FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# AST-Based Large-Scale Vulnerability Analysis for C/C++

**Diogo Ferreira de Sousa**


Mestrado em Engenharia Informática e Computação

Supervisor: João Bispo

October 31, 2022

# AST-Based Large-Scale Vulnerability Analysis for C/C++

**Diogo Ferreira de Sousa**

Mestrado em Engenharia Informática e Computação

October 31, 2022

# Abstract

As technology keeps advancing, our reliance on computers and software for various daily tasks increases, and with it, the importance of maintaining security and preventing malicious attacks. Breach of security in software can lead to multiple adverse outcomes, ranging from less severe, such as preventing users from using a service for a short time to very dangerous, such as theft of sensitive personal information.

To prevent these attacks, making sure the software in question does not have any security vulnerabilities is paramount. One of the methods we can use to achieve this is static analysis, analyzing programs before executing them, usually by inspecting their source code. This project explores a novel way of analysing C/C++ programs at scale.

These languages provide some access to low-level capabilities, making them powerful to use. In turn, they open programs up to more vulnerabilities, requiring more attention to security than other languages. This fact, combined with the popularity of C and it being an established language, ensuring its programs remain secure is a high priority due to its wide usage and widely documented vulnerabilities.

This project combines two existing tools: a tool that parses and extracts information from C/C++ code by generating data structures and analysing them and a tool used for patching C/C++ code, allowing us to parse it without requiring its dependencies. This novel approach lets us increase the amount of information that we can extract and have an easier time working on detecting vulnerabilities that we identify from analyzing this information. We tested and evaluated this solution compared to other tools using a large dataset of source code.

The unique patching capabilities of this solution allowed us to eliminate the strict need for dependencies when analysing C/C++ code. These capabilities increase the coverage of code that can be handled, reduce the effort required when analysing large amounts of code, and enable a method of scanning that would otherwise be possible. We expect this approach to be viable in helping identify security issues in software, particularly in situations where it is troublesome to include all the dependencies for the code to be analysed.

**Keywords**: Computer Security, Software Vulnerabilities, Abstract Syntax Tree (AST), Static Analysis

# Resumo

À medida que a tecnologia avança, a nossa dependência nos computadores e software para várias tarefas diárias aumenta, e consequentemente, a importância de manter a segurança e prevenir ataques maliciosos. Falhas na segurança de software podem levar a vários resultados adversos, desde menos graves, como impedir que os usuários usem um serviço por um curto período de tempo, até bastante perigosos, como roubo de informações pessoais confidenciais.

Para prevenir estes ataques, é fundamental garantir que o software em questão náo tenha vulnerabilidades de segurança. Um dos métodos que é usado para conseguir isto é análise estática, isto é, analisar os programas antes de executá-los, geralmente ao inspecionar o código-fonte. Este projeto explora uma abordagem nova para analisar programas escritos em C/C++ em escala.

Estas linguagens fornecem algum acesso a capacidades de baixo nível (e.g.: pointers), tornando-as mais completas. Por sua vez, isto abre os programas a mais vulnerabilidades, exigindo mais atenção à segurança do que outras linguagens. Este facto, combinado com a popularidade de C e C sendo uma linguagem antiga e estabelecida, garantir que os seus programas permaneçam seguros é uma alta prioridade devido ao seu amplo uso e vulnerabilidades bem documentadas.

Este projeto combina duas ferramentas existentes: uma ferramenta que analisa e extrai informação de código C/C++ ao gerar estruturas de dados e analisá-las e uma ferramenta que atualiza código C/C++, permitindo analisá-lo sem precisar das suas dependências. Esta abordagem original permite aumentar a quantidade de informação que se pode extrair, e tornar mais fácil detetar vulnerabilidades que nós identificamos ao analisar esta informação. Esta nova solução foi testada e avaliada comparada com outras ferramentas usando um grande *dataset* de código-fonte.

As capacidades de atualização desta ferramenta permitem eliminar a necessidade de dependências ao analisar código C/C++. Estas capacidades levam a um aumento da cobertura de código que pode ser analisado, reduzindo o esforço necessário para analisar grandes quantidades de código, e permitindo um método de análise que não seria necessário sem estas capacidades. Espera-se que esta abordagem seja viável para ajudar a identificar falhas de segurança em software, particularmente em situações onde é dificil incluir todas as dependências do código para análise.

**Keywords**: Segurança de Computadores, Vulnerabilidades de Segurança, Análise Estática

# Agradecimentos

Gostaria de deixar algumas palavras de agradecimento para quem me ajudou a chegar a onde estou hoje.

Quero agradecer à minha família, que ofereceu-me a educação e apoio que me permitiu focar nos meus objetivos.

Quero agradecer também aos meus amigos e colegas com quem tive o prazer de partilhar todos os desafios e sucessos nestes últimos anos.

Por último, quero agradecer ao meu supervisor, por todos os concelhos e ajuda que recebi no decorrer do meu trabalho, particularmente com o uso das ferramentas do laboratório.

Diogo Ferreira de Sousa

*"Try not to become a person of success,*
*but rather, try to become a person of value."*

Albert Einstein

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| ACE | Arbitrary Code Execution |
| AST | Abstract Syntax Tree |
| CFG | Control Flow Graph |
| CPG | Code Property Graph |
| CWE | Common Weakness Enumeration |
| PDG | Program Dependence Graph |
| HDF | Hierarchical Data Format |
| TP | True Positive |
| TN | True Negative |
| FN | False Negative |
| FP | False Positive |

# Chapter 1

# Introduction

## 1.1 Context

The amount of computing devices and software in use is steadily growing, and so is the userbase of these devices and software. People were using traditional methods to perform many activities only a few years ago, and now various software applications are replacing these methods, some examples being payment methods and communication.

Creating software is not a simple task. It is unlikely to develop products while having source code without any faults during this process, and these faults in the code can lead to security vulnerabilities. If they are not detected and fixed in time, the possibility of malicious attacks causing damage to their users arises.

The growth in software usage creates a higher demand for security since higher usage means more incentives and more gains for attackers. Detecting vulnerabilities before attackers can exploit them is very important for any software owner to prevent these attacks from succeeding.

Tools dedicated to detecting and mitigating vulnerabilities help minimize the risk of attack. Clava [35] is a C/C++ source-to-source compiler that can analyse code and perform transformations. For this project, we explored the viability of a new approach to detect and mitigate vulnerabilities based on an existing solution using Clava, using its analysis capabilities to search data structures derived from code for known vulnerability patterns.

## 1.2 Motivation

While compilers already perform some simple checks to find basic security faults while compiling code, a specialized tool can afford to spend more time and resources to ensure vulnerabilities are not present. We can never rely on analysis to know if a program is entirely rid of vulnerabilities. However, we want to create a new vulnerability scanning solution that improves current detection approaches and expands the scenarios where they can be applied. We can verify this solution by using a large labeled dataset of source code, with some code presenting vulnerable functions.

There already exist many vulnerability scanning tools that can achieve this. However, the primary motivation behind this project is handling scenarios that other tools cannot. Namely, we want to analyse code on a large scale, that is, take a large batch of source code and search through the code regardless of whether its dependencies are present or not. With Clava and TranslationUnitPatcher, a tool that can patch source files so we can parse them without their dependencies (explained in chapter 2), we have a unique position among the other programs. Tools that rely on scanning text can ignore dependencies but often cannot get as much information out of the code due to not working with parsed code. Meanwhile, tools that rely on parsing the code to analyse data structures do not work without all dependencies and project setup. With the combination of our two solutions, we can have the best of both approaches.

Clava also has code transformation features, which, if leveraged correctly, have the potential to apply automatic code fixes when encountering simple errors, another point that is of value and uncommon among current vulnerability scanners.

## 1.3 Objectives

The previous work is very extensible to make it easier for future development, and we ensured it stayed that way during our work. We want to establish the core parts of the new tool. Then, for any possible future work, there are clear avenues of expanding the number of cases it can detect as potential vulnerabilities or working on the patching process of TranslationUnitPatcher to patch more files successfully. Since we rely on extensibility to help further development, it is paramount that our solution keeps this factor.

The main goal of this work is to overcome one of the biggest obstacles when working with analysing C/C++ code. Files in projects using these languages will often depend on other files/libraries, so it is usually challenging to analyse them in isolation without setting up their project due to missing dependencies. It is impossible to know the functionality of the function calls defined in the dependencies if we only analyse the file in isolation. Using TranslationUnitPatcher, we can parse files in isolation by detecting compilation errors and iteratively fixing them to analyse the code. It lets us analyse C/C++ files on a large scale by ignoring the rest of the project and analysing whatever file we wish. We can build a solution that not only detects vulnerabilities but comes with the somewhat unique ability to parse files in isolation with the help of TranslationUnitPatcher. The patcher's inclusion makes these solution's results similar to fuzzy parsing's ( 3.2), but without requiring a custom parser, allowing any C/C++ compiler to parse the code after patching. Taking this goal into consideration, the main question we wish to answer with our research is the following: "Is it possible to parse C/C++ code using a standard parser without requiring its dependencies successfully?".

Additionally, if possible, we want to ensure our solution has good motives to be considered alongside other tools, apart from its novel features. Therefore, we need to verify that it is effective at vulnerability scanning. To achieve this, we want our solution to have performance that is a

good starting point for doing this job. We can evaluate this using standard pattern recognition performance metrics, such as accuracy, precision, and recall.

In summary, these are the objectives we are working to achieve with our work:

- Keep the easy extensibility factor on our tool to facilitate future work
- Achieve acceptable vulnerability scanning performance metrics
- Develop solution with the ability to parse C/C++ code using a standard parser with missing dependencies

## 1.4 Document Structure

The current chapter explains the context and importance of mitigating security vulnerabilities in software and the motivation behind working on a solution for this.

In the second chapter, we give some background necessary to understand the following chapters. Talking about the C/C++ languages, some security vulnerability examples, data structures used for analysing code, static analysis (which will be the approach used for scanning vulnerabilities), and Clava, the tool we will be working with for this project.

In the third chapter, we present some research on existing solutions to the problem we face, each with its different approach to tackling it. Namely, we also present some tools that use these different approaches to solve the problem and the static analysers that we will compare to our final product.

In the fourth chapter, we define our proposed solution's implementation and the methods we employed to validate it. We detail the dataset we selected, the use of TranslationUnitPatcher, the configurations we used on our various comparison analysers, and our methods of gathering statistics from the dataset analysis.

In the fifth chapter, we list the types of experiments we ran and explain/interpret the results obtained from experimenting with the implementation defined in the previous chapter.

In the sixth chapter, we summarize and conclude what we described in this document and provide some ideas and considerations for future work on this project.

# Chapter 2

# Background

## 2.1 The C Language

Despite their age, C and C++ are still some of the most used languages [30] in the world. Therefore, attacks on C programs are a concern as their higher popularity means attackers potentially have more targets to exploit. Namely, the Linux kernel is written in C [38], and many systems worldwide rely on Linux and its kernel to be safe. Successfully exploiting the Linux kernel is a major issue to prevent.

Another factor contributing to C's significant attack target is its powerful memory management capabilities [49]. These capabilities allow the programmer to control memory usage more directly than a language with automatic garbage collection, which, depending on the specific case, can allow for much better performance [42]. However, these same capabilities make C an unsafe language, as misusing them can open up vulnerabilities [52, 50].

C's dynamic memory allocation allows us to create storage for data precisely when needed and in the amount we need. It is also effortless to delete said storage whenever we do not need it anymore or change the data structure's size if we require more space after some operation. We do this using specific function calls, such as malloc(), to allocate memory and free(), to de-allocate memory once it is not required anymore. Taking advantage of this leads to very memory-efficient programs if used correctly. If misused, however, it can lead to unwanted issues.

A consequence of incorrect use is memory leaks. That is, when we allocate memory, use it, and once it is not required anymore, we do not free it. Memory leaks lower system resources as memory is effectively locked up without being used. If the program runs for a long time, it will likely crash due to exhausting all available resources. Another consequence would be the incorrect use of pointers, such as a dangling pointer, which can lead to security holes, as explained in section 2.2.2.

## 2.2 Security Vulnerabilities

Not following the proper memory management guidelines can lead to various consequences. For example, it can lead to program crashes due to segmentation faults when using a null pointer. However, it can also open up security bugs and allow an attacker to perform Arbitrary Code Execution (ACE) [50], that is, running commands on a target machine. ACE is dangerous, as it can potentially lead to the attacker having complete control of the system, depending on the privileges he has/can obtain (with privilege escalation) during the attack [44]. Even without complete control, unwanted damage can still occur.

The Common Weakness Enumeration (CWE) is a community-developed list of software and hardware weakness types [7]. It lists various types of vulnerabilities that can be present in systems and encompasses many different programming languages and computer systems. For this work, we will focus on the types in the context of C/C++ programs. Two examples of vulnerabilities in the CWE that apply to C/C++ are buffer overflows (CWE-119 [20]) and dangling pointers (CWE-825 [22]). We further explain how these vulnerabilities work in sections 2.2.1 and 2.2.2. They both can access memory in unintended ways. The critical difference is that one ignores spatial memory safety (buffer overflows go over memory bounds). In contrast, the other ignores temporal memory safety (dangling pointers access memory that was valid in the past but not currently) [31].

### 2.2.1 Buffer Overflow

As listed in the CWE, buffer overflows consist of reading/writing past the bounds of a buffer, leading to performing these operations on memory locations associated with other data instead of the one we were trying to access. An attacker can take advantage of this to perform ACE by fulfilling two objectives: inject attack code into the program's address space, and get the program to execute that code [39]. Buffers can exist in different memory locations: in the stack, the heap, or the static data area, and the methods of attacking vary depending on the location. We show a stack-based buffer overflow in the following simplified example [52].

When a function is called, a stackframe is created with information about it and a return address indicating the next instruction after the current function finishes executing. If attackers can write past the bounds of a buffer, they can overwrite the information about the function located on the stack and overwrite the return address. By overwriting the return address with the desired value, an attacker can control the program's next instruction. The attacker can also inject attack code within the same write operation on the buffer if memory allows. We can achieve ACE by pointing to the new return address at the injected code.

In figure 2.1, we can see this in practice. An injected piece of code fits in the bounds of the buffer, and afterwards, we add some extra data until we reach the return address on the stack, at which point we alter the return address to point to the injected code. After executing the current function, the program will continue by executing the new injected code instead of returning to the normal program flow.

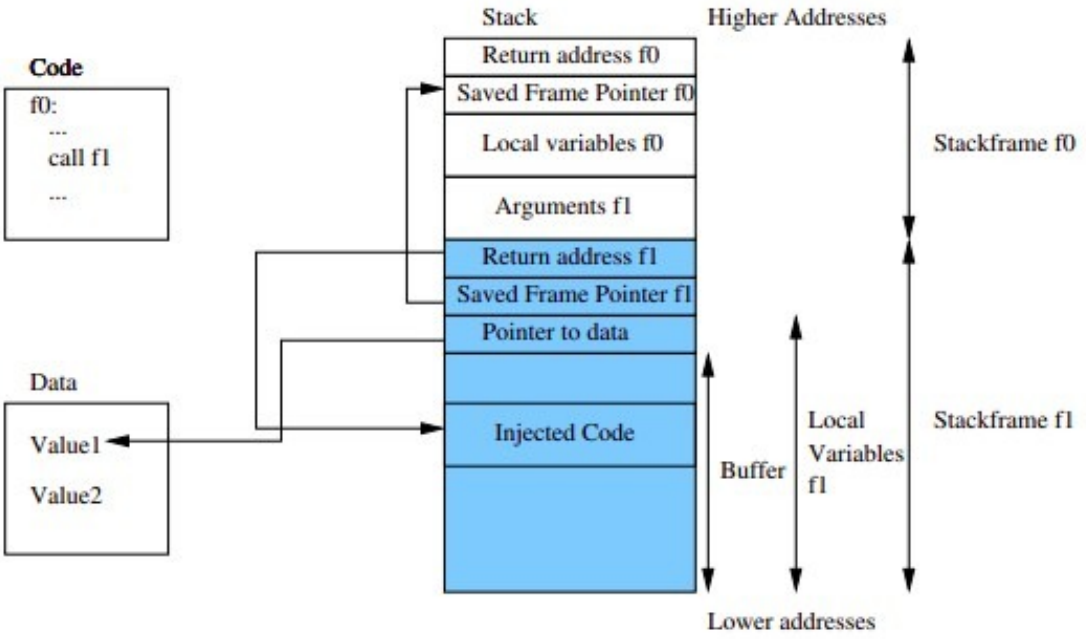


Figure 2.1: Stack-based buffer overflow attack [52]

### 2.2.2 Dangling Pointer

Dangling pointers, or Expired Pointer Dereference (CWE-825), consist of a program freeing memory while keeping the pointer intact and accessing it later. When the pointer is accessed, there is a chance that the program will now access data that is in use somewhere else, as it is no longer allocated to that specific process. This access generally results in a crash or undefined behavior if the value was changed somewhere along the program normally [52]. However, if an attacker can exploit this value by arranging data to end up in this particular place in memory before being dereferenced, he can obtain control of the program [31].

However, a particularly dangerous case of dangling pointers, the Double Free (CWE-415 [21]), refers to when memory already de-allocated gets freed a second time. This vulnerability, when exploited, corrupts the memory management process and can cause the program's malloc() calls to return the same pointer. If the attacker has control over the data in one of these pointers, executing a buffer overflow attack is possible.

In figure 2.2, we have an example of a dangling pointer being exploited [31]. The pointer initially points to a value of type A that contained a pointer field. However, the program freed it and now contains some random value of type B. The program will still treat this value as a valid instance of type A initially, so if this is a regular occurrence, the program will likely crash. However, suppose an attacker constructs the value of type B and manages to get that value in the exact location of the previously freed memory. In that case, the program will access the constructed data, and in this case, since it is a pointer field, the attacker can supply a function pointer, altering the flow of the program.

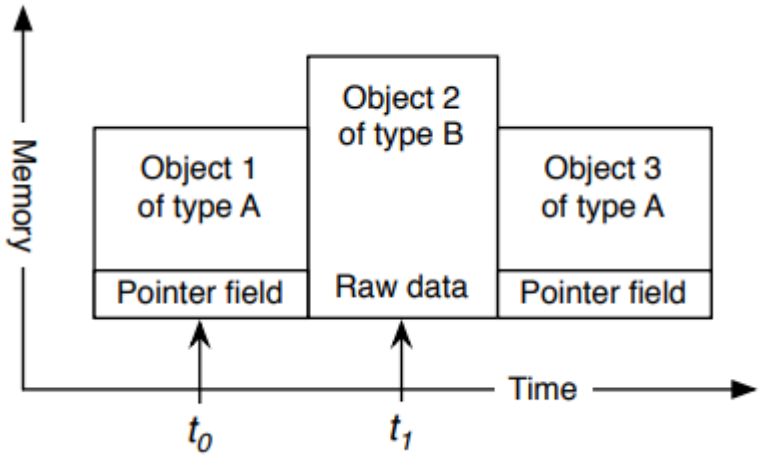


Figure 2.2: Dangling pointer attack [31]

## 2.3 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a data structure, in the form of a tree, representing the structure of code [33]. ASTs are often used with compilers, program analysis, and program transformation to make these processes easier. ASTs will be important in the context of this work as we will be dealing with analysis and transformations, namely with Clava, and Clava makes use of ASTs to perform these two processes.

In the context of analysis, we can search for nodes that match patterns attributed to specific vulnerabilities [48]. As an example, a common way of having a buffer overflow vulnerability in C is with the strcpy() function, as this function does not check bounds, so a buffer overflow will occur if we try to copy a string to a buffer where the string is larger than the buffer. Using an AST to represent the code, we can look for nodes that contain the strcpy() function call, then look at its parameters, and finally get the possible sizes of these parameters to see if a buffer overflow can happen.

Figures 2.3 and 2.4 show an example of a simple function and its representation as an AST, respectively.

```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

Figure 2.3: AST Example - Function [1]

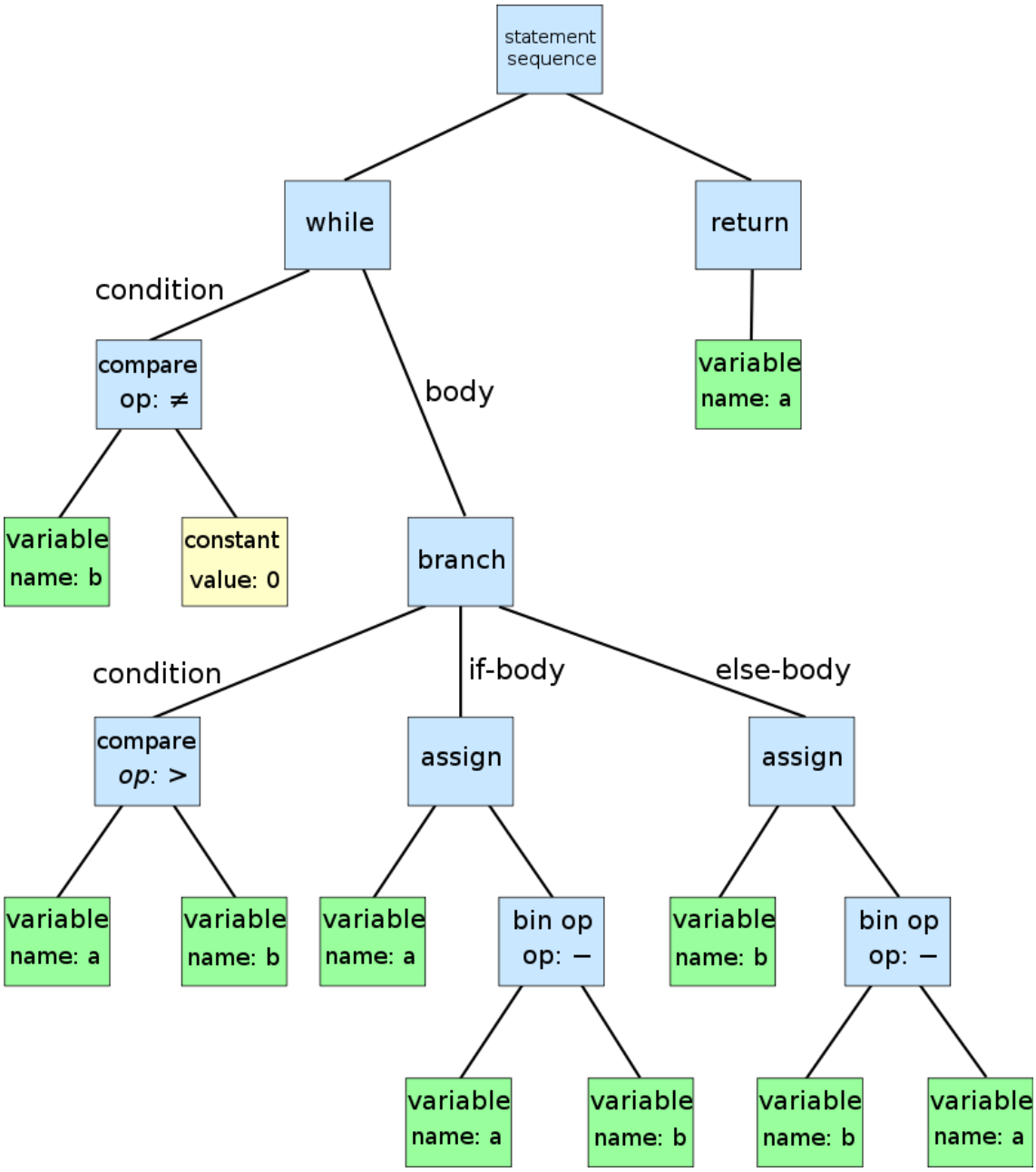


Figure 2.4: AST Example - AST [1]

## 2.4 Clava

As explained in section 1.2, Clava is a C/C++ source-to-source compiler with code analysis capabilities that we can leverage to create a vulnerability scanning tool. By giving source code as input to Clava, the tool will parse the code and generate data structures (AST and CFG) from the parsed code. With the help of Clava's API and user-created scripts using JavaScript, we can build queries that go through the data structures that Clava generates after code parsing to search for patterns that match known vulnerabilities.

For example, one of the queries we built into our tool searches for occurrences of CWE-457 [15] (Use of Uninitialized Variable) and CWE-119 [9] (Improper Restriction of Operations within the Bounds of a Memory Buffer). As shown in a code snippet in figure 2.5, this query analyses the AST to look for nodes corresponding to an array access. The query analyses information in the node to determine if said access is out of bounds or if the array we are trying to access is uninitialized.

```
if ($child.instanceOf("arrayAccess")) {
    var arrayName = $child.arrayVar.code;
    for (var result of boundsResultList) {
        if (result.arrayName === arrayName) {
            var indexes = $child.subscript.map(node => node.code);     // list of indexes in square brackets
            for (var i = 0; i < indexes.length; i++) {
                if (indexes[i].length > 1) {        // formats list of indexes
                    indexes[i] = indexes[i].substring(1, indexes[i].length - 1);
                }
                if (result.initializedFlag === 0) {
                    result.unsafeAccessFlag = 1;
                    result.line = $child.line;
                    continue;
                }
                if ((indexes[i] > result.lengths[i] - 1) || (indexes[i] < 0)) {     // access out of bounds
                    result.unsafeAccessFlag = 1;
                    result.line = $child.line;
                    continue;
                }
            }
        }
    }
}
```

Figure 2.5: Code Snippet of a Clava query

One of the main advantages of this approach using queries to analyse data structures, compared to other traditional static analysis tools that rely on text pattern matching, is easy customization and expansion of the vulnerabilities that we wish to search for by adding, removing, or even creating queries based on our needs. Another significant advantage is having data structures to work with instead of just text. Using these data structures can enable different approaches to navigating the code and searching for patterns, depending on the analysis performed.

However, due to it relying on parsing the code, it also has a few downsides. Notably, running Clava with this intention takes much more time than running text-based analysis tools (shown in section 5.3). The difference in time comes from the extra initial parsing step taking much more time than just analysing data (whether text or data structures). Also, Clava will not work on source code with compilation errors, unlike text-based tools. We mitigate this downside by introducing the use of a tool (TranslationUnitPatcher, explained in detail in section 4.2) that patches source code with compilation errors to create header files that attempt to fix these errors, permitting parsing and subsequent Clava analysis.

## 2.5 TranslationUnitPatcher

TranslationUnitPatcher [3] is a program that is part of the Clava repository that attempts to check the code for parsing errors iteratively. If it finds an error, a fix is applied depending on the error code obtained from the failed parsing attempt. For example, if the program expects to use a missing global variable of type int, it will create that variable when parsing. The program then creates a header file for each processed source file that contains all these fixes. The end goal is successful parsing after modifying the source file to include its new respective header file.

Figures 2.6 and 2.7 show an example of an isolated (missing its dependencies) source .c file and its corresponding .h file created after a successful patching attempt. When looking at the original .c file by itself, it is missing the definitions of the functions: rb_yield_values(), rb_int_succ(), and rb_ary_new4(). It also misses the definitions of NODE, VALUE, u1, and u1.value. Inspecting the .h file, we can see that all these definitions are covered - the patcher created functions definitions to return an appropriate value (in this case, 0) and also defined the missing types/structs appropriately. After these changes, even though the program would not function correctly due to missing the actual content of the dependencies, it does not report any errors while parsing, so the main function in the .c file can be analysed successfully.

```c
#include "function_98_patched.h"
TYPE_PATCH_00 enumerator_with_index_i(VALUE val, VALUE m, int argc, VALUE *argv)
{
    NODE *memo = (NODE *)m;
    VALUE idx = memo->u1.value;
    memo->u1.value = rb_int_succ(idx);

    if (argc <= 1)
    return rb_yield_values(2, val, idx);

    return rb_yield_values(2, rb_ary_new4(argc, argv), idx);
}
```

Figure 2.6: Example source code file

```c
#define NULL 0
typedef int TYPE_PATCH_2008;
typedef int TYPE_PATCH_2009;
typedef int TYPE_PATCH_2006;
typedef int TYPE_PATCH_2007;
typedef int TYPE_PATCH_00;
typedef int VALUE;
typedef struct {
    TYPE_PATCH_2006 value;
} TYPE_PATCH_2005;
typedef struct {
    TYPE_PATCH_2005 u1;
} NODE;
TYPE_PATCH_2008 rb_yield_values(...) { return 0;}
TYPE_PATCH_2009 rb_ary_new4(...) { return 0;}
TYPE_PATCH_2007 rb_int_succ(...) { return 0;}
```

Figure 2.7: Example patched header file

The patching capabilities of TranslationUnitPatcher let us solve the problem of missing dependencies shown in section 1.2, by providing a fix to the errors that come from trying to parse the problematic code. Successfully integrating the patcher into our approach is key to its viability.

## 2.6 Static Analysis Metrics

To evaluate the performance of static analysers (some shown in section 3.3), we can use performance metrics that are used in pattern recognition/information retrieval that apply to a subset of data resulting from a search. Namely, in the context of static analysers, these metrics aim to quantify the number of vulnerabilities that are correctly detected and the number of vulnerabilities that slip by undetected. We use seven of these terms/metrics throughout our work to quantify the performance of our proposed approach compared to other existing ones. These are the explanations for each of them:

- **True Positives (TP)** - Detection of a real, present vulnerability
- **False Positives (FP)** - False alarm, a vulnerability is said to be present when it is not
- **False Negatives (FN)** - Vulnerability present that went undetected
- **True Negatives (TN)** - No vulnerability report, with no vulnerability present
- **Accuracy** - Correct results, calculated by $\frac{TP+TN}{TP+FP+FN+TN}$
- **Precision** - Detected vulnerabilities from total detections and false alarms, calculated by $\frac{TP}{TP+FP}$
- **Recall** - Detected vulnerabilities from total vulnerabilities, calculated by $\frac{TP}{TP+FN}$

# Chapter 3

# Related Work

## 3.1 Code Property Graph

More data structures exist for source code representations with a similar purpose to ASTs (section 2.3). The Program Dependence Graph (PDG) [41] is one of them, and it makes explicit both the data and control dependencies for each operation in a program. It allows us to discover all operations that can change the value of a variable. The Control Flow Graph (CFG) [32] is another representation. The flow part of the graph stands for the order of statement execution. The control part stands for the conditions met for different execution paths. Like ASTs, each has its uses in aiding the analysis of programs, namely for security. The PDG is helpful when looking for vulnerabilities that arise from dependencies. Meanwhile, the CFG helps search for vulnerabilities that depend on statements previously executed in the control flow, such as dangling pointers (section 2.2.2).

The Code Property Graph (CPG) is a novel data structure that combines the previously described structures: the AST, the CFG, and the PDG [53, 4]. Using CPGs for program analysis, we can take advantage of each structure's features at the same time [4]. Taking advantage of multiple features is very useful for finding security vulnerabilities. Some become easier to find, relying on one of the three structures or using a mix. Keeping every piece of information on the same structure makes it easy to traverse it looking for the information we need.

In table 3.1, taken from a CPG article [53], we have an analysis of the possible combinations of data structures that we can rely on to cover certain vulnerability types. Since the CPG is a complete combination, naturally, it can cover anything the other data structures can cover on their own and more.

Figure 3.1, taken from the same article, shows an illustrative example of a CPG. It is a very promising approach to use when working with data structures since, as previously established, we can work with the information that three different data structures provide simultaneously.

Our approach will rely on traversing ASTs and CFGs for information, data structures that Clava can generate and provide an interface to work with. However, we can learn how to model

| Vulnerability Types | Data Structures | | | |
|---|---|---|---|---|
| | AST | AST+PDG | AST+CFG | AST+CFG+PDG |
| Memory Disclosure | | | | X |
| Buffer Overflow | | (X) | | X |
| Resource Leaks | | | X | X |
| Null Pointer Dereference | | | | X |
| Missing Permission Checks | | X | | X |
| Integer Overflows | | | | X |
| Division by Zero | | X | | X |
| Use After Free | | | (X) | (X) |
| Integer Type Issues | | | | X |
| Insecure Arguments | X | X | X | X |

Table 3.1: Vulnerability coverage by data structure [53]

specific vulnerabilities by exploring CPGs that contain them and adapting what we have learned to implement their detection in our solution.

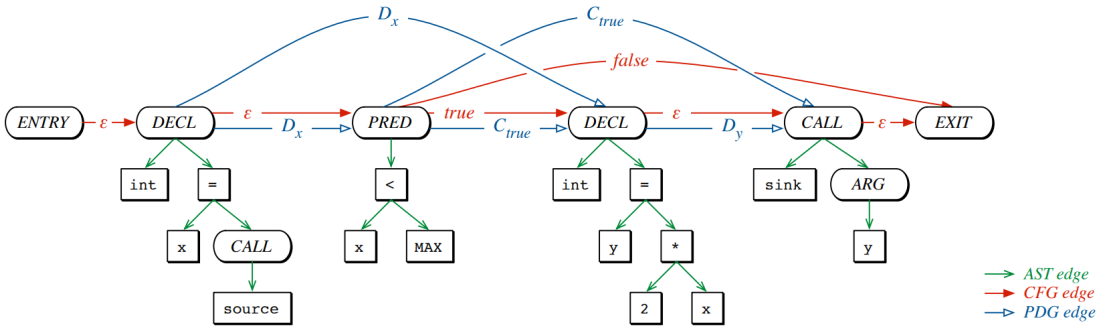


Figure 3.1: Code property graph example [53]

## 3.2 Fuzzy Parsing

As mentioned in section 1.3, a critical obstacle for this work is analysing C/C++ code while not having complete access to that code's project or if some part of the code is missing. An approach to get around this problem is using a fuzzy parser. A fuzzy parser works by recognizing only parts of a language according to some specification or set of rules [36]. Using a fuzzy parser means we do not need to parse considering elements from an entire language, so it is possible to process incomplete code containing errors by selectively choosing what to parse and what to ignore [46].

With fuzzy parsers, we can analyse C/C++ code in isolation because we can ignore the parts of the code that come from dependencies and generate errors if we do not have access to the whole build environment. By selectively parsing the code we do know and have access to, we can focus on scanning only this code, which is the one that is relevant to us.

### 3.2.1 Joern

Joern is a platform for robust analysis of C/C++ code [27, 28]. Joern is an example of a platform that uses a fuzzy parser and code property graphs (explained in section 3.1), solving the problem of missing code and generating a representation of the code's syntax, control-flow, data-flow, and type information. We can perform code analysis using search queries that can be manually formulated to find vulnerabilities with this information. Joern is also extendable, allowing users to include additional information in the graph and extend the query language appropriately to fit their needs.

This platform has similar characteristics to what we will be working on: it can analyse code in isolation and traverse data structures to find vulnerabilities. However, it still relies on a lot of manual work and query building, while our tool is intended to be more automatic. Regardless, we can take various good pointers from researching this approach for its use of fuzzy parsing and code property graphs.

## 3.3 Vulnerability Detection and Mitigation

Given the age and popularity of the language, as established previously in section 2.1, it is no surprise that many static analysis tools already exist to scan C/C++ code to find vulnerabilities and fix errors. We present in this chapter a few of these tools.

### 3.3.1 VulDeePecker

VulDeePecker is a vulnerability detection tool that uses a deep learning-based approach to detect software vulnerabilities [47]. Traditional deep learning is not usually suitable for these problems (vulnerability detection) due to a major obstacle in representing software programs in a form suitable for the learning algorithms. VulDeePecker overcomes this by assembling code into code

gadgets, which are some lines of code semantically related to each other, labeling them as vulnerable or not vulnerable, and then using these as input for the machine learning model. It is trained with two extensive code datasets, totaling 61638 code gadgets. After training, we can give the model the program we want to analyse, and it will extract its code as code gadgets and evaluate it using the trained model.

It shows very good false positive and false negative results, with a False Positive Rate of 5.7% and a False Negative Rate of 7.0%; however, the results can vary greatly depending on the data used in model training due to the nature of deep learning. This research, in particular, offers much insight into the evaluation and analysis of experiment results and testing different datasets and vulnerabilities compared to other tools, which are tasks that we will need to perform as well, even though our approach is different.

### 3.3.2 ITS4

ITS4 is a static vulnerability scanner for C/C++ code that aims to have a good middle ground between accuracy and efficiency [51]. Its main goal is to make finding vulnerabilities easier by having a database of potential problems instead of relying on the user to be aware of various scenarios. It also aims to lower the number of false positives encountered with traditional methods - high false positives can lead to vulnerabilities not being found by manual audit due to programmers investing less effort. Finally, it attempts to be efficient enough to be used alongside a programming environment, alerting programs with real-time feedback to errors introduced in the code.

This work predates many static analysers, so its main comparison is to manual audits with grep. Nevertheless, it has a comprehensive database of many vulnerabilities encountered in C. It can easily be modified to disregard particular vulnerabilities or include more if necessary, and it is very efficient. Its intended feature of providing real-time feedback is a particular point of interest.

### 3.3.3 Text-Based Analysis

Many other simpler static analysis tools perform basic checks for function calls that can prove unsafe, such as functions with buffer overflow vulnerabilities such as strcpy() or gets(). These tools generally do not have good FP/FN results. However, they all have the advantages of being simple and easy to use. Each tool also focuses on excelling at one specific aspect for each one to be used in different use cases. Even without perfect accuracy, different tools will pick up a variety of vulnerabilities that others will not. Therefore, while using only one will not catch many vulnerabilities, we can always use a mix of them and have decent coverage. It goes without saying, of course, that catching any vulnerability in the first place is better than not doing any security-related work at all.

Cppcheck is one of these tools, focused on having very few false positives and designed to analyze C/C++ code even with non-standard syntax [8].

Flawfinder is another one, focused on its simplicity and sorting each unsafe function/vulnerability found by risk level [24]. This risk level is based on the function call and the parameters

used. It primarily does simple text pattern matching, so there are many things it will not pick up, but it can still be helpful as a quick check, as it can always pick up on some simple errors.

RATS (Rough Auditing Tool for Security) is another tool focused on being very fast and easy to integrate without causing issues [2]. It focuses on finding buffer overflows and Time-of-check to time-of-use race condition problems.

This paper [45] studies these three tools comparatively. They tested them with a test suite containing 118 vulnerabilities listed in the CWE. In table 3.2, we have the number of vulnerabilities each tool could detect.

As previously discussed, the detection ratio is not very favorable. However, it also proves that while using only one tool is insufficient to ensure total security, using many can be beneficial. It shows that despite one tool failing to detect vulnerability X, another tool was often able to detect that one, and vice-versa.

| Tools | Number of Vulnerabilities Detected | Detection Ratio |
|---|---|---|
| Flawfinder | 52 | 52/118 |
| RATS | 84 | 84/118 |
| Cppcheck | 59 | 59/118 |

Table 3.2: Vulnerabilities detected per analysis tool [45]

## 3.4 Comparison Tools

To assess the performance and feasibility of our Clava-based tool, we have to establish some comparisons to other existing programs. As previously shown in section 3.3.3, it is impossible to claim one solution to be the best. Instead, it is often a good idea to use multiple. However, it is still necessary to prove that our approach is viable enough to be considered compared to the others.

In order to take more relevant conclusions from the comparisons, we decided to use three other analysers. Two of them are based on text analysing, Cppcheck and Flawfinder. Meanwhile, CodeQL (3.4.1) has a similar approach to Clava's by performing analysis at the AST level. Even before experimentation, we expect significant differences between Cppcheck and Flawfinder due to both focusing on different aspects [24, 8]. Cppcheck tends to be broader, detecting more types of CWEs. It also focuses on high precision, that is, a low amount of false positives, even if it leaves many undetected issues. Meanwhile, Flawfinder tends to focus more on a few specific vulnerabilities and on triggering a hit even if the actual risk level is low to ensure there are very few vulnerabilities undetected, even at the expense of a high false positive rate. We confirmed these findings during experimentation with the two tools in section 5.3.

### 3.4.1 CodeQL

CodeQL is an open-source semantic code analysis engine that allows users to write and run queries on codebases and search through code as if it was data [5]. Useful queries can then be shared with

other users, enabling users to skip the query writing process and get packs of queries already created and tested by others or contribute additional ones. We first need to create a database to use the tool, giving the engine the source files and a method to build them. After we create the database, we can select packs of queries to run on the database based on our needs and subsequently interpret the results.

Compared to Cppcheck/Flawfinder's approach, there are a few main advantages. First, similarly to Clava, the ability to traverse data structures created from parsing code will consistently benefit in analysing specific patterns compared to just reading text. Secondly, users sharing their tried and tested queries, paired with the fact that we can choose which queries we want to run, makes the solution a lot more customizable for what a user wants. We can potentially save a lot of time and effort going through unnecessary results if we wish to only focus on specific issues. It also lets users easily create a query of their own if they want to cover an additional problem not present in any of the "default" queries. Despite this technically also being possible on the other tools, given they are open source, the process is significantly more straightforward using CodeQL.

# Chapter 4

# Implementation

In order to evaluate our proposed approach, we need to envision and develop an environment in which we can test our prototype and other tools, gather data from these tests, and have an established process on how to analyse this data to arrive at our evaluation conclusions. This chapter details each step of the plan we came up with for implementing these tests and analyses.

In summary, it consists of choosing a dataset (4.1), applying the source code patching (4.2), setting up the tools (4.3), preparing our prototype (4.4), and finally, running tests and gathering statistics from the results obtained to formulate our evaluation (4.5).

## 4.1 Dataset Selection

To test our scanning solutions, we need a dataset consisting of source code that contains vulnerabilities and is labeled for them, so we can know if our tools are correct in their results or not. For this purpose, we considered two different datasets: Draper VDISC [23] and Juliet [29].

Both have their share of differences. Draper focuses on four vulnerabilities in particular: Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119 [9]), Buffer Copy without Checking Size of Input (CWE-120 [10]), Use of Pointer Subtraction to Determine Size (CWE-469 [16]), and NULL Pointer Dereference (CWE-476 [17]). All instances of these four vulnerabilities have their own label, while all other vulnerability instances have the label "CWE-other" (and are present in relatively lower numbers). Draper also has a lot more source code available, 1.27 million functions, to be precise. However, due to time constraints and limitations of our current approach, for this study, we decided to work with 10% of them (127476 functions).

The Juliet dataset instead covers 118 different CWEs, all accurately labeled, and has 64099 test cases compared to Draper's 1.27 million functions. However, instead of only having vulnerable code, Juliet's test cases include different scenarios depending on the number of files needed for one test case, flow variants, whether they are based on classes, and a couple more variables. Ultimately, these factors made the setup of Juliet more complex and consisted of features that we considered unnecessary for our study.

Therefore, we decided to use the Draper dataset for our research. The focus on 4 CWEs, in particular, was a crucial point, given that we would not have the time/resources to build our tool for analysing a lot of different CWEs. We consider that focusing on only a few CWEs was necessary.

The dataset we chose had the functions all stored in files using the Hierarchical Data Format (HDF) file format, specifically `HDF5` files. HDF5 is a file format designed to store and manage large amounts of data [26]. We had to separate and organize the various functions contained in the file, as we could not perform code analysis with any of our tools with the functions stored in the `HDF5` format. To achieve this, we developed a short python script using `h5py` [25], a library that provides an interface with the format, to separate each entry in the dataset into two files: a .c file containing the function and a .txt file containing the vulnerability labels of that function. We ended up with 254952 files, half of them being .c files and the other half .txt files (one .c file and one .txt file for each function). It is worth noting that even though the dataset consisted of C and C++ functions, we stored all of them in .c files due to having no distinction in the dataset for both programming languages. However, this is not a problem, provided the tools used can detect that the files have C++ code despite the inappropriate extension.

## 4.2 Source File Patching

The files extracted from our chosen dataset consist of only the typical source code contained in .c files, missing all the #include directives and respective headers/dependencies. The usage of Clava (2.4) and CodeQL (3.4.1) require parsable code, and one of our objectives was to develop a method to parse C/C++ code using a standard parser without requiring the code's dependencies. With that said, we had to find a way to solve this problem.

To achieve this goal, we paired the use of this dataset with TranslationUnitPatcher 2.5. When we apply the patching to the dataset's .c files, it will attempt to successfully generate a corresponding .h file that fixes the parsing errors, allowing us to continue the analysis process with the vulnerability scanning tools.

The patching program has some limitations. The major problem we encountered was the amount of time it took to run through the dataset - which we expected given that the program tries to compile the file on every iteration, checking for errors. In this case, using the default settings means it tries to compile each file 100 times. However, the main problem causing the long runtime is not using a multi-threaded approach; the program runs entirely in single-thread. There is an option to run it with multiple threads, but we had to stick with single-thread due to bugs that would lower the number of files successfully patched using multiple threads. There were some attempts at fixing these bugs, but they were unsuccessful - this is another avenue to explore for future work on this project. To resolve this, we ran multiple instances simultaneously using groups of dataset files. The setup consisted of two Windows computers executing the program, running one Linux virtual machine on each computer, and executing the program for four concurrent executions. This setup was the best solution we found, as executing the program multiple times on the same computer was impossible. We did not time the total patching time precisely, as it was difficult to

monitor all four processes simultaneously, but it was sometime between 12-15 hours. On the other hand, the patching process only needs to be applied once, and the results can be cached until there are changes in the file.

Since the compilation process is quite complex with various errors, TranslationUnitPatcher is not yet complete to the point of being able to patch all files successfully; we were able to parse 46.6-48% (depending on the tool/setup used) of the dataset files successfully. In some cases, it might not be able to parse due to the inability to handle some errors, and in other cases, it might introduce some unwanted changes to the code. The introduction of patching caused some of our scanning tools to detect additional CWEs compared to the unpatched files. This last fact could be positive or negative on a case-by-case basis, depending if the patching allowed the tool to parse the code better or if it introduced unwanted changes due to improper handling of the errors received. We address these findings in more detail in chapter 5.

## 4.3 Scanning Configurations

In this section, we detail how we set up the configurations of each tool used for the processed files, the versions of the programs used, the different groups of files used, and the queries we sourced to use in the case of CodeQL and Clava. The results of the experimentation performed with these setups will be shown and interpreted in chapter 5.

### 4.3.1 Cppcheck

The version used was 2.8, and we set up the analyser to only show errors and warnings (no style, portability, or performance warnings).

### 4.3.2 Flawfinder

The version used was 2.0.19, and unlike Cppcheck, we tested two different configurations: one using default values; and the other with a minimum risk level of 2 and with the `-falsepositive` flag. We used this second configuration to make use of Flawfinder's customizable risk level (by default, it triggers a hit on any vulnerability that triggers even a low risk). We can then run a group of files with both configurations to evaluate the potential differences.

### 4.3.3 CodeQL

There are three versions we can choose to use CodeQL: a GitHub integration, a Visual Studio Code extension, and a command-line interface. The GitHub integration was not a good choice for our project, as the integrations caps analysis to 5000 results or 10MB file size for the generated file containing the results. As we are working with a large dataset, we predicted a very high chance of hitting these caps, so we prepared to prevent this. We chose the command-line interface version as it was easier to configure than the Visual Studio Code extension. We used CodeQL CLI version v2.10.5.

Before running queries on the code, we first need to generate databases that contain all the data required to run queries that we can analyse after [6]. After creating the databases with the command "database create", we can then analyse them with the command "database analyze", passing the queries/query pack we wish to run on the database. The analysis command will then output the results of the queries in either Comma-separated values (CSV), Static Analysis Results Interchange Format (SARIF), or graph formats.

Taking advantage of CodeQL's query sharing approach, we used various queries from the bundle included by default on the CLI tool. The bundle organizes these queries under various types and content tags to categorize the type of problem they tackle. In our case, we were interested in all queries with the "security" tag, as these queries have labels with specific CWEs, and their purpose is to find them.

### 4.3.4 Clava

We used the latest Clava version (as of 2022/09/01). Two colleagues that worked on the same project developed the queries we used in the scanning process[40, 43]. Both works had different approaches and focused on different CWEs. From the first one, we have an approach based on ASTs that looks to identify occurrences of CWE-119 [9] and CWE-415 [14], and from the second one, an approach based on CFGs that searches for CWE-457 [15]. There were other queries developed from these works for CWE-401 [13] and CWE-120 [10], but they were not used due to a combination of some implementation issues and not fitting well on our dataset.

## 4.4 Developed Tool

As the final product of this research, we created a vulnerability scanning tool, Cppscanner, that effectively combines TranslationUnitPatcher and Clava, running the code analysis methods we specified. We did not use this tool in the first stage of the experiments, as it would not be wise to patch the code twice (once for the other analysers and another running Cppscanner). Especially because TranslationUnitPatcher takes significant time to run for the number of files we were processing. However, we did verify the results of the experiments obtained with the two programs separately, and they remained the same. Therefore, this program is a prototype that attempts to answer the question we initially proposed at the start of the research: "Is it possible to parse C/C++ code using a standard parser without requiring its dependencies successfully?".

Initially, we also wanted to explore the possibility of applying automatic vulnerability mitigation fixes. Due to time constraints, we did not attempt to implement this, but it is possible by taking advantage of Clava's code transformation capabilities. Although we would need to ensure the code's function does not change with these changes, at least for some CWEs, the changes needed would be minimal. As an example, for CWE-120 [10] (Buffer Copy without Checking Size of Input), we could change out the use of dangerous functions that do not check for the size of the buffers before copying data onto them for functions that do check the size (and try to infer

which size we would specify on the check). Of course, there are situations where this could break unexpectedly, but this is a line of investigation better suited for future work on this project.

## 4.5 Statistics Gathering

Gathering statistics about the vulnerabilities found in scans was done through every program outputting information in .txt, .csv, or .xml files. Through the combination of information contained in these files, the dataset labels, and counting time/files scanned, we were able to note down the following:

- Unique types of CWEs found, and their respective number of hits
- True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN)
- Files successfully analysed (differences here due to TranslationUnitPatcher, and queries)
- Number of files present with each labeled vulnerability
- Execution time

We consider all these statistics for evaluating the potential of our developed solution and the upsides/downsides of each analyser, shown and interpreted in detail in sections 5.3 and 5.4.

# Chapter 5

# Experimental Evaluation

## 5.1 Experimental Setup

To evaluate the value of our proposed solution, we had to establish comparisons to various other analysers, but this is a delicate process because many variables can affect the choice of an analyser, depending on the type of project requiring analysis and the prioritization of various statistics. Some of these variables we can infer from the data we will gather, described in section 4.5. There are many points of interest, such as:

- Evaluation measures we can calculate from hits (such as accuracy, precision, and recall)
- Scanning time
- Ease of setup
- Types of files/projects it can handle
- Range of vulnerabilities covered

These are all critical things to consider and can have different priorities depending on user needs. For example, large-scale projects will surely value scanning time heavily while likely not being worried about the ease of setup (as it will be a one-time thing). Of course, the ideal tool would be perfect in all of these aspects, but in many cases, to improve on one aspect requires sacrificing another. If we wish our solution to have a very high recall, that is, get as many TPs as possible, it is somewhat inherent that its accuracy will suffer. A more lenient hit trigger will also mean many more FPs. Therefore, there is room for all analysers to be viable solutions; perhaps the best approach is to combine some of them to cover one's weaknesses with another's strengths. With these experiments, we intend to study if users can consider our solution in the selection of these solutions.

In this section, we detail the various experiments we ran on each tool to gather as much data as possible from our dataset and maximize the number of comparisons we can make between each experiment. More comparisons will help increase the number of relevant conclusions we can arrive at from looking at the data.

### 5.1.1 Cppcheck

We tested three groups of source files with Cppcheck: the entire unpatched dataset (all 127476 files), the group of successfully patched files according to TranslationUnitPatcher's report (61191 files), and the same group of successfully patched files, but in their unpatched version, which we will refer to as the "parsable" group (61191 files). Despite patching not being necessary to use Cppcheck, we wanted to see what impact it would have on the analyser's results.

### 5.1.2 Flawfinder

Similarly to Cppcheck, we tested the same three groups of files: unpatched dataset, successfully patched files, and the parsable files. However, we had a second run on the successfully patched group, using the other configuration established in section 4.3.2, which we will refer to as the restricted group, as it is restricted in the number of hits it reports. As previously said, Flawfinder's approach, by default, is to try and find as many vulnerabilities as possible regardless of FPs (3.4). We wanted to see if we could have more similar results to other programs if we restricted the number of hits using Flawfinder's options by only triggering a hit on risk level 2+ and using the `-falsepositive` flag. The `-falsepositive` flag does not include hits that are likely to be false positives but, in turn, can potentially miss some important hits. The other three runs use the default settings.

### 5.1.3 CodeQL

The database creation process had a few problems. Notably, out of the 61191 files we had successfully patched, we could only include 59375 of them (about 97%). After some investigation, we established two issues that cause this. First, to verify the successfully patched files, we used a simple run of Clava, checking if the files were parsable without errors. Due to potential differences in compilers from CodeQL and Clava, we could have had a few files successfully parsed under Clava but not under CodeQL. Secondly, our approach of splitting TranslationUnitPatcher's execution under two different operating systems (explained in section 4.2) meant that any file with branches with different definitions for different operating systems was patched for that operating system that the tool was running under. For example, if we patch a file with these branches under Windows, it will generate a program that will only run on Windows and not Linux. The solution would be to ensure the patching process would all occur on the same operating system, but again this was not feasible due to the long runtime. We decided to create CodeQL databases under Linux, as Linux's patching operation was faster than Windows', so we ended up with a larger percentage of files parsed under Linux.

The next step after database creation is choosing the queries to run. We created a pack that includes all queries with the "security" and specific CWE tags to associate every hit with a CWE. After removing a few queries that had bugs stopping the program on the last execution step, we ended up with 133 queries on this original pack.

However, we encountered some problems and could not run this pack successfully on our database containing all the files. Since CodeQL extracts relational data from code to create codebases, when using a vast number of files, the complexity of the codebase increases tremendously. While trying to run the pack, we observed that a group of simple queries finished fast, around the three-minute mark, and another group of queries finished after them, around the 51-minute mark. The remaining queries, 49 of them, were left to run overnight for 8 hours, and none of them finished.

Trying to figure out the answer as to why these queries would not run, we did some experiments with two smaller databases. As a reminder, the original database contained all the successfully parsed dataset files and had 59375 functions. We then created two additional databases. One of them, "vulnerable", contained all parsable files labeled with CWE-119, CWE-120, CWE-469, and CWE-476 vulnerabilities (2821 functions). The other, "vulnerable-other", contained the same files as the previous, with the addition of the files labeled with "CWE-other" vulnerabilities (3884 functions).

This time, scanning the "vulnerable-other" database using the original query pack with 133 queries proved successful. However, it still took a significant amount of time considering the database size (2h43m13s). After running the same experiment on the "vulnerable" database, the runtime was much shorter (1h10m45s). From the difference in time observed in those two experiments, we can conclude that the time taken to run queries is not directly proportional to the amount of code in the database. This time difference is presumably due to the relational nature of the database and to spending more time searching through more functions than in smaller databases.

The "vulnerable-other" database had 37.7% more files than the "vulnerable" database. However, it took 130.6% more time to finish, confirming a somewhat exponential growth in time based on the database size. By applying the same logic to the database with all the patched files, we can see that the time to finish the scan would amount to days, if not weeks, so it was not feasible to include the big time-consuming queries on that database's scanning process. Instead, we created a separate pack that would run on the original database by removing the troublesome queries, this time only with 84 queries, removing the 49 queries causing the time problem.

Since all the functions in this dataset are isolated and do not depend on one another, we could employ a potential solution to solve this time issue - split the entire database into various small databases and run the queries on every database. However, this would still require much time and effort, especially as the CodeQL engine seems to take a fixed amount of time (around 4-5 minutes) to start up every time we run it. It also feels like a solution that does not take care of the original problem but instead tries to circumvent it. CodeQL also does not provide any way to automate this case, so we would either have to do it manually or spend extra time figuring out a way to automate this process. Ultimately, all this confirms that CodeQL is not very desirable to use in the scenario we wish to provide an answer for with our proposed solution, which is a large number of source files that we want to analyse independently. The problem is that it cannot process the fact that they are isolated, leading to spending a ton of time considering unnecessary relations between the

isolated functions.

### 5.1.4 Clava

Since Clava's scanning method was time-consuming, we split the workload on two computers, similar to the patching process (4.2). However, this time, only two versions ran under Windows to prevent a repeat of the different operating systems problem. As was seen in section 4.3.3, we expected the number of files that we could include to be lower than the original successfully patched group due to the operating system differences, but we ran the tool under Windows this time to see the difference compared to CodeQL running under Linux. While CodeQL could include 59375 files, Clava managed to include 59996. Whether this is because there were more files parsable under Windows compared to Linux, or if it was due to differences in the compilers of the two tools (or possibly a mix of both), is uncertain. We would need to perform the same test with Clava under Linux to verify this.

As for the method used in Clava to scan through the files, we tested two different approaches. As we are only testing two queries, the main bottleneck of the scanning process would be the parsing of the files, so we attempted to speed up this process. The first approach launched the engine for each function file one by one. In contrast, the other approach took 5000 function files simultaneously and proceeded to parse them, taking better advantage of multiple threads. The difference in time was significant; the first approach took roughly double the time of the second one (5h2m compared to 2h29m). However, the second approach has a potential issue: it treats all the functions as a single codebase (when they are entirely separate, and the scan should treat them as such). The main problem that arises from this is function calls, especially when dealing with files with main() functions. While scanning, the program kept throwing warnings of having multiple identical function definitions. It is worth noting that the nature of the queries we used did not affect the results, but this could be affected by future queries. This approach also does not adhere exactly to what we want to achieve, as the tool can end up parsing a file with its dependencies if the simultaneous scan includes both files. We can work with this "multiple files" approach for our testing scenario, but if we wish to expand the tool with more queries and be more strict with the independence requirement, this issue needs to be one of the first things addressed.

## 5.2 CWEs Encountered

After processing the results each tool made with each group of files, we compiled all the CWEs found and came up with the following numbers, separated by tool and group of files used:

- Cppcheck
  - **Cpp#1** (patched, 61191 functions) - 6728 hits, 34 unique CWEs
  - **Cpp#2** (unpatched, 127476 functions) - 23559 hits, 35 unique CWEs
  - **Cpp#3** (parsable, 61191 functions) - 11322 hits, 32 unique CWEs

- Flawfinder
  - **Flaw#1** (patched, 61191 functions) - 32260 hits, 15 unique CWEs
  - **Flaw#2** (unpatched, 127476 functions) - 37730 hits, 15 unique CWEs
  - **Flaw#3** (parsable, 61191 functions) - 18548 hits, 15 unique CWEs
- CodeQL
  - **CodeQL#1** ("original" dataset, 59375 functions) - 12146 hits, 30 unique CWEs
  - **CodeQL#2** ("vulnerable-other" dataset, 3884 functions) - 2403 hits, 35 unique CWEs
- **Clava** - 6154 hits, 3 unique CWEs

In table 5.1, we have the total number of vulnerabilities each tool could detect for all experiments, except for Flawfinder's restricted group, since we can better see the difference between those results and the patched group's results in table 5.3. Any vulnerabilities that did not reach a minimum of 40 hits with a single tool are not present on the table to prevent too many entries and point our focus to the ones found more often. Clava hit only 3 unique CWEs, so we did not include its results in the table, as most of its entries would be left blank. Clava had 3221 hits for CWE-119, 552 hits for CWE-457, and 2381 hits for CWE-415.

## 5.3 Statistical Comparison

By combining all the data obtained from the analysers' generated report files and the labels from the original dataset, we can generate statistics about our hits - which were real vulnerabilities, which were not, and which the analysers missed. The following tables provide information about TPs, FPs, FNs, TNs, and three evaluation measures calculated from them: accuracy, precision, and recall, as explained in2.6.

In table 5.2, we have data about the three Cppcheck scans, in table 5.3, we have data about the four Flawfinder scans, in table 5.4, we have data about the two CodeQL scans, and finally, in table 5.5, we have data about the Clava scan.

| CWE | Cpp #1 | Cpp #2 | Cpp #3 | Flaw #1 | Flaw #2 | Flaw #3 | CodeQL #1 | CodeQL #2 |
|---|---|---|---|---|---|---|---|---|
| 20 | - | - | - | 2318 | 2600 | 1226 | - | - |
| 78 | - | - | - | 321 | 364 | 197 | - | - |
| 119 | 62 | 127 | 62 | 3609 | 7324 | 3609 | 37 | 9 |
| 120 | - | - | - | 10933 | 13805 | 6549 | 171 | 91 |
| 121 | - | - | - | - | - | - | 900 | 49 |
| 126 | - | - | - | 6428 | 7697 | 3958 | - | - |
| 134 | - | - | - | 4362 | 1013 | 460 | - | - |
| 190 | 5 | - | - | 1113 | 1246 | 692 | 174 | 73 |
| 327 | - | - | - | 230 | 309 | 150 | 10 | 4 |
| 362 | - | - | - | 2117 | 2436 | 1196 | - | - |
| 377 | - | - | - | 85 | 137 | 64 | 1 | 1 |
| 398 | 227 | 586 | 162 | - | - | - | - | - |
| 401 | 79 | 133 | 77 | - | - | - | 135 | 31 |
| 456 | - | - | - | - | - | - | - | 1011 |
| 457 | 863 | 956 | 202 | - | - | - | - | 71 |
| 467 | 57 | 26 | 12 | - | - | - | 15 | 13 |
| 468 | - | - | - | - | - | - | 92 | 11 |
| 476 | 1403 | 493 | 144 | - | - | - | 9271 | 740 |
| 587 | - | 282 | 1 | - | - | - | - | - |
| 595 | 54 | 866 | 133 | - | - | - | - | - |
| 664 | 2 | 123 | 27 | - | - | - | - | - |
| 665 | - | - | - | - | - | - | 797 | 76 |
| 676 | - | - | - | 113 | 102 | 63 | 203 | 41 |
| 686 | 3413 | 376 | 194 | - | - | - | - | - |
| 732 | - | - | - | 76 | 110 | 48 | 186 | 101 |
| 758 | 414 | 19249 | 10161 | - | - | - | - | - |
| 807 | - | - | - | 541 | 573 | 326 | - | - |

Table 5.1: Total CWEs detected per experiment

| Experiment | CWE | TP | FP | FN | TN | Accuracy | Precision | Recall | Time |
|---|---|---|---|---|---|---|---|---|---|
| Patched | 119/120 | 24 | 28 | 2504 | 58635 | 95.86% | 46.15% | 0.95% | 10m19s |
| | 469/476 | 44 | 575 | 415 | 60157 | 98.38% | 7.11% | 9.59% | |
| | other | 286 | 2559 | 1532 | 56814 | 93.31% | 10.05% | 15.73% | |
| Unpatched | 119/120 | 50 | 55 | 4701 | 122670 | 96.27% | 47.62% | 1.05% | 14m41s |
| | 469/476 | 121 | 236 | 1324 | 125795 | 98.78% | 33.89% | 8.37% | |
| | other | 659 | 20155 | 2920 | 103782 | 81.93% | 3.17% | 18.41% | |
| Parsable | 119/120 | 24 | 81 | 2504 | 58582 | 95.78% | 22.86% | 0.95% | 7m33s |
| | 469/476 | 22 | 335 | 437 | 60397 | 98.74% | 6.16% | 4.80% | |
| | other | 361 | 20413 | 1457 | 38960 | 64.56% | 1.74% | 19.86% | |

Table 5.2: Data from the Cppcheck scans

| Experiment | CWE | TP | FP | FN | TN | Accuracy | Precision | Recall | Time |
|---|---|---|---|---|---|---|---|---|---|
| Patched | 119/120 | 2155 | 3650 | 373 | 55013 | 93.43% | 37.12% | 82.25% | 40.57s |
| | 469/476 | 0 | 0 | 459 | 60732 | 99.25% | n/a | n/a | |
| | other | 1212 | 6402 | 606 | 52971 | 88.55% | 15.92% | 66.67% | |
| Unpatched | 119/120 | 4037 | 7456 | 714 | 115269 | 93.59% | 35.13% | 84.97% | 37.12s |
| | 469/476 | 0 | 0 | 1445 | 126031 | 98.87% | n/a | n/a | |
| | other | 2037 | 7182 | 1542 | 116715 | 93.16% | 22.10% | 56.92% | |
| Parsable | 119/120 | 2136 | 3415 | 392 | 55248 | 93.78% | 38.48% | 84.49% | 23.17s |
| | 469/476 | 0 | 0 | 459 | 60732 | 99.25% | n/a | n/a | |
| | other | 1149 | 3627 | 669 | 55746 | 92.98% | 24.06% | 63.20% | |
| Restricted | 119/120 | 1177 | 2166 | 1351 | 56547 | 94.33% | 35.21% | 46.56% | 46.49s |
| | 469/476 | 0 | 0 | 459 | 60732 | 99.25% | n/a | n/a | |
| | other | 976 | 4201 | 842 | 55172 | 91.76% | 18.85% | 53.69% | |

Table 5.3: Data from the Flawfinder scans

| Dataset | CWE | TP | FP | FN | TN | Accuracy | Precision | Recall | Time |
|---|---|---|---|---|---|---|---|---|---|
| original | 119/120 | 41 | 82 | 2435 | 56817 | 95.76% | 33.33% | 1.66% | 45m37s |
| | 469/476 | 104 | 6534 | 334 | 52393 | 88.42% | 1.57% | 23.74% | |
| | other | 224 | 1729 | 1541 | 55881 | 94.49% | 11.47% | 12.69% | |
| vulnerable-other | 119/120 | 47 | 8 | 2429 | 1400 | 37.26% | 85.45% | 1.90% | 2h44m57s |
| | 469/476 | 101 | 409 | 347 | 3027 | 80.54% | 19.80% | 22.54% | |
| | other | 508 | 321 | 1257 | 1798 | 59.37% | 61.28% | 28.78% | |

Table 5.4: Data from the CodeQL scans

| CWE | TP | FP | FN | TN | Accuracy | Precision | Recall | Time |
|---|---|---|---|---|---|---|---|---|
| 119/120 | 782 | 2439 | 1716 | 55041 | 93.07% | 24.28% | 31.31% | 2h29m |
| 469/476 | 0 | 0 | 453 | 59525 | 99.21% | n/a | n/a | |
| other | 239 | 2183 | 1541 | 56015 | 92.94% | 8.15% | 13.43% | |

Table 5.5: Data from the Clava scan

## 5.4 Results Interpretation

### 5.4.1 CWE List

Starting with the list of found CWEs, we will document here some conclusions we can arrive at from interpreting the data.

When it comes to Cppcheck, it was the best tool in terms of CWEs covered, covering between 34-35 unique CWEs. CodeQL managed to cover 35 unique CWEs when using all queries, but this was only possible on the smaller dataset - the original dataset could only cover 30. Cppcheck using the default configuration on the successfully patched group had the lowest amount of hits, 6728, and something that makes sense given the tool prioritizes having low false positives.

There is a significant difference in results between unpatched and patched files when it comes to Cppcheck, and it mostly comes down to CWE-686 [18] and CWE-758 [19]. CWE-686 has 3413 hits on the patched group but 184 hits on the unpatched version of that group. CWE-758 has 414 hits on the patched group but 10161 hits on the unpatched version of that group. There are justifiable reasons for this - one is inherent to looking at unpatched code, and the other hints at a problem with TranslationUnitPatcher. CWE-758 is "Reliance on Undefined/Unspecified/Implementation-Defined Behavior", and the number of hits increases massively when unpatched. This increase makes sense, as all the function calls are missing their definitions on dependencies, so the scanner correctly calls this out. Meanwhile, CWE-686 is "Function Call with Incorrect Argument Type", and it increases massively in hits when patched - which can only mean the patching process for creating function definitions is not always working correctly.

The unique point of Cppcheck on this dataset is its analysis of CWE-476, one of the CWEs focused by the dataset. On average, it had 680 hits across all three runs, while Clava and Flawfinder failed to find this CWE, and CodeQL has a massive outlier due to one of its queries triggering heavily (9271 hits on the original database). This outlier could be another symptom of a patcher error, but we could not confirm this.

When it comes to Flawfinder, it is the most consistent and narrow tool in terms of the range of CWEs covered, always covering 15 unique CWEs regardless of the experiment. Meanwhile, all the other tools cover between 30-35, depending on the tool and the group of files scanned. Flawfinder is also by far the tool with the most hits, with an average of 29513 hits across the three initial runs - we expected this as that is the main selling point of the tool. It reports the slightest possible risks by default when looking for vulnerabilities, even if it raises a lot of false positives.

Going deeper into the actual CWE numbers, we see Flawfinder is by far the best at detecting CWE-119/120, the two vulnerabilities based on buffers on the dataset. On the three initial runs, it has an average of 15276 hits on these two vulnerabilities, while Cppcheck has 84, CodeQL has 131, and Clava has 3221. However, when it comes to CWE-469/476, the other two vulnerabilities the dataset focuses on, Flawfinder, could not find them. Apart from these dataset CWEs, the others worth pointing out are CWE-126 [11], "Buffer Over-read" and CWE-362 [12], "Concurrent Execution using Shared Resource with Improper Synchronization (’Race Condition’)". Flawfinder

found a considerable amount of these (6028 and 1936 on average, respectively), while none of the other tools covered these.

CodeQL has nothing particularly interesting to mention that we did not mention before, apart from the outlier on CWE-476. If we remove that outlier, it was by far the tool with the lowest amount of hits; the scan on the original database has a total of 2875 hits, far from all the other tools (as a reminder, Cppcheck's patched group run had 6728 hits, and Flawfinder's same run had 32260 hits).

As for Clava, since we only equipped it to search for CWE-119, CWE-457, and CWE-415, it has the lowest amount of unique CWEs encountered. Naturally, due to the combination of limited time and inherent complexity to query design, Clava will fall behind in terms of CWEs covered as there are only three queries developed. In comparison, Cppcheck and Flawfinder have several test cases, and CodeQL has an established query repository (using 133 queries in our case). Despite that, it managed to get 3221 hits for CWE-119, one of the main CWEs we wanted to focus on for our scanning, severely outperforming both Cppcheck (62 hits) and CodeQL (37 hits) for this CWE.

Finally, there are some conclusions we can take about the patching process. In the case of CodeQL and Clava, patching allowed these tools to get any hits; otherwise, they would have 0. In the case of Cppcheck and Flawfinder, even though they technically did not need patching to analyse the code, we wanted to see if it would make a difference.

It definitely made a difference. However, it was both positive and negative, and in some cases, hard to tell which. As mentioned earlier, patching the code massively lowered the number of false positives of Cppcheck hits on CWE-758. This difference is a positive impact, as it prevents time investigating these results that are false alarms. However, again with Cppcheck, when looking at CWE-686, we can see a case of apparent negative impact, as it seems that the patching process was introducing this vulnerability (perhaps as a false positive). Another possible negative impact we found was on CWE-134 under Flawfinder, where it went from 4362 on the patched group to 460 on the parsable group.

Then there are the cases where we cannot be sure if the patching helped find more vulnerabilities or introduced them without extensive manual review, something we did not have time to perform. Some examples comparing the patched group to the parsable group on Flawfinder: CWE-20 went from 2318 to 1226; CWE-126 went from 6428 to 3958; CWE-190 went from 1113 to 692.

### 5.4.2 Statistics

Looking at the performance metrics, we can see Flawfinder had the highest recall (excluding CWE-469/476, which it is not equipped for), likely due to its focus on low false negatives. Changing between the three groups did not seem to do much for Flawfinder, mostly keeping its results the same. The only relevant change was that the patched group had double the false positives under the CWE-other label compared to the parsable group, possibly due to the patcher introducing errors.

As for Cppcheck, it shows its precision to be better than Flawfinder for finding the buffer-based vulnerabilities (although with a much, much lower recall), and as shown previously, it is the tool with the best combination of performance metrics for finding CWE-469/476. CodeQL has a higher recall on CWE-476, but this comes at significantly lower precision, to the point that there is the possibility that the hits happen by sheer coincidence instead of finding the proper vulnerability patterns. Cppcheck also shows very low precision on the CWE-other label for both patched and unpatched groups. However, we believe this to be primarily due to the patcher errors: on the unpatched group, it is filled with "false positives" of CWE-758, and on the patched group, there is a high chance it can be due to CWE-686 (as discussed at the end of section 5.4.1).

When we came up with a solution to the missing 49 queries under the CodeQL experimental setups (5.1.3), we thought the new smaller database would have significantly better results due to all the extra queries it would have. Unfortunately, although it did show a pretty good improvement under the CWE-other label (the main conclusion to note being having more than double the recall), it barely showed any improvement on the buffer-based vulnerabilities. Somehow, it managed to get worse results for CWE-476.

When it comes to Clava, we consider it to have been a success when it comes to buffer-based vulnerabilities. Clava managed to have a much higher recall than CodeQL and Cppcheck for this, and although it does not reach Flawfinder's results, there is something vital to consider: Clava was only checking for CWE-119 and not CWE-120. We grouped CWE-119 and CWE-120 under these statistics, so we could not analyse them separately, but looking at Flawfinder's hits in table 5.1, CWE-120 had between double to triple the hits of CWE-119. Assuming the dataset does not have a very disproportionate amount of CWE-119 compared to CWE-120, if we were to implement a similarly successful query for CWE-120 under Clava, we believe it could reach similar results to Flawfinder's on this aspect.

Finally, our observations speculate that we should take all results under the "CWE-other" label with a grain of salt. We speculate this primarily because since the dataset mainly focuses on CWE-119/120/469/476, it is reasonable to assume that those vulnerabilities had a larger focus when the dataset was built. There are over a thousand CWE entries, and many were likely skipped when labeling for other CWEs. The dataset specified that they were labeled using static analysis tools but did not say which ones. However, if they used tools that focused only on those four vulnerabilities (similar to how Flawfinder narrowed down to a few CWEs compared to the wide range of Cppcheck/CodeQL), it is likely to reach this state.

# Chapter 6

# Conclusions and Future Work

As we have established various times during this document, maintaining security is something we believe is crucial in any modern software application. There is too much risk in leaving vulnerabilities on programs, especially when we keep seeing frequent cyberattacks on many services and the impact they can have [37, 34].

C/C++ is a particularly delicate language and hard to analyse on a large scale, which many approaches cannot do effectively. It is relevant, though, because C/C++ programs are widespread and many computer systems rely on this language due to its features. Analysing the data from our experiments, we conclude that the proposed approach can contribute to solving this problem. It successfully analyzed source code after parsing, using a conventional C/C++ parser, for code that was missing its dependencies, one of the goals we sought to achieve. It could also do it in a reasonable time, considering the computational effort of patching and parsing the vast amounts of source code we tested. We also successfully used patching to allow other existing tools (CodeQL) to analyse code that they otherwise would not be able to (due to missing dependencies).

We also achieved another one of our goals: to maintain the easy extensibility of the previous vulnerability detection approach. By creating a new program that combines the patching from TranslationUnitPatcher, and the scanning from Clava, we have streamlined the developing process significantly. Anyone that wants to work on this project in the future only needs to worry about improving either of the two steps in the process: increasing the number of errors that the patcher can effectively handle; or developing new JavaScript-based queries for Clava's scanning, catching more vulnerabilities.

Unfortunately, we could not meet our other objective - achieving good metrics on vulnerability scanning. Ultimately, we do not believe our results qualify for this. Our tool's most successful query searches for CWE-119 occurrences; in retrospect, it does not do a bad job at this. However, other tools still outperform it on this particular CWE, and the methods we developed to search for other CWEs are incomplete or are not currently working.

Another objective we initially considered but did not investigate for our work was the possibility of using Clava's code transformation features to apply vulnerability fixes automatically. Due to time restraints, this was not part of our proposed approach. However, it is a strong point

of interest, with the ability to provide another novelty to our approach and the potential to further contribute to security improvement if explored successfully.

With that said, we believe there is a demand and space for this approach. We expect it to be easy to further research and develop due to the wide usage of the C language and its several static analysis tools; there is plenty of research and tools to further our knowledge by studying. Our solution still aims to be novel, but we laid the groundwork for it to succeed using existing work.

Regarding future work, the nature of software always opens the possibility to extend it for new functions. However, thinking only of the remaining issues, we believe starting with investigating the points in the following list is the best approach:

- **TranslationUnitPatcher**
  - Add more compiler error handling; when all compiler error codes have a proper answer, the percentage of successfully patchable source code should increase drastically.
  - Investigate if the current error handling methods are introducing unwanted changes to the code to ensure it does not interfere with scanning results.
  - Fix the current bug that prevents patching in multi-threaded mode: fixing this is incredibly important so that analysis of large datasets does not take longer than a day.
- **Clava**
  - Add more queries searching for vulnerability patterns to cover a wider range of CWEs.
  - Improve existing queries to achieve better metrics.
  - Investigate the possibility of automatically applying vulnerability fixes
- **Cppscanner**
  - The current version of the tool is a very rough prototype and requires some cleanup of various aspects of the program for it to be ready for general use. Some important points would be adding the possibility of splitting the patching/scanning process and adding the ability to more easily customize Clava queries (currently, it is always set to run the queries we have developed, with no ability to change this)

# References


[1] Abstract syntax tree - wikipedia. Available at https://en.wikipedia.org/wiki/Abstract_syntax_tree, September 2022.

[2] Cern computer security information. CERN. Available at https://security.web.cern.ch/recommendations/en/codetools/rats.shtml, February 2022.

[3] clava/translationunitpatcher at master · specs-feup/clava. GitHub. Available at https://github.com/specs-feup/clava/tree/master/TranslationUnitPatcher, September 2022.

[4] Code property graph documentation. ShiftLeft. Available at https://docs.shiftleft.io/core-concepts/code-property-graph, February 2022.

[5] Codeql. GitHub, Inc. Available at https://codeql.github.com/, September 2022.

[6] Codeql cli – codeql. GitHub, Inc. Available at https://codeql.github.com/docs/codeql-cli/, September 2022.

[7] Common weakness enumeration. CWE. Available at https://cwe.mitre.org/, February 2022.

[8] Cppcheck - a tool for static c/c++ code analysis. Cppcheck. Available at https://cppcheck.sourceforge.io/, February 2022.

[9] Cwe - cwe-119: Improper restriction of operations within the bounds of a memory buffer (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/119.html, September 2022.

[10] Cwe - cwe-120: Buffer copy without checking size of input ('classic buffer overflow') (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/120.html, September 2022.

[11] Cwe - cwe-126: Buffer over-read (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/126.html, September 2022.

[12] Cwe - cwe-362: Concurrent execution using shared resource with improper synchronization ('race condition') (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/362.html, September 2022.

[13] Cwe - cwe-401: Missing release of memory after effective lifetime (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/401.html, September 2022.

[14] Cwe - cwe-415: Double free (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/415.html, September 2022.

[15] Cwe - cwe-457: Use of uninitialized variable (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/457.html, September 2022.

[16] Cwe - cwe-469: Use of pointer subtraction to determine size (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/469.html, September 2022.

[17] Cwe - cwe-476: Null pointer dereference (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/476.html, September 2022.

[18] Cwe - cwe-686: Function call with incorrect argument type (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/686.html, September 2022.

[19] Cwe - cwe-758: Reliance on undefined, unspecified, or implementation-defined behavior (4.8). MITRE. Available at https://cwe.mitre.org/data/definitions/758.html, September 2022.

[20] Cwe-119: Improper restriction of operations within the bounds of a memory buffer. CWE. Available at https://cwe.mitre.org/data/definitions/119.html, February 2022.

[21] Cwe-415: Double free. CWE. Available at https://cwe.mitre.org/data/definitions/415.html, February 2022.

[22] Cwe-825: Expired pointer dereference. CWE. Available at https://cwe.mitre.org/data/definitions/825.html, February 2022.

[23] Draper vdisc dataset - vulnerability detection in source code. OSF. Available at https://osf.io/d45bw/, February 2022.

[24] Flawfinder home page. Flawfinder. Available at https://dwheeler.com/flawfinder/, February 2022.

[25] Hdf5 for python. Available at https://www.h5py.org/, September 2022.

[26] The hdf5® library & file format - the hdf group. The HDF Group. Available at https://www.hdfgroup.org/solutions/hdf5/, September 2022.

[27] Joern - the bug hunter's workbench. Joern. Available at https://joern.io/, February 2022.

[28] Joern documentation. Joern. Available at https://docs.joern.io/home/, February 2022.

[29] Juliet c/c++ 1.3 - nist software assurance reference dataset. NIST. Available at https://samate.nist.gov/SARD/test-suites/112, September 2022.

[30] Top programming languages. IEEE Spectrum. Available at https://spectrum.ieee.org/top-programming-languages, February 2022.

[31] Periklis Akritidis et al. Cling: A memory allocator to mitigate dangling pointers. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[32] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[33] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, 1998.

[34] Andreea Bendovschi. Cyber-attacks–trends, patterns and security countermeasures. *Procedia Economics and Finance*, 28:24–31, 2015.

[35] João Bispo and João M.P. Cardoso. Clava: C/c++ source-to-source compilation using lara. *SoftwareX*, 12:100565, 2020.

[36] Pedro Carvalho, Nuno Oliveira, and Pedro Rangel Henriques. Unfuzzying fuzzy parsing. 2014.

[37] Brian Cashell, William D Jackson, Mark Jickling, and Baird Webel. The economic impact of cyber-attacks. *Congressional research service documents, CRS RL32331 (Washington DC)*, 2, 2004.

[38] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery.

[39] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129 vol.2, 2000.

[40] Thomas Devoulon. Selection and analysis of c/c++ features for identifying the presence of code vulnerabilities, April 2022.

[41] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[42] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 313–326, New York, NY, USA, 2005. Association for Computing Machinery.

[43] Jules Hervault. Selection and analysis of c/c++ code characteristics for vulnerabilities identification, April 2021.

[44] Hannes Holm, Teodor Sommestad, Ulrik Franke, and Mathias Ekstedt. Success rate of remote code execution attacks: expert assessments and observations. *Journal of universal computer science (Online)*, 18(6):732–749, 2012.

[45] Arvinder Kaur and Ruchikaa Nayyar. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[46] Rainer Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1997.

[47] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[48] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. A machine learning approach to detection of javascript-based attacks using ast features and paragraph vectors. *Applied Soft Computing*, 84:105721, 2019.

[49] Richard M Reese. *Understanding and Using C Pointers: Core Techniques for Memory Management*. " O'Reilly Media, Inc.", 2013.

[50] Robert C Seacord. *Secure Coding in C and C++*. Pearson Education, 2005.

[51] J. Viega, J.T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267, 2000.

[52] W. Joosen Y. Younan and F. Piessens. Code injection in c and c++ : A survey of vulnerabilities and countermeasures. Technical report, Department of Computer Science, K.U.Leuven, July 2004.

[53] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.