181
node_path_length : -0.286232572315
ricardo@RicardoPC:~$

```

4 Evaluation

In this chapter we will first show what our methodology for experimentation was, the reasoning behind each experiment, their value and contribution towards this thesis's goal, and how we went about doing each one. Secondly, we will show, and analyze all the results of said experiments.

4.1 Experiments

4.1.1 Docker container setup

4.1.2 Pre-processing the data

4.1.3 CVSS score assignment

4.1.4 Running MACKE on programs

4.1.5 Getting Call Graphs and Node Attributes

4.1.6 Learning Setup

Split dataset

4.1.7 Front-end implementation

4.1.8 Survey

4.1.9 Including Feedback

4.2 Results

4.2.1 CVSS Scores found

4.2.2 MACKE vulnerabilities found

Common vulnerabilities found on NVD

Unique undocumented vulnerabilities found

4.2.3 Learning results

4.2.4 Survey Results

4.2.5 Future work

5 Related Work

This chapter surveys previous work in the automatic assessment of vulnerabilities. We will see how their approach differs from what we have done, the viability of their work, in respect to what we have learned in this thesis. Moreover, we will see if what they have done matches in some way what we have also done, and try to mirror, or approximate their ideas to what we have researched and developed.

5.1 MACKE's ranking of vulnerabilities

As mentioned previously in 3.1.3, MACKE is a compositional symbolic analysis, of which we have talked extensively throughout this thesis, but only emphasising on its first two steps, while leaving the third one completely on the sidelines. The reason being, it has its own severity assessment tool, which is not automatized, and the results have to be parsed to get them. Nonetheless, the research done in [15] is relevant to us because of its closeness to the topic – assessing the severity of vulnerabilities found through symbolic execution – at hand.

In [15] 2.2.3 "Ranking the Vulnerabilities" it is mentioned that :

"A thorough compositional analysis for finding low-level vulnerabilities is more useful when there is a process to prioritize those vulnerabilities. After consulting with our industry partners, we decided to implement in our framework an interactive procedure to assign severity scores to vulnerable functions that are found in the analysis stages of MACKE. This severity score is based on the functions described below (with their intuition), and a weight (impact factor) between 1 and 5 associated with each function (i) The function `len chain(f)` returns a natural number representing the depth of function hierarchy through which a vulnerability in `f` might be exploited. It has the impact factor `L`. If a function can be exploited through a long hierarchy, it's more likely somebody forgot to sanitize the exploit input. (ii) The function `is int(f)` returns a boolean according to whether the function `f` is an exposed interface or not. It has the impact factor `I`. Vulnerability in an exposed interface, such as the main function, is easily exploitable and must be fixed with higher priority. (iii) The function `vuln inst(f)` returns the number of distinct

instructions that were found to contain a vulnerability and has the impact factor N . More vulnerabilities strongly indicates a missing input sanitization check somewhere in the function. (iv) The function `d interface(f)` returns the proximity (length of nested function chains) of the function to an exposed interface and has the impact factor D . A vulnerable function closer to an exposed interface may be easier to exploit. (v) The function `is outlier(f)` returns a boolean depending on whether the number of vulnerable instructions found (Section 2.2.1) is much greater than the average number of vulnerable instructions per function in the program. The intuition behind this is the same as that for vulnerable `inst(f)`. It has the impact factor O . We formulated the above functions with our industry partners and, based on our combined intuitions on the programs that we analyzed, we used the following function, `s`, to calculate the total severity value:

$$s(f) = L * \text{len chain}(f) + I * \text{is int}(f) + N * \text{vuln inst}(f) + D * \text{d interface} + O * \text{is outlier}(f)$$

Functions with higher severity scores are, in our view, more vulnerable to attacks. The specific values for the impact factors are also, as the function `s` itself, dependent on the context of development and vulnerability analysis, as we clarify once more in Section 3. As a final presentation step, `MACKE` color codes the ranges of severity scores for all functions in the program and displays the call graph, with function and instruction level details of the test cases that cause a vulnerability to be exposed with compositional analysis.”

To decompose what has been done by them is extremely important, and putting it into perspective, we can see where we align and where our ideas deviate, and see how this area of research could be improved upon in the future.

5.1.1 `MACKE`’s severity assessment

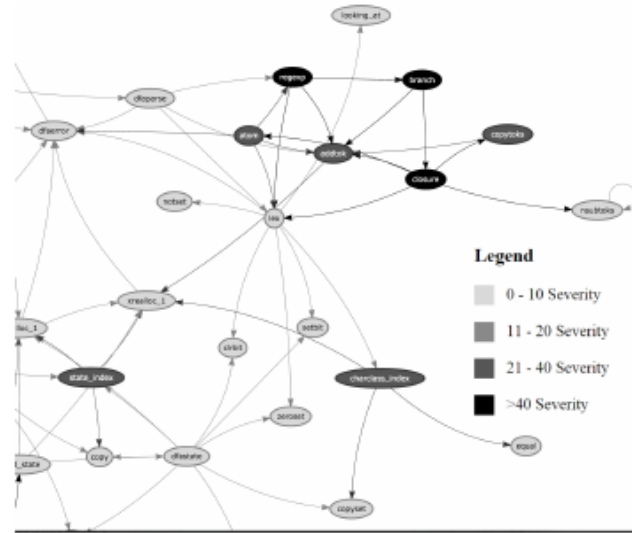
As it was previously quoted, they use a formulaic approach to find the severity of a vulnerability. They used the following formula :

$$s(f) = L * \text{len chain}(f) + I * \text{is int}(f) + N * \text{vuln inst}(f) + D * \text{d interface}(f) + O * \text{is outlier}(f)$$

Decomposing the formula is of great importance. Knowing that $s(f)$ is the severity of a given function s , we continue to evaluate the variables involved in the formula.

- L - Impact factor of len chain(f).
- len chain (f) - The depth of function hierarchy through which f can be exploited.
- I - Impact factor of is int (f).
- is int(f) - Whether the function is an interface or not.
- N - Impact factor of vuln inst (f).
- vuln inst(f) - Amount of vulnerabilities found in a function
- D - Impact factor of d interface(f).
- d interface (f) - Proximity of the function to an exposed interface.
- is outlier(f) - If the amount of vulnerabilities found in the function is greater than the average for the program.

Figure 5.1: MACKE's severity ranking framework using ($L=3, I=5, N=2, D=4, O=1$)



First, it is worth mentioning that their use of impact factors differs from our approach, but it is clearly seen as to what it was done. The reasoning behind it was to disregard these values when they were not found, thus being multiplied by 0 and therefore not being taken into account.

L and len chain (f)

When looking at $L * \text{len chain}(f)$ we can see that it is doing exactly the same we are doing in our framework, which we called "Macke bug chain length".

They do not go into the intricacies of how it was done, but we can deduce that they are using MACKE's phase 2 values to come up with a chain of function calls, to know how long it is. L is its impact factor.

I and is int(f)

These values tell us how far a function is from an entry point or "interface", which we do not use, since we used the external node of every call graph as our interface for all of our measurements. Therefore calculating this value in our framework would have been futile (only one node would have been a 1).

N and vuln inst(f)

This value yet again aligns with another value used in our framework, this time with "Macke bug chain length". What $\text{vuln inst}(f)$ does is return how many vulnerabilities were found by MACKE in that specific function. N is its impact factor.

D and d interface(f)

This value is identical to one we also use : "distance to interface". It has the same meaning, and they do the calculation the same way, the only difference being that they calculated several interfaces instead of just one as is in our case (using the external node, always). D is its impact factor.

is outlier(f)

This value, being a boolean like many others, but without an impact factor, depicts only whether the function has more vulnerabilities than the average found for the entire program. This we did not use, nor consider, though it can have some importance. The main reason why this didn't cross our minds is that generating an average could be misleading sometimes, if the values were generally low, and just a few functions

had anything above it, they would have been counted as outliers, which would have impacted the severity, when in truth, it could be that it is just one vulnerability spreading throughout the function. MACKE would find this as several vulnerabilities, though for example something as simple as the sanitization of a variable could go a long way between having 5 vulnerabilities in a function and having 0, which does not mean it is more critical than another function that has a buffer overflow that grants root privileges.

MACKE's severity assessment

The severity retrieved from using the aforementioned $s(f)$ is a number, which can be a number from 0 to > 40 , which can fall into several categories as shown in Figure 5.1 [15]. The problem with this approach, in our minds, is that it is not standardized. While MACKE might be using these values, and color coding, it will differ greatly from what the industry is doing, therefore rendering these results useless when given to someone who is clueless to the idea of symbolic execution, not to say MACKE. We strongly believe that their approach was one in the right direction, and has given us ideas, which we used as building blocks for this thesis. Our work would not have been possible without this previous knowledge.

What can we take away?

The most important thing to be taken away from this research is that it corroborated our approach in regards to the node attributes and MACKE results used in our interface. We believe that we have taken a step in the right direction by using a similar approach, but assessing the severity in terms of the biggest vulnerability assessment standard in the world, CVSS[7].

5.2 A Novel Automatic Severity Vulnerability Assessment Framework

In [17] a new framework named by its developer "Automatic Security Vulnerability Assessment Framework" or ASVA, aims to resolve the problems that Quantitative Vulnerability Assessment Standards (QVAS) such as CVSS have.

The focus of this paper is mainly on network security, rather than general vulnerabilities, though the proposed framework is generic. Furthermore, the framework has taken CVSS into account, and tried to build upon it.

5.2.1 How to improve upon current QVAS?

They mention how CVSS, being an objective, authoritative, and transparent quantitative vulnerability assessment standard (QVAS), is still difficult to implement in all projects, their reasons being some of the following:

- The number of vulnerabilities grows ever so larger in all software projects, and adding CVSS scores only adds to the complexity of filing, fixing and deploying vulnerabilities.
- There is not a huge collection of bugs in vulnerability databases because of it – CVSS – being so recently adopted.
- The categorization of certain base scores of CVSS can be subjective, depending on who is making the assessment.

What it is proposed in this paper [17], is something similar to what we have developed: not a new QVAS, but the introduction of an application process of an existing QVAS, which in our case was CVSS.

The main methodology used to get the base scores would be by using Text Mining. With their tool ASVA, they would compute the severity values and ranks of vulnerabilities using other QVAS, when there was a lack of information.

Interetingly, they also used the NVD database, along with three non-NVD Databases. They also introduce three dimensions (accuracy, coverage rate and dispersity) to analyze the results, results show that the accuracy of severity rank is 90.1% and the dispersity is perfect. They then go on to explain the reasoning behind errors in the vulnerability assessment, which are all related to machine learning.

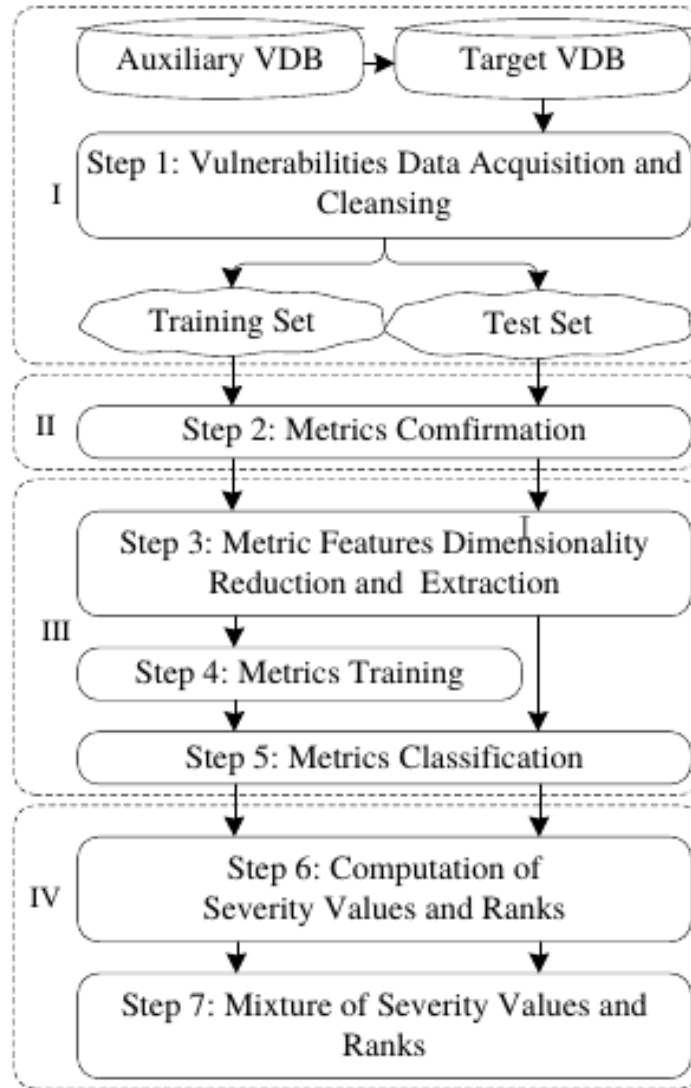
5.2.2 Pipeline of the ASVA framework

The following is an abstract of the explanation of each step given directly from the author in [17] and shown in Figure 5.2:

- Stage I includes Step 1. The task of Stage I is acquiring and cleansing data. We can obtain the one-to-one corresponding vulnerabilities between Auxiliary VDB and Target VDB as Training Set, and obtain the rest vulnerabilities of Target VDB as Test Set.
- Stage II includes Step 2. The task of Stage II is determining the classification metrics. ASVA contains three modes, all of them have different metrics.

- Stage III includes Step 3, Step 4 and Step 5. The task of Stage III is classifying metrics with Text Mining and obtaining the values of metrics.
- Stage IV includes Step 6 and Step 7. The task of Stage IV is assigning values of metrics to formulas of the QVAS and computing the severity values and severity ranks of vulnerabilities.

Figure 5.2: The process of ASVA



Since our thesis does not touch on the topic of Text Mining, nor is its intention, we will focus on the matter at hand which is the extraction of the results, in this case the CVSS scores. In this regard this tool uses, unfortunately, CVSS 2.0, which had a much more simplistic way of assigning base scores, but also was not so accurate.

With that in mind, we can tell that while they used a similar approach to ours (e.g using NVD as the main vulnerability database used because of its reliability among other attributes, and their fixation on using CVSS as the main standard), it is clearly they took a more statistical approach, rather than an algorithmic one, such as the one we did.

They mention that when their coverage rate is 100%, the rank – base score – accuracy is 82.5%, but when the coverage is 76% the accuracy goes up to 90.1%. We believe that this has to do with the statistical approach that they took, and how based on a smaller amount of words analyzed, the hit chance is higher.

5.2.3 Is ASVA an improvement over our current framework

Their approach is extremely interesting, and the way they went about predicting the CVSS base scores by "cleansing" as they mentioned in [17], and then checking for patterns is noteworthy, and something we had not considered. Nonetheless, we do not believe that their platform is better, nor worse, but complementary.

The main reasoning behind it being that just parsing through the text, as explained in this paper is not enough, since they remove too much context when cleaning the text. What if a single digit, is removed, and it was an assignment that lead to a vulnerability, or if a function called was disregarded when cleansing the source code.

It has been shown throughout this thesis that the relationship between functions, the vulnerabilities found in them, and their neighboring functions are of extreme importance when piecing together the CVSS score of a function.

Therefore, we can say that while this was an extremely good approach, and their results are outstanding, it still lacked the precision, and the right approach for it to be adopted, not to mention the use of CVSS 2.0, which has already been phased out in favor of a new and much improved version. Furthermore, combining the work from ASVA and our approach could yield an even better result, and we would strongly encourage future work to take both approaches into consideration, since both cover the same problem from a different angle.

6 Conclusion

2 pages

List of Figures

2.1	Program callgraph	6
3.1	C program analyzed by EXE	10
3.2	EXE symbolic execution	11
3.3	AutoTrace 0.31.1	12
3.4	Node degree	15
3.5	Clustering coefficient	16
3.6	MACKE vulnerabilities found for AutoTrace 0.31.1	26
3.7	NVD documented error for ReadImage	27
3.8	NVD documented error for rlefreed	27
3.9	AutoTrace 0.31.1 callgraph with redundant edges	28
3.10	AutoTrace 0.31.1 callgraph without redundant edges	29
3.11	AutoTrace 0.31.1 node attributes JSON file	30
3.12	Correlation found of node attributes to CVSS3 scores	32
5.1	MACKE's severity ranking framework using (L=3,I=5,N=2,D=4,O=1)	37
5.2	The process of ASVA	41

List of Tables

3.1	Bugzilla's Severity Categories	22
-----	--	----

Bibliography

- [1] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. 1976.
- [2] e. a. Cadar C. *EXE: Automatically generating inputs of death*. 2006.
- [3] bugzilla.org contributors. *Bugzilla*. URL: <https://www.bugzilla.org/> (visited on 10/03/2017).
- [4] bugzilla.org contributors. *NIST*. URL: <https://nvd.nist.gov/> (visited on 10/03/2017).
- [5] D. E. Cristian Cadar Daniel Dunbar. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. 2008.
- [6] N. developers. *NumPy*. URL: <http://www.numpy.org> (visited on 10/03/2017).
- [7] I. (FIRST.Org. *Common Vulnerability Scoring System v3.0: Specification Document*.
- [8] A. G. *Demand-Driven Construction of Call Graphs*. 2000.
- [9] Graphviz. *Graphviz*. URL: <http://www.graphviz.org/Documentation.php> (visited on 10/03/2017).
- [10] L. D. Group. *The LLVM Compiler Infrastructure*. 2006. URL: <http://llvm.org/> (visited on 10/03/2017).
- [11] T. Hutzelmann. "Compositional Analysis for Exposing Vulnerabilities – A Symbolic Execution Approach." MA thesis. Technische Universität München, Universität Augsburg und Ludwig-Maximilians-Universität München, 2016.
- [12] INRIA et al. *SciKit*. URL: <http://scikit-learn.org/stable/#> (visited on 10/03/2017).
- [13] G. Li, M. Semerci†, B. Yener, and M. J. Zaki. *Graph Classification via Topological and Label Attributes*. 2011.
- [14] R. Nales. *Thesis's Codebase*. URL: <https://github.com/Rydros/thesis> (visited on 10/03/2017).
- [15] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer. *MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution*. 2016.
- [16] U. K. A. S. B. Sathe. *Data Flow Analysis: Theory and Practice*. 2009.

Bibliography

- [17] T. Wen, Y. Zhang, Y. Dong, and G. Yang. *A Novel Automatic Severity Vulnerability Assessment Framework*. 2015.