UNIVERSITY OF TARTU
Institute of Computer Science
Cyber Security Curriculum

Roman Amjaga

Static Analysis to Detect Memory Corruption Vulnerabilities

Master's Thesis (21 EAP)

Supervisor:
Vesal Vojdani, PhD

Tartu 2025

Static Analysis to Detect Memory Corruption Vulnerabilities

Abstract:

Memory corruption attacks have existed for a long time, and despite that, they are still considered a major threat to modern software. In fact, memory safety is such a major problem that in 2023, the U.S. Cybersecurity and Infrastructure Security Agency [1] and in the year 2024, the Office of the National Cyber Director [2] released articles, addressing the need for memory safety in modern and future software. The most widespread solution to memory safety problems is the use of memory-safe programming languages. In addition to not solving the problem completely, such an approach also does not take into consideration all the software that is written using non-memory-safe programming languages. Due to different constraints, it is often not realistic to rewrite a whole application to another programming language. The need for code written in a non-memory-safe programming language to be secure has several solutions with their advantages and downsides. This paper focuses on one such solution, that is, static code analysis. Static code analysis inspects the code without executing it and can detect a wide range of vulnerabilities. This paper contributed to the field by examining the cause of modern memory corruption bugs in the code. During the analysis, modern static code analyzers were tested to determine whether static code analysis is an effective measure against memory corruption vulnerabilities. In addition, a test suite of simplified real-world vulnerabilities was created for further refinement of static code analysis tools.

This thesis is written in English and is 47 pages long, including 6 chapters, 9 figures and 19 tables.

Keywords: Memory corruption, static code analysis, test suite
CERCS: T120 – Systems engineering, computer technology

Staatiline analüüs mälu rikkumise haavatavuste tuvastamiseks

Lühikokkuvõte:

Mälu korruptsiooni rünnakud on pikaaegne probleem ning vaatamata sellele, on see endiselt kaasaegse tarkvara jaoks tõsine ohutegur. Mälu turvalisus on laialdane probleem, mistõttu 2023. aastal avaldas Ameerika küberkaitseagentuur [1] ja 2024. aastal Ameerika Valge Maja Riikliku Küberturvalisuse Direktorite Büroo [2] artiklid, milles käsitleti vajadust mälu turvalisuse tagamiseks kaasaegses ja tuleviku tarkvaras. Kõige levinum mälu turvalisuse probleemide lahendus on mälu turvaliste programmeerimiskeelte kasutamine. Antud viis aga ei lahenda probleemi täielikult ning lisaks ei võta selline lähenemine arvesse kogu tarkvara, mis on kirjutatud mitte mälu turvalistes programmeerimiskeeltes. Erinevate piirangute tõttu ei ole sageli realistlik kogu rakendust kirjutada ümber teise, ehk mälu turvalisse programmeerimiskeelde, vaid lisaks on aktuaalne ja oluline mitte mälu turvalises programmeerimiskeeles kirjutatud koodi turvalisus. Selle saavutamiseks on olemas mitmeid lahendusi, millel on oma eelised ja puudused. Käesolev töö keskendub ühele võimalikule lahendusele – staatilisele koodianalüüsile. Staatiline koodianalüüs uurib koodi seda käivitamata ning on võimeline tuvastama laias ulatuses haavatavusi. Käesolev töö annab panuse antud valdkonda, uurides kaasaegsete mälukorruptsiooni vigade põhjuseid koodis. Analüüsi käigus testiti kaasaegseid staatilisi koodianalüüsi tööriistu, et teha kindlaks, kas staatiline koodianalüüs on tõhus meede mälu korruptsioonist tulenevate haavatavuste vastu. Lisaks loodi testide komplekt lihtsustatud reaalsetest haavatavustest, et aidata staatilisi koodianalüüsi tööriistu edaspidiselt arendada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 47 leheküljel, 6 peatükki, 9 joonist ja 19 tabelit.

Võtmesõnad: Mälu korruptsiooni, statiline koodianalüüs, testide komplekt
CERCS: T120 - Süsteemitehnoloogia, arvutitehnoloogia

List of abbreviations and terms

AST	Abstract Syntax Tree
CFG	Control Flow Graph
CG	Call Graph
CPG	Code Property Graph
CTU	Cross Translation Unit
CV-COMP	International Competition on Software Verification
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DDG	Data Dependency Graph
DE	Diagnostic Efficiency
DFG	Data Flow Graph
FN	False Negative
FP	False Positives
FPR	False Positive Rate
JSON	JavaScript Object Notation
PDG	Program Dependence Graph
RGB	Red, Green, Blue
RQ	Research question
SDG	System Dependency Graph
SE	Symbolic Execution
SES	Symbolic Execution State
SET	Symbolic Execution Tree
TN	True Negative
TP	True Positives
TPR	True Positive Rate
UAV	Use-After-Free
URL	Uniform Resource Locator
VFG	Value Flow Graph
WSL	Windows Subsystem for Linux

Table of contents

1. Introduction ... 9
1.1 Problem Statement and Motivation ... 9
1.2 Scope ... 10
1.3 Limitations ... 11
1.4 Contributions ... 11
1.5 Novelty ... 12
1.6 Research Questions ... 14
1.7 Structure ... 15
1.8 Methodology ... 15
2. Systematic Literature Review ... 17
2.1 Literature Sources and Search Keywords ... 17
2.2 Inclusion and Exclusion Criteria ... 18
2.3 Data Extraction ... 19
2.4 Paper Selection ... 19
2.5 Excluded Papers ... 20
2.6 Overview of Fundamental Approaches Used by Reviewed Papers ... 20
2.6.1 Symbolic Execution ... 20
2.6.2 Source Code Graph Traversal ... 21
2.6.3 Program Model Checking ... 25
2.6.4 Code Matching ... 25
2.7 Conclusion ... 27
3. Analysis of Memory Corruption Vulnerabilities ... 33
3.1 CVEs Selection Criteria ... 33
3.2 Selected CVEs ... 34
3.3 Test Suite ... 36
3.3.1 International Competition on Software Verification (SV-COMP) ... 37
3.4 Patterns ... 38
3.4.1 Improper Bound Checking ... 38
3.4.2 Integer Overflow Leading to Buffer Overflow ... 39
3.5 Statistics ... 40
4. Static Code Analysis of CVEs ... 42
4.1 Tools from the Literature Review ... 42

4.2 Tools Utilized for Static Code Analysis ... 43
4.2.1 Tools Configuration ... 43
4.2.2 Tools Produced Diagnostics Classification ... 44
4.3 Real-world Projects Analysis ... 44
4.3.1 Results ... 45
5. Static Code Analysis of the Test Suite ... 48
5.1 Results ... 48
5.2 SV-COMP ... 51
5.2.1 Results ... 52
6. Conclusion ... 55
References ... 56
Appendix ... 64
I. Data Extraction Form ... 64
II. Systematic Literature Review Paper Summaries ... 65
III. Commands Used for Static Code Analyzers ... 72
IV. Reviewed Paper Tools Configuration ... 73
V. License ... 78

List of figures

Figure 1. Systematic literature review paper selection diagram 19
Figure 2. Abstract Syntax Tree of foo function [27] 22
Figure 3. Control Flow Graph of foo function [34] 23
Figure 4. Program Dependence Graph of foo function [34] 24
Figure 5. Code Property Graph of foo function [34] 24
Figure 6. Length of functions containing vulnerability 40
Figure 7. Length of files containing vulnerability 41
Figure 8. Percentage of successfully detected vulnerabilities relative to function length in real projects 47
Figure 9. Percentage of successfully detected vulnerabilities relative to function length in a test suite 51

List of tables

Table 1. Evaluation criteria per paper and technique used 27
Table 2. Self-reported precision ranking of reviewed approaches 29
Table 3. Not ranked research reasons 30
Table 4. Self-reported ranking of techniques for vulnerability detection based on precision .30
Table 5. Number of papers that find specific vulnerabilities 31
Table 6. Graph utilization 31
Table 7. Publicly accessible tools of reviewed approaches 32
Table 8. CVEs per category 35
Table 9. Analyzed vulnerability count by CWE category 35
Table 10. Test suite benchexec classification with test case count 37
Table 11. Reviewed tools configuration errors 42
Table 12. Tools used for vulnerability analysis 43
Table 13. Mapping of Cppcheck diagnostic 44
Table 14. Real-world projects analysis results 46
Table 15. Test suite analysis results 48
Table 16. Test suite analysis results 49
Table 17. SV-COMP test suite analysis tools 52
Table 18. SV-COMP test suite analysis results 53
Table 19. SV-COMP official competition verification results 53

1. Introduction

All modern computers operate using the Von Neumann architecture, which does not separate program instructions from program data. All the information a process uses is stored in random-access memory and can be treated as any data type depending on the context. Meaning that there are no strict distinctions between process instructions, process data, library code, and even within data types. The whole memory is just a continuous block of bytes, and without a context, it is hard to understand where one value ends and another starts. For instance, if a processor is asked to add two values, these values are treated as numbers even if the values stored originally were not supposed to be interpreted as numbers. Furthermore, if a processor's instruction pointer is pointing to a location in a program's data section, then the processor will try to interpret the value at that location as a program instruction and execute it accordingly [3]. Such computer architecture makes it possible to tamper with the memory of a process, which can lead to various consequences, including hijacked execution flow and altered program data [4].

1.1 Problem Statement and Motivation

Memory corruption is a vulnerability that changes the memory of a running program in ways not accounted for by the program developers. The primary cause of memory corruption is the dereference of an invalid pointer. An invalid pointer is a pointer that either points outside the bounds of the memory area it was intended to reference, holds a NULL value, or refers to a memory segment that has already been freed. Invalid pointer can be caused by reasons including but not limited to allocation failures, improper bounds checking, integer overflows, and incorrect code flows when a pointer is freed and then dereferenced. External change to the program’s memory results in a change of the program’s internal state and may lead to a variety of consequences, including, for instance, corruption of internal data, leakage of information, or even control flow hijacking [4].

Despite the memory corruption problem being very old, it remains highly relevant. According to the public vulnerability database as of April 2025, memory corruption vulnerabilities make up a total of 20.18% of all registered vulnerabilities [5]. Furthermore, the overflow vulnerabilities that were also examined in this thesis account for 14.98% of the total documented vulnerabilities [5]. These two categories combined comprise more than a third (35.16%) of all reported vulnerabilities. Such a large number of vulnerabilities from both

categories highlights the relevance of this topic, as well as justifies the efforts for further development of mitigation strategies.

Static code analysis is widely used to detect various bugs in the source code, including memory corruption vulnerabilities. However, the high number of incidents related to memory corruption that continue to happen raises concerns. Are static code analysis tools incapable of detecting memory corruption vulnerabilities, or is the developers' community not utilizing said tools? This thesis aims to address this gap by evaluating current static analyzers and developing a test suite to support future research and tool improvement in this domain.

1.2 Scope

This thesis focuses only on the most popular and most used memory-unsafe programming languages: C and C++. There are more memory-unsafe languages like assembler; however, because they are not so widely used, they are not in the scope of this thesis. Furthermore, apart from the main memory corruption problem, other memory safety violations and integer overflows were also reviewed during vulnerability analysis. The justification for broadening the scope is that vulnerabilities for analysis were taken from a public vulnerability database called CVEdetails [5]. The database assigns all vulnerabilities into categories, and during research, the vulnerabilities from 2 of these categories, overflow and memory corruption, were reviewed. Both categories contain memory corruption vulnerabilities, but additionally contain vulnerabilities that do not exactly change the memory of a running program. For instance, the overflow category, in addition to buffer overflows, also contains integer overflow vulnerabilities. Furthermore, the memory corruption category also contains undefined behavior vulnerabilities. Additional vulnerabilities are not exactly memory corruption, but they frequently lead to memory corruption, like integer overflow [4] or are a serious memory safety violation, like undefined behavior or buffer over-read. Therefore, it was decided not to exclude such vulnerabilities from analysis, leading to a slightly broader scope during vulnerability review.

1.3 Limitations

As with any research, this thesis is not without its limitations. The most significant constraint was time, which was considered in all parts of this thesis. It influenced the scope of the systematic literature review and was also an important factor when analyzing real-world vulnerabilities. In particular, during vulnerability analysis, a time limit was set so as not to invest too much in a single vulnerability. Such a limitation was deemed necessary to maintain steady research progress and not to sacrifice the number of analyzed vulnerabilities for the analysis quality. Another limitation comes from the fact that to analyze vulnerabilities that happened in open-source software, Linux or at least macOS is preferred. While some projects could be built on Windows, the process was generally more efficient on alternative operating systems. Because the author uses Windows as his operating system, all analyzed projects were built on Windows Subsystem for Linux (WSL) version 2.4.13.0. The limitation comes from the fact that not all projects were compatible with WSL, leading to the omission of certain vulnerabilities from the analysis.

1.4 Contributions

This research contributed to the field of static code analysis in multiple different theoretical and practical ways. First, a systematic literature review of modern approaches for detecting memory corruption vulnerabilities was conducted. Afterwards, real-world vulnerabilities were analyzed using a variety of static code analysis tools, whereas each vulnerability's logic was extracted into a separate test case. The created test suite gives an overview of modern memory corruption vulnerabilities and serves as a good base for further development of static code analyzers. In addition, the created test suite is benchexec [6] compatible, making it easy to use, as all test cases can be automatically executed with only one command. Lastly, part of the test cases were contributed to the International Competition on Software Verification (CV-COMP) [7] to be used in verification contests.

1.5 Novelty

This thesis produced three artifacts, which are as follows:

- Systematic literature review of modern approaches for detecting memory corruption vulnerabilities
- Analysis and a test suite of recent memory corruption vulnerabilities
- Overview of static code analysis tools' capabilities on real-world memory corruption vulnerabilities.

All artifacts that were produced are unique and contribute to their respective fields. The first artifact, a systematic literature review, is unique as there has never been a literature review analyzing that topic. While looking for similar systematic literature reviews, the following databases were explored:

- IEEE Xplore [8],
- ScienceDirect [9],
- Scopus [10],
- ACM Digital Library [11],
- International Database of Education Systematic Reviews [12],
- ResearchGate [13].

In all sources the following query was executed to try and find similar systematic literature reviews:

- ("systematic literature review" OR "systematic review") AND ("memory corruption" OR "memory safety" OR "static code analysis")

The closest systematic literature review found is a paper that tries to answer general questions related to static code analysis [14]. Found systematic literature review answered questions like whether there is a static code analysis tool that could analyze any general-purpose programming language.

The second artifact, a test suite of real-world projects, is also unique compared to existing test suites. Test suites that exist are all either synthetic, like the Juliet test suite [15] and NIST test suite [16], or are automatically extracted and are neither simplified nor executable [17, 18, 19]. The problem with synthetic data sets is that while they contain vulnerabilities grouped by a Common Weakness Enumeration (CWE), they do not represent real-world vulnerabilities. This means that test results of analyzing a synthetic dataset may not necessarily reflect static code analyzers' capabilities of identifying vulnerabilities in real projects. Meanwhile, existing real-

world vulnerability datasets are constructed in a way that final test cases cannot even be compiled. For example, BugBench [17] contains only patch files for vulnerabilities, which are not mapped to any Common Vulnerabilities and Exposures (CVE) number or CWE. DiverseVul [18] and Devign [19] both contain vulnerable functions in JavaScript Object Notation (JSON) format that were automatically extracted from projects. Extracted functions are not simplified or altered in any way and thus cannot be executed due to missing dependencies. While the ability to compile a test case is not mandatory for every static code analyzer tool, it is definitely an advantage. The ability to compile a test case allows code analyzer tool authors to better understand the vulnerability by executing the test case and either observing the error trace log or exploring execution in a debugger. Moreover, because of the complex nature of real-world functions that may contain hundreds or even thousands of lines of code, it is not always clear where the vulnerability is in a function that was just extracted from a program. In the test suite created as part of this research, the vulnerability is always highlighted by a comment with an explanation, making it easier to understand the context.

While simplification may make the vulnerabilities easier to find for static code analyzers, the simplified dataset of real-world vulnerabilities greatly benefits the code analyzers community. The main idea of the test suite is to be utilized for the further refinement of static code analysis tools. That is also the benefit the test suite brings to the community. The test suite consists of test cases that are all single files that utilize only standard C/C++ libraries. Such test cases can be more easily executed than the programs from which they were extracted, which frequently require external dependencies. Besides, simplified vulnerabilities are more concise, which makes it easier to follow the internal state of the analyzer due to the reduced number of statements.

The code simplification with all its advantages also has its downsides. As code simplification before analysis is a form of preprocessing, using the created dataset as test data for training an artificial intelligence model would be pointless. If a model was trained on the dataset, it would require any future input to the model to undergo the same preprocessing. However, because there already exist datasets like DiverseVul [18] and Devign [19] that serve as good test data for artificial intelligence models, this thesis instead focused on test data for conventional code analyzer tools.

Lastly, the analysis of existing static code analysis tools on real-world memory corruption vulnerabilities is also unique. Existing research on industry standard static code analysis tools

either utilize test suites for evaluation [20, 21, 22, 23] or have an unrepresentatively low number of evaluated projects [21, 24]. In this study, more than a hundred CVE vulnerabilities were analyzed in open-source projects. Such a large number of evaluated projects gives an unbiased and sophisticated overview of the capabilities of static code analyzer tools on real-world vulnerabilities.

1.6 Research Questions

The main goal of this thesis is to contribute to the field of static code analyzers, with a particular focus on the detection of memory corruption vulnerabilities. While there are numerous ways this thesis could contribute to the chosen research area, a testing/validation approach was chosen. At one point, a development or a contribution to a static code analyzer was also considered. Ultimately, this idea was abandoned as it was hard to determine the scope of the development and justify the contribution. Instead, this thesis aims to provide an overview of the capabilities of static code analyzers. As such, the main research question is as follows.

- **How well can modern static code analyzers detect memory corruption vulnerabilities?**

The main research question is further broken down into four smaller-scope research questions that are as follows:

- **RQ1**: What are the approaches to detecting memory corruption vulnerabilities using the source code?
- **RQ2**: What are the memory corruption vulnerabilities that exist nowadays?
- **RQ3**: How well can modern static code analyzers detect memory corruption vulnerabilities in real-world projects?
- **RQ4**: How well can modern static code analyzers detect memory corruption vulnerabilities in a test suite?

The whole thesis is built around the aforementioned research questions, where each of the research questions is answered in a separate section of the thesis.

1.7 Structure

The thesis is divided into six parts, as follows:

- Chapter 1 is an introduction that presents the main topic of the thesis as well as states the author's motivation, defines the scope, lists limitations, identifies a problem, outlines contributions with their novelty, and formulates research questions.
- Chapter 2 is a systematic literature review that provides an overview of modern approaches for detecting memory corruption vulnerabilities.
- Chapter 3 presents an overview of modern memory corruption vulnerabilities. The most significant contribution of this section is a test suite based on real-world vulnerabilities.
- Chapter 4 includes the selection of static code analysis tools utilized during research, as well as provides results of real-world vulnerabilities static code analysis.
- Chapter 5 provides an overview of the static code analysis results of the extracted test suite.
- Chapter 6 is the conclusion, where the results of this thesis are summarized. In addition, the section also refers to the main research questions to give final answers, as well as states potential further research ideas.

1.8 Methodology

The research methodology used varies between sections of the thesis. The whole thesis consists of 3 distinct parts, each focusing on a different type of information and making its unique contribution. Therefore, each part utilizes a different methodology.

The first part, that is, the systematic literature review, utilizes qualitative research methodology. While performing the literature review, scientific literature was found from academic literature databases, and each paper was read and summarized. The nature of information that is processed during this part is not numeric and depends on the context, making it subject to interpretation. The literature review section aims to explore modern approaches for finding memory corruption vulnerabilities. Probably the only threat to validity that can be related to this section is paper selection bias. This threat was partially mitigated by a systematic literature review approach, where all papers found using a specific query were reviewed. If only a systematic literature review were to be used as part of this research, a systematic approach would be sufficient to mitigate any bias for source selection. However, the number of articles found by utilizing a systematic literature review was insufficient, so snowballing was also used to gather review sources. During snowballing, selection bias was mitigated by

inclusion/exclusion criteria as well as the general idea that any new reviewed research should benefit the literature review with new knowledge.

The second part, which involves the analysis of existing memory corruption vulnerabilities, mainly focuses on the construction of a test suite. The section also falls under qualitative research mainly because of the nature of the information analyzed. The data analyzed is the source code of open-source programs, and such information is hard to measure or quantify. In addition, vulnerability logic extraction is a subjective action with no single correct answer. Not having a single correct answer again shows how context-sensitive the nature of information processed is, making the information fall under qualitative research. The idea behind this section is straightforward. First, reported CVEs related to memory corruption vulnerabilities that occurred in either C or C++ source code were collected. Each vulnerability was then reviewed, and the logic related to the vulnerability was extracted into a test case. During the extraction of vulnerabilities, vulnerable logic was simplified, and comments were added to either add context or explain source code logic. Furthermore, the mapping of vulnerability origins and vulnerabilities was preserved by adding a comment to each test case that would uniquely trace back to the original place from which the vulnerability was extracted.

Test suite creation mainly has two problems. These problems are vulnerability selection bias and the extraction process, which could lose vulnerability. The vulnerability selection bias was reduced by making the selection process semi-systematic. This means that all vulnerabilities that satisfied a certain selection criteria were analyzed. The second problem was mitigated by utilizing AddressSanitizer [25] to make sure the vulnerability is present in the extracted logic.

The last two sections contain the evaluation of static analyzers on real-world programs and the test suite, respectively. Both sections utilize quantitative research methodology as they operate on structured information that is not subject to different interpretations and does not depend on the context. The information processed is represented in numbers, and the whole analysis is very specific, with the only subjective part being an interpretation of the analysis results. Potentially, the biggest threat to validity in this section is bias during the selection of static code analyzers. In this research, selection bias was mitigated by composing a list of requirements that a static code analyzer had to meet in order to be selected. Such a selection process allowed for the same evaluation of various tools, thus making the selection less biased, more predictable, and consistent with academic norms.

2. Systematic Literature Review

In this section, a systematic literature review is performed to answer the first research question. This section aims to provide an overview of modern methods that can be used to detect memory corruption vulnerabilities using the source code.

2.1 Literature Sources and Search Keywords

To perform a systematic literature review, the following academic paper databases were used:

- IEEE Xplore [8],
- ScienceDirect [9],
- Scopus [10],
- ACM Digital Library [11].

The same query was executed in all previously mentioned sources to find all papers related to the first research question. Search was performed only on the abstract, and if the search engine allowed, on the title and keywords. Such a limitation was needed first to limit the number of papers found and second to find more suitable papers. The query that was used for searching is provided below:

- ("memory corruption" OR "memory safety" OR "use-after-free" OR "use after free" OR "buffer overflow" OR "use-before-initialization" OR "use before initialization") AND ("static analysis" OR "source code analysis" OR "static code analysis" OR "static detection" OR "source code detection" OR "static code detection") AND ("C" OR "C++" OR "Cplusplus")

2.2 Inclusion and Exclusion Criteria

Inclusion and exclusion criteria are meant to further refine the found paper selection by providing a framework for evaluating whether a paper is suitable for analysis. The inclusion criteria were composed to pick only the most relevant papers that describe approaches to memory corruption detection. The exclusion criteria were selected first to exclude articles for which information extraction is impossible. Information extraction may not be possible if the paper is written in a language the author does not know or if the paper is just not accessible. Also, exclusion criteria remove old papers that were released before the year 2020, as well as papers that are too short. While removing old papers was done to focus only on the most recent development, the exclusion of papers shorter than five pages was done to filter papers that may lack quality and depth. Only the content length was considered when computing paper length, and paper parts like the abstract or sources were omitted.

Inclusion criteria were as follows:

- The paper describes one or more approaches to detecting memory corruption vulnerabilities;
- The paper talks about general concepts or is related to C or C++ programming languages;
- Described approaches rely on the source code.

Exclusion criteria were as follows:

- The paper is not written in English;
- The paper is not freely accessible to the author;
- The paper is older than five years (released before 2020);
- The paper is shorter than five pages.

2.3 Data Extraction

A data extraction form was used to consistently extract information from papers that can be found in Appendix 1 of this document. A data extraction form was created to record all the implementation details, working principles, utilized technologies, as well as advantages and limitations of a particular approach. In addition, the data extraction form also included fields to gather approach evaluation results, if any were available, and implementation availability.

2.4 Paper Selection

The papers for this systematic literature review were extracted from four different academic paper databases. After removing all duplicate documents from the search results, articles were evaluated based on title and abstract. A total of 17 articles were selected based on the initial title and abstract filtering. Next, all papers were read, which eliminated seven more documents that were not suitable for the research. Papers that were eliminated in this step are brought out with the elimination reason in the following section. Because the number of papers found and accepted was insufficient to provide a comprehensive overview, the snowballing technique was further utilized to increase the number of reviewed studies. Utilizing snowballing, 10 additional articles were found that not only provided new information to the literature review but also met the inclusion/exclusion criteria. The overall paper selection process is shown in Figure 1.

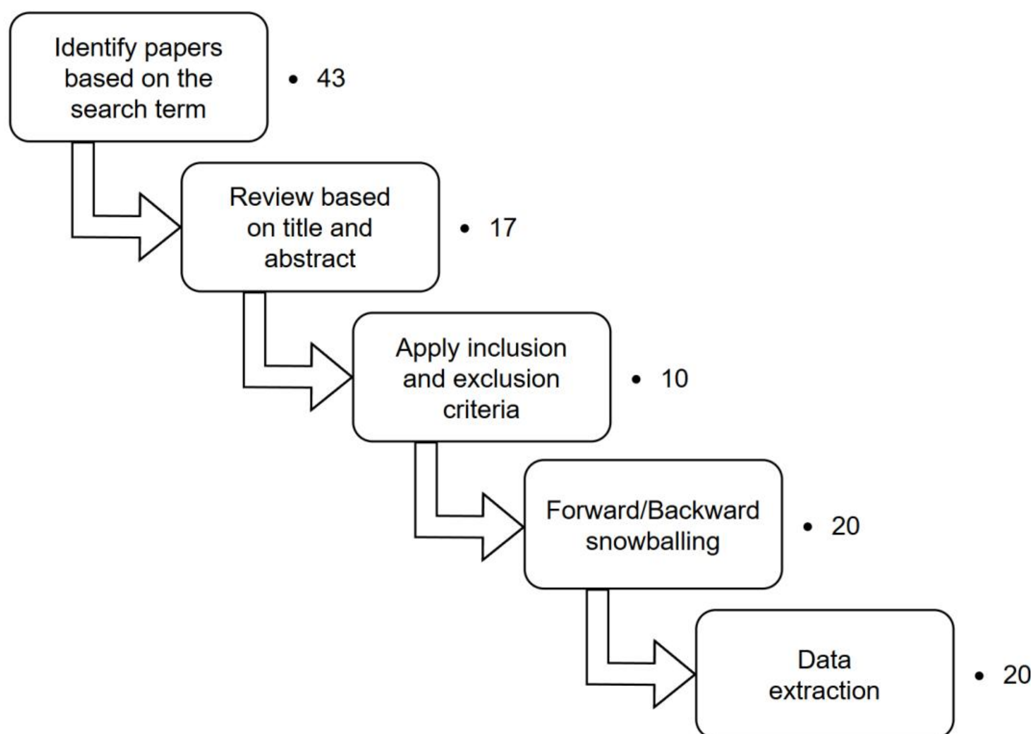

Figure 1. Systematic literature review paper selection diagram

2.5 Excluded Papers

Out of 17 initially picked papers, seven were excluded for various reasons. Three papers, namely, **Automatic Buffer Overflow Warning Validation** [26], **Towards a Technique to Detect Weaknesses in C Programs** [27] and **SAVER: Scalable, Precise, and Safe Memory-Error Repair** [28] were excluded because they do not present any new approach to finding memory corruption vulnerabilities and instead use existing static code analyzers for vulnerability finding. Their research targets ways for automatic patch generation and reducing the number of false negatives produced by static code analyzers. **Compositional Vulnerability Detection with Insecurity Separation Logic** [29] is a fascinating study that focuses on finding vulnerabilities related to information leakage. Despite the problem close to memory, it is not related to memory corruption, which is outside of the scope of this systematic literature review. **Effectiveness on C Flaws Checking and Removal** [30] is only two pages long, making its length smaller than minimally acceptable by the exclusion criteria. In addition, two papers were excluded because the author could not access them. These two papers are: **Enhancing the Quality and Security of IoT Software Systems Using Cloud-Based Vulnerability Detector** [31] and **Detecting use-after-free bugs in embedded C programs** [32].

2.6 Overview of Fundamental Approaches Used by Reviewed Papers

Overall, all reviewed papers relied on some kind of fundamental approach for static code analysis. Before summarizing the read papers, it may be beneficial to understand some of the foundational techniques utilized for program verification. Therefore, most fundamental verification approaches used in the reviewed papers are briefly introduced before the literature review summary. The fundamental approaches explained are the following:

- symbolic execution, [33]
- source code graph traversal, [34]
- program model checking, [35]
- code matching. [36]

2.6.1 Symbolic Execution

Symbolic Execution (SE) is an approach for software verification that statically runs the program with symbolic inputs. The main idea is that program variable values are substituted with symbolic values, and during the static execution of the program, symbolic values get constrained by code statements. During the execution, a Symbolic Execution Tree (SET)

structure is created that holds all possible symbolic states of a program. A SET is a tree data structure that grows downward, and where the very first node is an entry point. Each node (also known as *Symbolic Execution State* (SES)) holds mapping of program variables to symbolic values (also known as *symbolic store*), constraints to symbolic values the program has enforced up to that point (also known as *path condition*) and a program counter that points to the statement of execution (also known as *program counter*). In case the symbolic execution engine encounters a control statement (like if, while, switch, for loop, etc.), it evaluates its feasibility before exploring it. If a statement is feasible with applied constraints (*path conditions*), then the branch that reached a control statement splits to explore all feasible paths. For example, for an if statement, that means that one SET branch splits into two branches, where one branch explores the program path where the if statement is evaluated to false, and another branch where the if statement is evaluated to true. In addition, the constraint that the if statement introduces gets appended to the path conditions of both branches [33].

The main problem with symbolic execution is path explosion. In SET each path from the root node to a leaf node is one possible program execution flow. In case the program under analysis is complex with many possible execution paths, the number of branches of a SET would increase very fast. While this is the most serious obstacle for SE, there are several ways to mitigate this drawback. One approach is to drop paths that are similar or paths that were previously visited by another branch in SET. Another approach to reduce the number of SET branches is to merge the symbolic states of nodes with the same program counter (nodes that evaluate the same line of code in the source code). Additionally, function calls can be abstracted in a way that only changes the SET branch once with a summary (or a contract) of the function's behavior [33].

2.6.2 Source Code Graph Traversal

The idea of source code graph traversal lies in the fact that source code can be represented as a graph. Graph representation can then be automatically traversed to identify potential vulnerabilities based on predefined rules. Various code graph representations have been developed to capture different properties of the source code [34]. In the background section, not all the graphs that were encountered during the systematic literature review are explained. This is mostly because to understand the systematic literature review, there are no strict requirements for the reader to understand each graph. All reviewed graphs also have an example that represents the function foo [34], whose implementation can be found below.

```
void foo() {
  int x = source();
  if (x < MAX) {
    int y = 2 * x;
    sink(y);
  }
}
```

Abstract Syntax Tree

One of the first graphs to be produced by the compiler is an Abstract Syntax Tree (AST). This form of code representation captures how statements and expressions are nested to produce a program. The nodes of a tree represent operators, and the tree leaves represent operands [34]. An example of AST of a foo function is shown in Figure 2.

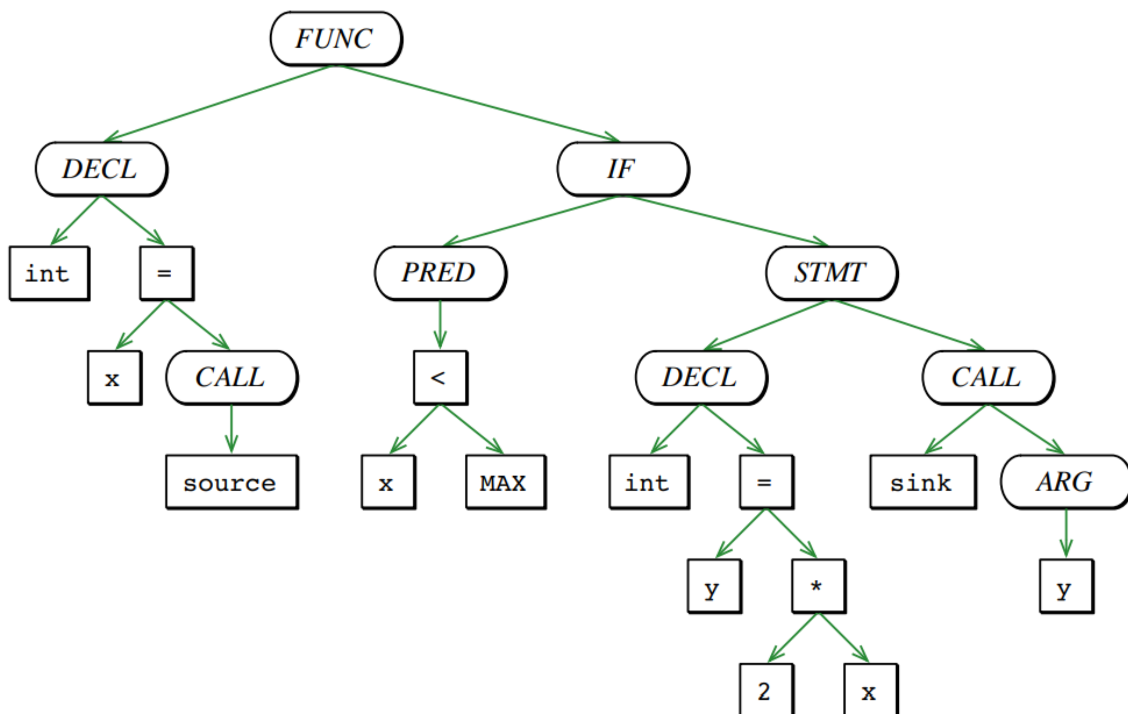

Figure 2. Abstract Syntax Tree of foo function [27]

Control Flow Graph

Control Flow Graph (CFG) explicitly shows the order in which source code statements are executed. The nodes of a graph represent program statements and predicates. The edges represent the flow of control during program execution. Each control statement node in the CFG has multiple edges that originate from it that represent different control flows depending

on the evaluation of the control statement [34]. An example of a CFG of a foo function is shown in Figure 3.

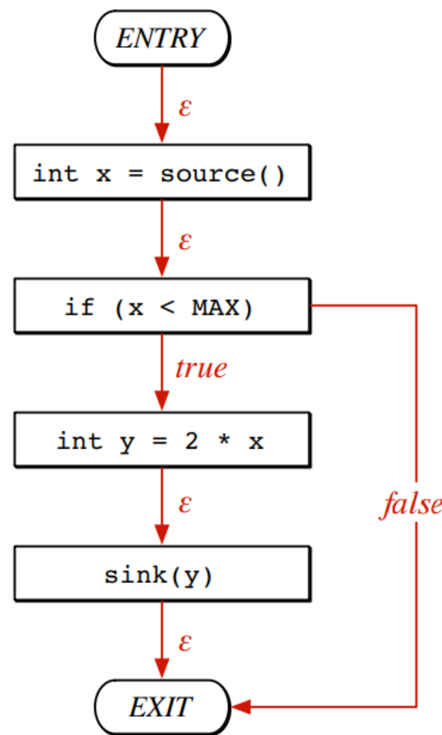

Figure 3. Control Flow Graph of foo function [34]

Dataflow Analysis

One example of program analysis that can be done using a CFG is called dataflow analysis. This type of analysis traverses through CFG nodes and assigns constraints for each variable based on data inferred from the graph nodes. The idea is that throughout the execution of a program, there are statements that define or modify a variable, and these statements ultimately constrain the variable as they propagate through the CFG. By gathering all constraints that a certain variable has at a certain point in a program, its value can be approximated. By reasoning about the variable's approximate value, a certain program property can be verified. For instance, it can be verified that all variables used in a program are initialized. This can be done for each statement, e.g., node in a CFG, by traversing through nodes and asserting that every variable used at a particular program point has been initialized along all possible execution paths leading to that point [37].

Program Dependence Graph

A Program Dependency Graph (PDG) is a graph code representation that explicitly captures dependencies among statements and predicates. The graph nodes are program statements, and the edges reflect either the connection of one variable to another (called a *data dependency edge*) or the connection between a predicate and the value (called a *control dependency edge*) [34]. An example of a PDG of a foo function is shown in Figure 4.

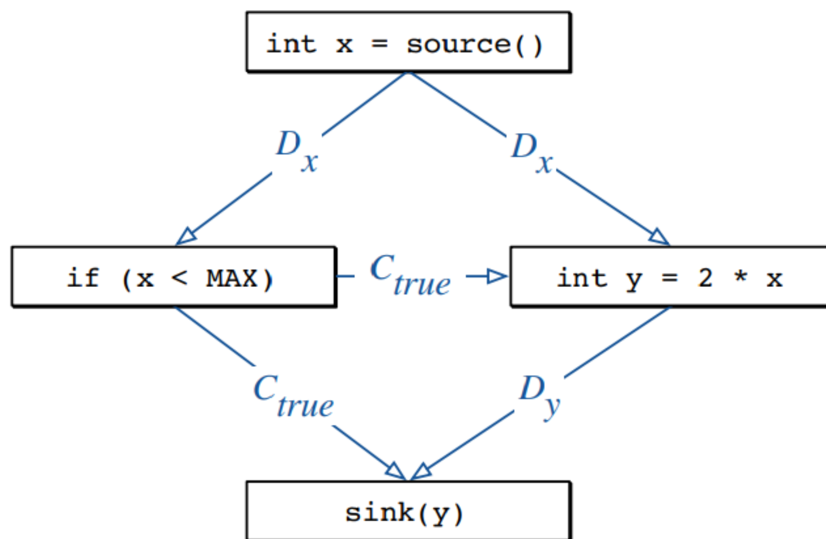

Figure 4. Program Dependence Graph of foo function [34]

Code Property Graph

Code Property Graph (CPG) is a graph code representation that combines AST, CFG, and PDG into one. Its idea is that a single code representation is not always sufficient to uniquely characterize a vulnerability, and thus, by combining multiple representations together, vulnerabilities can be identified more precisely [34]. An example of a CPG of a foo function is shown in Figure 5.

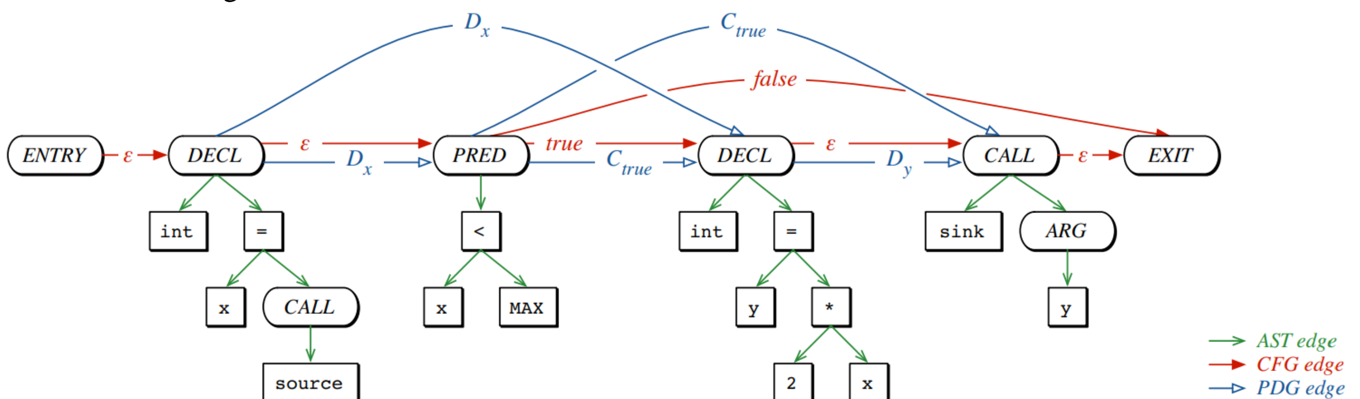

Figure 5. Code Property Graph of foo function [34]

2.6.3 Program Model Checking

Model checking is a technique that can be used to verify software for different properties that the system may or may not have. The main idea behind model checking is that every program can be split into different states that follow each other. Simple programs that do not utilize threads have only one chain of states during the whole execution process. More complex programs that execute many threads concurrently can be visualized as a complex graph. The idea of model checking is to prove that some property exists or is absent (such as the absence of use-after-free vulnerability), a model needs to be explored that consists of finite states. Depending on the property that needs to be verified, a lot of different verification approaches are available. If the property needs to hold for every state in a program, then to prove it, every state in a model is evaluated. It is also possible to check properties only for a single path (a sequence of states in a model) or for a path until a certain state [35].

Model checking can be automated, and there are several different model checking engines that operate on different formal system specifications. Nevertheless, such an advantage requires a program specification to be written in a way that a model-checking engine can understand. Model checking for formally described systems can be done in two ways. The first approach is to construct a model-checking engine that would be able to operate on the specification of a system. The second approach is to convert a specification of a system into an already existing model-checking engine input language. While the second approach is usually easier, to ensure the correctness of the model, it is important for the translation to be correct [35].

2.6.4 Code Matching

A code matching technique, also often called a clone detection technique, is used to detect similarities between two fragments of source code [36]. The detection of duplicate code can also be utilized to detect vulnerabilities in source code. For instance, this technique was used in **MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures** [38]**,** where functions with a known vulnerability were compared to other functions to detect whether a vulnerability exists in a function or not. The idea of utilizing a code clone detection technique in program verification is to compare a known vulnerable fragment of code to another. If two fragments are similar, then the code fragment may contain a vulnerability present in the code fragment to which it was compared [36].

Code clone detection methods and granularity of analysis can vary depending on the scope. Clone detection can be conducted across an entire program or on smaller code fragments, such as functions, classes, interfaces, statements, or any code blocks that can be logically separated. In addition, there are different techniques to compare code fragments with each other [36].

The simplest is a text-based/string-based approach that directly compares source code lines between two code fragments. If one fragment's code lines are found in another fragment's code, then the two fragments are duplicates to some extent. The text-based approach is sensitive to formatting variations. Code lines are compared directly, and the presence of non-important characters such as whitespace may already make two lines of code different. A common way to mitigate this limitation is to normalize the source code (e.g., adjusting variable names, removing comments, standardizing formatting), although this is not always done [36].

A more complex clone detection technique is token-based detection. In the token-based technique, fragments of source code are first split into individual code components called tokens. To detect code clones, sequences of tokens are compared between two code fragments. Such an approach is more robust because it can detect similar code even when the source code has different formatting [36].

Another technique to detect code duplicates is a tree-based technique that utilizes an Abstract Syntax Tree (AST) for code clone detection. After AST is generated from the source code, it is traversed to find similar subtrees between two fragments of the AST representations [36].

More advanced approaches also consider control flow and data flow information in code matching. This can be achieved by utilizing a Program Dependency Graph (PDG) instead of an AST because a PDG contains both control flow and data flow information. PDG-based methods analyze similarities in subgraphs, providing a more comprehensive detection of clones even when compared code fragments have structural differences [36].

Finally, there is also a metric-based technique that does not compare code statements directly and instead compares code properties. Such an approach can find code clones between two fragments based not on syntax but on code properties such as the number of use definitions of non-local variables or the number of lines in a function [36].

2.7 Conclusion

In total, 20 articles were reviewed as part of this systematic literature review, with short summaries of each included in Appendix 2 of this thesis. From the read articles, 14 rely on graphs, four rely on artificial intelligence, five rely on code matching, four rely on fuzzing, three rely on symbolic execution, one relies on taint pointers analysis, and three rely on model checking to find vulnerabilities in the code. As seen from the articles reviewed, approaches often relied on multiple techniques for vulnerability detection. Of all the reviewed approaches, only eight utilize a single method for vulnerability detection, 10 utilize a combination of two methods, and two use a combination of three methods for vulnerability detection.

The most interesting part is definitely to compare novel approaches in terms of their practical results. Unfortunately, the ranking cannot be complete since different research has focused on various parameters. For instance, papers that describe fuzzing approaches [39, 40, 41] mostly focus on the time taken to find vulnerabilities rather than focusing on accuracy or the number of false positives. In addition, some papers [42, 43, 44] do not state the number of false positives/false negatives and focus on different metrics relevant to their research. The exact evaluation criteria for each approach with the utilized techniques are provided in Table 1.

Table 1. Evaluation criteria per paper and technique used

Paper	Evaluation programs	Technique
An Efficient Metric-Based Approach for Static Use-After-Free Detection [45]	Juliet test suite + 8 real-world programs	Graphs and code matching
Enhancement in Buffer Overflow (BOF) Detection Capability of Cppcheck Static Analysis Tool [46]	Juliet test suite	Graphs
Bounded Model Checking for LLVM [44]	1 real-world program	Model checking
On the Path to Buffer Overflow Detection by Model Checking the Stack of Binary Programs [47]	10 programs from NIST SARD	Model checking
CorCA: An Automatic Program Repair Tool for Checking and Removing Effectively C Flaws [48]	NIST SARD dataset + 7 real-world programs	Fuzzing
Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel [49]	Linux kernel	Graphs and symbolic execution
Lightweight Shape Analysis Based on Physical Types [50]	Li Et Al. C benchmarks	Model checking

DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network [51]	NIST SARD dataset + 2 real-world programs	Graphs and artificial intelligence
Vulnerability Analysis of Similar Code [43]	Big-Vul dataset	Code matching
Static Checking of Array Index out of Bounds Defects in C Programs Based on Taint Analysis [52]	11 real-world programs	Graphs, tainted analysis, and symbolic execution
SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer [41]	Linux kernel	Code matching and fuzzing
A Method of Firmware Vulnerability Mining and Verification Based on Code Property Graph [42]	DVRF dataset	Graphs
Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs [53]	4 real-world programs	Graphs and symbolic execution
Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences [54]	Linux kernel	Graphs and artificial intelligence
Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [40]	14 real-world programs	Graphs and fuzzing
Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths [55]	4 real-world programs (including Linux kernel)	Graphs
MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures [38]	10 real-world programs (including Linux kernel)	Graphs and code matching
Learning to Detect Memory-related Vulnerabilities [56]	MemoryVul test suite	Graphs and artificial intelligence
LineVD: Statement-level Vulnerability Detection using Graph Neural Networks [57]	Big-Vul dataset	Graphs and artificial intelligence
HotCFuzz: Enhancing Vulnerability Detection through Fuzzing and Hotspot Code Coverage Analysis [39]	4 real-world programs	Graphs, code matching, and fuzzing

Nevertheless, the most common metrics across all reviewed research were the number of total vulnerabilities found and how many of them were further classified as True Positives (TP) or False Positives (FP). The formula that considers both TP and FP and that was used to rank reviewed papers is FP / (FP + TP). The formula gives a percentage of found vulnerabilities that were wrongly classified as vulnerable, and such a metric is called the False Positive Rate (FPR).

As there are research that used both real-world applications along with a test suite [45, 48, 51] it is essential to note that real-world results were prioritized for ranking. Using the formula, the ranking of reviewed papers is shown in Table 2.

Table 2. Self-reported precision ranking of reviewed approaches

Paper	FPR (%)
On the Path to Buffer Overflow Detection by Model Checking the Stack of Binary Programs [47]	0
CorCA: An Automatic Program Repair Tool for Checking and Removing Effectively C Flaws [48]	0
Lightweight Shape Analysis Based on Physical Types [50]	11.76
Static Checking of Array Index out of Bounds Defects in C Programs Based on Taint Analysis [52]	14.13
MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures [38]	16.38
Learning to Detect Memory-related Vulnerabilities [56]	31.47
Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences [54]	34.96
An Efficient Metric-Based Approach for Static Use-After-Free Detection [45]	37.5
Enhancement in Buffer Overflow (BOF) Detection Capability of Cppcheck Static Analysis Tool [46]	40.05
Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs [53]	40.7
Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel [49]	50
Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths [55]	63.53
LineVD: Statement-level Vulnerability Detection using Graph Neural Networks [57]	72.68

While paper **On the Path to Buffer Overflow Detection by Model Checking the Stack of Binary Programs** [47] is ranked first, its evaluation is very confusing, and the paper only states that it chose 10 small C programs from NIST SARD. The paper does not state what programs it chose and provides no further clarification of test results, only stating that all 10 instances of vulnerabilities were found. Nevertheless, it is ranked first based on the data provided in the article. Table 2 only ranks 13 research, the remaining seven articles that could not be ranked with corresponding reasons are presented in Table 3.

Table 3. Not ranked research reasons

Paper	Not ranking reason
Bounded Model Checking for LLVM [44]	The time taken for program analysis was analyzed.
DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network [51]	Research gives several metrics (like accuracy, F1 score, false positives rate), but it is not possible to calculate a universally used value to rate this paper based on the values provided.
Vulnerability Analysis of Similar Code [43]	Purely theoretical approach
SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer [41]	Analyzed was the time-to-exposure of vulnerabilities.
A Method of Firmware Vulnerability Mining and Verification Based on Code Property Graph [42]	Analyzed was graph traversal coverage
Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [40]	Analyzed was the time taken to find vulnerabilities and the number of vulnerabilities found at the end of 24 hours.
HotCFuzz: Enhancing Vulnerability Detection through Fuzzing and Hotspot Code Coverage Analysis [39]	Analyzed was the time taken to find vulnerabilities and the number of vulnerabilities found at the end of 24 hours.

Based on the ranking of the reviewed papers, it is also possible to construct a ranking table for the fundamental approaches utilized in the reviewed research. For that, the score of each technique is calculated based on the average sum of all approaches in which it is utilized. It is important to note that, as there are two papers whose score is 0, they are both considered to be first. A code matching technique will be used to demonstrate the logic behind score computation. The code matching technique appears 2 times in the ranking table for approaches ranked in 4th and 7th places. From here it can be calculated that its score is the average of these ranking places, which is 5.5. Techniques for vulnerability detection with their respective scores in a diminishing order are presented in Table 4.

Table 4. Self-reported ranking of techniques for vulnerability detection based on precision

Techniques for vulnerability detection	Score
Fuzzing	1
Model Checking	1,5
Tainted analysis	3
Code matching	5,5
Symbolic execution	7,3
Graphs	7,5
Artificial intelligence	7,6

A variety of programs and test suites were used for the experimental evaluation of reviewed papers. Of the 20 reviewed research papers, 10 utilized some test suite for evaluation, and 13 used real-world programs for evaluation. Once again, the total sum is greater than 20 because some papers like DeepWukong [51] have utilized both a test suite and real-world applications for the tool evaluation. Among all reviewed papers that use real-world applications for approach evaluation, the average number of programs used is 5. The most frequently analyzed real-world application is the Linux kernel, which was examined in 5 separate studies. Among test suites, the most used was a synthetic test suite called NIST SARD [16] that was utilized in three studies.

Different papers also focus on various types of vulnerabilities. From all 20 reviewed papers, only six focus on a single vulnerability, like buffer overflow or use-after-free, and most of the reviewed papers can find a wide range of vulnerabilities. While the ability to find memory corruption vulnerabilities is a part of the inclusion criteria for this systematic literature review, some papers also find other types of vulnerabilities (including memory corruption vulnerabilities). Table 5 shows how many papers find a certain type of vulnerability.

Table 5. Number of papers that find specific vulnerabilities

Vulnerability	Number of research that can find it
All kinds of vulnerabilities (including memory violations)	10
Memory violations	3
Use after free	2
Buffer overflow	3
Index out of bounds	1
Use before initialization	1

As mentioned previously, many approaches rely on graphs for code representation. The graphs used vary, and overall, nine different graphs were used. As with techniques for vulnerability detection, papers often rely on multiple graphs for code representation. All graphs used with their respective usage count are provided in Table 6.

Table 6. Graph utilization

Graph	Number of papers using it
Code Property Graph	4
Control Flow Graph	7

Data Flow Graph	1
Program Dependency Graph	5
Data Dependency Graph	1
Call Graph	5
Value Flow Graph	1
Abstract Syntax Tree	6
System Dependency Graph	1

Finally, the approach's implementation accessibility can also be reviewed. Out of all 20 reviewed papers, only 8 have some kind of implementation publicly available. The result is provided based on references found in articles, as well as a separate search by the article/approach name. Out of all eight tools, six can be used without much configuration, while **SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer** [41] and **Learning to Detect Memory-related Vulnerabilities** [56] tools require either per-project configuration or training before the tool can be utilized. All available tools are shown in Table 7.

Table 7. Publicly accessible tools of reviewed approaches

Paper	**Tool**	**Can be used out-of-the-box**
CorCA: An Automatic Program Repair Tool for Checking and Removing Effectively C Flaws [48]	CorCA [58]	Yes
Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel [49]	IncreLux [59]	Yes
Lightweight Shape Analysis Based on Physical Types [50]	- [60]	Yes
DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network [51]	DeepWukong [61]	Yes
SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer [41]	SyzRisk [62]	Requires per project configuration
Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs [53]	NDI [63]	Yes
Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths [55]	IPPO [64]	Yes
Learning to Detect Memory-related Vulnerabilities [56]	MVD [65]	Requires training

3. Analysis of Memory Corruption Vulnerabilities

This section investigates current memory corruption vulnerabilities and extracts representative benchmarks from this investigation to answer **RQ2**. The section includes CVE selection criteria, as well as the final number of vulnerabilities selected with their CWE classification. In addition, the contribution to SV-COMP is explained, and some common patterns that lead to memory corruption are reviewed based on vulnerabilities found during this research. Lastly, some statistics are also provided for the test suite and real-world projects that were analyzed.

3.1 CVEs Selection Criteria

As mentioned in the methodology section, a set of criteria was made to reduce vulnerability selection bias and, at the same time, define a clear upper limit for the number of vulnerabilities under analysis. The requirements that vulnerability had to meet were as follows:

- CVE should be classified as Overflow or Memory corruption;
- CVE should be registered during 2023;
- CVE should have a link to GitHub.

The classification restriction comes from the fact that the CVEdetails [5] database contains CVEs related not only to memory corruption or overflow. As for the year constraint, it was included as it helps to greatly reduce the number of CVEs that must be analyzed. As more than 29 thousand vulnerabilities occurred during 2023 [5], such a restriction is not overly restrictive as it still includes thousands of vulnerabilities related to memory corruption. The reason for choosing the year 2023 comes from the fact that the analysis was started in 2024. The decision to analyze last year's vulnerabilities was made as most of the previous year's vulnerabilities have been patched, making analysis easier in some cases. It is easier to understand where vulnerability happened when you not only see the source code with a stack trace but also what changes were made to mitigate the vulnerability. Finally, the process of gathering CVEs should be explained first to understand the last criterion. The analysis started by gathering all CVEs using a script written in JavaScript that went through all pages on CVEdetails [5] and automatically saved each CVE that fit the criteria into a JSON file. Both the JavaScript script used and the composed JSON file with vulnerabilities can be found on GitHub in a repository where all analysis results are stored [66]. Going back to the last selection criterion, as lots of CVEs come from proprietary software without publicly accessible source code, there was a need to somehow filter based on source code accessibility. Such a filtering condition was the presence of the GitHub link. While it helped to reduce the number of CVEs without publicly

available source code, it was not an absolute solution. In fact, half of all gathered CVEs still did not have a source code. The reason for that is that many vulnerabilities had a link to a GitHub with pictures and a vulnerability description but without actual source code. Such vulnerabilities were not analyzed and were immediately skipped.

3.2 Selected CVEs

In total, 864 CVEs were gathered from CVEdetails [5] website, but the final number of analyzed CVEs is much lower due to various factors. These factors are as follows:

- Proprietary source code;
- Not a C or C++ programming language;
- Undetermined cause of vulnerability or high time consumption.

Out of the three factors, the first two are straightforward and easy to understand. Meaningful analysis is impossible without the source code. In addition, as this thesis focuses only on C/C++ programming languages, all vulnerabilities in other programming languages were not reviewed. The last factor is controversial because many CVEs were excluded even though their C/C++ source code was available. While such a criterion may seem inappropriate as it is too subjective, it is crucial to understand why and where the vulnerability happens in the source code. Such information is required not only for correct vulnerability logic extraction but also to determine whether a vulnerability was found using a static code analysis tool or not. While in most cases it was possible to understand where vulnerability occurs, in some instances it was extremely challenging, and vulnerabilities where the exact place of occurrence could not be determined were skipped. Additionally, if vulnerability extraction took too long, the vulnerability was also skipped. The time allocated for the analysis of each vulnerability was 2 hours. During that time, it should have been possible to understand vulnerability, compile the project, analyze vulnerability using different static code analyzer tools, and extract the vulnerability logic. Because of the dominance of drivers, operating systems, low-level encoding/decoding software, compilers, and other complex software, many vulnerabilities were skipped as their analysis required too much time. In total, 150 CVEs were reviewed across 92 unique projects. The breakdown of all scrapped vulnerabilities can be seen in Table 8.

Table 8. CVEs per category

Analyzed	No source code	Not C/C++	Skipped
150	434	51	237

The sum of all numbers in Table 8 results in 872 vulnerabilities, which is slightly greater than the total number of CVEs scrapped from CVEdetails. The difference comes from the fact that before starting with full-scale analysis, some vulnerabilities were analyzed to determine the feasibility and relevance of this research. As such, some analyzed CVEs do not fall under the selection criteria but still count as analyzed CVEs. All analyzed vulnerabilities are categorized into one or several CWE groups. All CWE categories with a respective number of vulnerabilities analyzed are shown in Table 9.

Table 9. Analyzed vulnerability count by CWE category

CWE	Vuln. number	CWE	Vuln. number
Incorrect Access of Indexable Resource (CWE-118)	1	Off-by-one Error (CWE-193)	1
Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119)	4	Unchecked Return Value (CWE-252)	1
Buffer Copy without Checking Size of Input (CWE-120)	50	Uncontrolled Resource Consumption (CWE-400)	1
Stack-based Buffer Overflow (CWE-121)	4	Double Free (CWE-415)	4
Heap-based Buffer Overflow (CWE-122)	12	Use After Free (CWE-416)	1
Out-of-bounds Read (CWE-125)	1	NULL Pointer Dereference (CWE-476)	17
Improper Validation of Array Index (CWE-129)	2	Unchecked Return Value to NULL Pointer Dereference (CWE-690)	1
Use of Externally-Controlled Format String (CWE-134)	1	Improper Check for Unusual or Exceptional Conditions (CWE-754)	1
Numeric Errors (CWE-189)	1	Out-of-bounds Write (CWE-787)	62
Integer Overflow or Wraparound (CWE-190)	14	Use of Uninitialized Resource (CWE-908)	2

3.3 Test Suite

Out of 150 analyzed vulnerabilities, 113 were converted into test cases. Not all 150 vulnerabilities were converted into a test case for multiple reasons. One of them is that several vulnerabilities are similar. For example, despite occurring in different projects, CVE-2023-50965, CVE-2021-46901, and CVE-2023-49287 are so similar that it would make little sense to create three different test cases for each of them. In addition, there were multiple cases where a similar vulnerability occurred in a single project in 2 or more places, and each got its unique CVE number. For instance, CVE-2023-38632 and CVE-2023-40296 happened in the same project, and their nature is the same. The difference is that one occurred in the User Datagram Protocol and the other in the Transmission Control Protocol. Another reason for skipping some vulnerabilities is complex code logic and the inability to extract the vulnerability logic accurately. In cases like CVE-2022-47663, where the line with vulnerability is known due to the stack trace provided by the CVE, it is apparent where the vulnerability happened. Nevertheless, due to the complex logic that leads to this vulnerability, it required too much time to extract the vulnerability while preserving its essence.

The test suite also contains two test cases that feature vulnerabilities discovered during the writing of this thesis. One of them occurred in the Linux driver for Razer devices called openrazer [67] and received a CVE number of CVE-2025-32776. Another occurred in an open-source multimedia framework called GPAC [68] and as of the time of writing this thesis, it has not received a CVE number, thus only the opened and already resolved GitHub issue can be shared [69]. Both vulnerabilities were accidentally found when reviewing and extracting the CVEs logic into test cases. The addition of these two accidentally found vulnerabilities makes the total number of test cases in a test suite equal 115.

Each test case in the created test suite is a single-file program that can be compiled and executed without additional data, as each test case uses only C/C++ standard libraries. All 115 test cases contain vulnerability logic extracted from analyzed CVEs. The mapping between CVE and a test case is preserved as each test case includes not only the CVE number of its original vulnerability but also additional metadata that refers to the original project version, file, and function. For the ease of use, all test cases were made compatible with benchexec [6], so that whole test suite could be executed with one command. The benchexec vulnerability classification with the number of test cases in each category is shown in Table 10.

Table 10. Test suite benchexec classification with test case count

Category/Property	Number of test cases
Valid memsafety	100
Integer overflow	17
Null dereference	8
Undefined behavior	2

The sum of all test cases in all categories in Table 10 is greater than the total number of test cases in a test suite. This comes from the fact that one test case may have more than one vulnerability category associated with it. For example, integer overflow often leads to invalid pointer dereference, making a single test case have two different vulnerabilities.

After the test suite was created, some test cases were patched and included in the final test suite. This was done for a couple of reasons. First, it enables computation of additional statistical measurements. Patched test case versions introduce False Positives (FP) and True Negatives (TN) that would otherwise not be applicable. This is so as in a test suite where each test case has a vulnerability, FP and TN cannot happen, as they both assume that the test case does not contain a vulnerability. Secondly, the introduction of patched test cases gives a more in-depth overview of static code analyzer tools used for analysis. With nonvulnerable versions of test cases, tools cannot blindly output an error for each test case and instead must produce a result based on their analysis. The total number of test cases patched and added to the test suite is 36, bringing the total number of test cases to 151. During the creation of patched version test cases, official vulnerability fixes have been utilized whenever possible.

3.3.1 International Competition on Software Verification (SV-COMP)

Most of all the extracted vulnerabilities were contributed to SV-COMP to be used in a static code analysis competition. Contribution required test cases not only to be compatible with benchexec [6] but also to include custom functions that would introduce randomness to the test cases. In the created test suite randomness was often achieved by utilizing program command line arguments, but SV-COMP supports functions like __VERIFIER_nondet_char and __VERIFIER_nondet_int. These two functions return a random character and a random integer, respectively. Because usage of such functions is encouraged, most test cases have been slightly modified to rely on this type of randomness introducing functions, rather than command line arguments. Overall, the test cases remained similar, with the only difference

being the input acquisition and all the differences coming from the input acquisition logic changes.

Since the SV-COMP scope is smaller than the scope of this thesis, not all created test cases were contributed. More specifically, SV-COMP does not support the C++ programming language. In addition, not all memory corruption vulnerabilities considered in this thesis are verified as part of the competition. More specifically, such vulnerability categories as undefined behavior and null deference are not part of the SV-COMP verification contest. Because of these differences in scope, the number of vulnerable test cases contributed was 83 out of 115 overall vulnerable test cases. In addition, out of 36 patched test cases, only 30 could be contributed due to the same limitation in scope. In the end, the total number of test cases contributed to SV-COMP was 113 [70].

3.4 Patterns

Despite coming across multiple similar vulnerabilities, most extracted vulnerabilities were still unique, and each had its own logic. Therefore, the answer to the research question this section attempts to answer does not have a concrete single answer. Ultimately, the answer to the research question is a test suite that can be found in a GitHub repository [66]. The test suite is the answer as it contains different examples of vulnerabilities that have happened recently in a simplified way that can be easily understood. Nevertheless, it could be insightful to review two patterns that very often lead to severe memory violations and that have frequently occurred in reviewed vulnerabilities.

3.4.1 Improper Bound Checking

This pattern is as simple as it sounds, meaning that before dereferencing a pointer, its bounds are not properly checked. An example of that pattern can be observed in the found vulnerability in GPAC software [69]. In that particular instance, a buffer access out of bounds occurred in a while loop, and the buffer accessed out of bounds was called *data*. Before the while loop, a condition checked for the variables *pos* and *descs_size* total sum to be at most equal to the buffer *data* size. Such a condition made the last verified valid buffer offset equal to *pos* + *descs_size* – 1. This is so because the buffer with length of *descs_size* last valid offset is *descs_size* – 1, as the buffer's first element is at position 0. Next, in the while loop condition, it was checked for *d_pos* + 1 to be less than *descs_size*. Such a condition made the last verified valid offset to the *data* buffer to be *pos* + *d_pos* + 1. This is because the following equation holds:

$$pos + desc_size - 1 \geq pos + d_pos + 1 \; if \; d_pos + 1 < descs_size$$

The vulnerability occurred in the loop where the buffer was accessed with an offset that exceeded the last verified valid offset. Such vulnerabilities often come from assumptions that developers make when writing code. Even in the case of the GPAC project, the vulnerability would have never occurred if the processed data had not been malformed. In case of valid input, the program worked correctly, and the edge case with malicious input submitted was not given enough thought.

3.4.2 Integer Overflow Leading to Buffer Overflow

Another pattern frequently observed in analyzed vulnerabilities is a buffer overflow caused by a prior integer overflow. While not every integer overflow leads to memory corruption, it is still a common scenario. Found vulnerability in openrazer [67] software with an assigned vulnerability number of CVE-2025-32776 can be taken as an example of such a pattern. In openrazer a vulnerability was found while processing a buffer filled with Red, Green, and Blue (RGB) data. The buffer data consisted of rows where each row started with a start and stop column that showed the number of bytes that the row consisted of. The problem was that the row length calculation was inconsistent between the two functions. The formula for calculating row length in both functions was as follows:

$$row\ length = \big((stop\ column + 1) - start\ column\big) * 3$$

Both stop and start columns were unsigned character variables whose value was previously read from the same buffer being processed. In the first function, the computation result was stored in an unsigned character variable, and in certain cases, the result could easily overflow. For instance, if the stop column is 85 and the start column is 0, the computed row length is 258. However, when stored in an unsigned character data type, this value overflows and becomes 2. After row length computation, the buffers' remaining length was checked to ensure it was sufficient to read the entire row, based on the overflowed computed length. After verifying the buffer remaining length, another function was called where the row length was again calculated. Calculation was done on the same stop and start column values, but the result was stored in an unsigned integer type variable. As the buffer size was assumed to be validated earlier in a call chain, it was not done a second time. Under these circumstances, the code accessed the buffer based on the correctly computed row length, leading to out-of-bounds access.

3.5 Statistics

Some statistics about the original projects were also gathered during test suite creation and vulnerability analysis. Namely, file and function lengths where the vulnerability occurred were recorded. During this research, the average length of functions where vulnerabilities occurred totaled 133 lines. The shortest function was six lines of code, and the longest was 2947. The lengths of all analyzed functions containing vulnerability are shown in Figure 6.

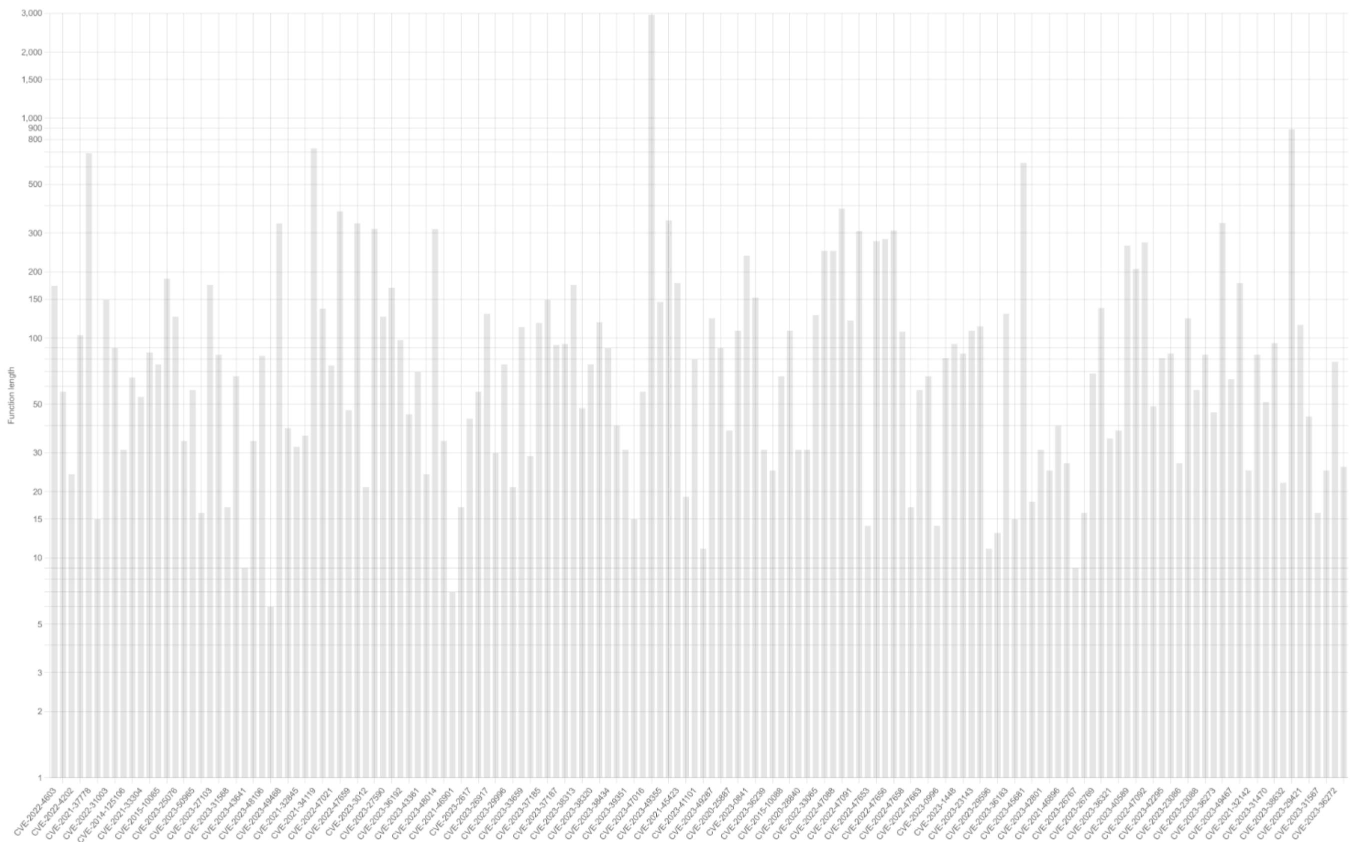

Figure 6. Length of functions containing vulnerability

As for files where vulnerabilities have been observed, the average number of lines of code was 2755. The shortest file was only 79 lines of code, and the longest file was 13026 lines of code. The lengths of all analyzed files containing functions with vulnerability are shown in Figure 7.

The numbers are much smaller for the extracted test suite due to the simplification and removal of excessive business logic. The average number of lines in a file in a test suite is 79, with a minimum of 17 and a maximum of 254. As for functions containing vulnerability, the average number of lines is 28, with a minimum of only three and a maximum of 166.

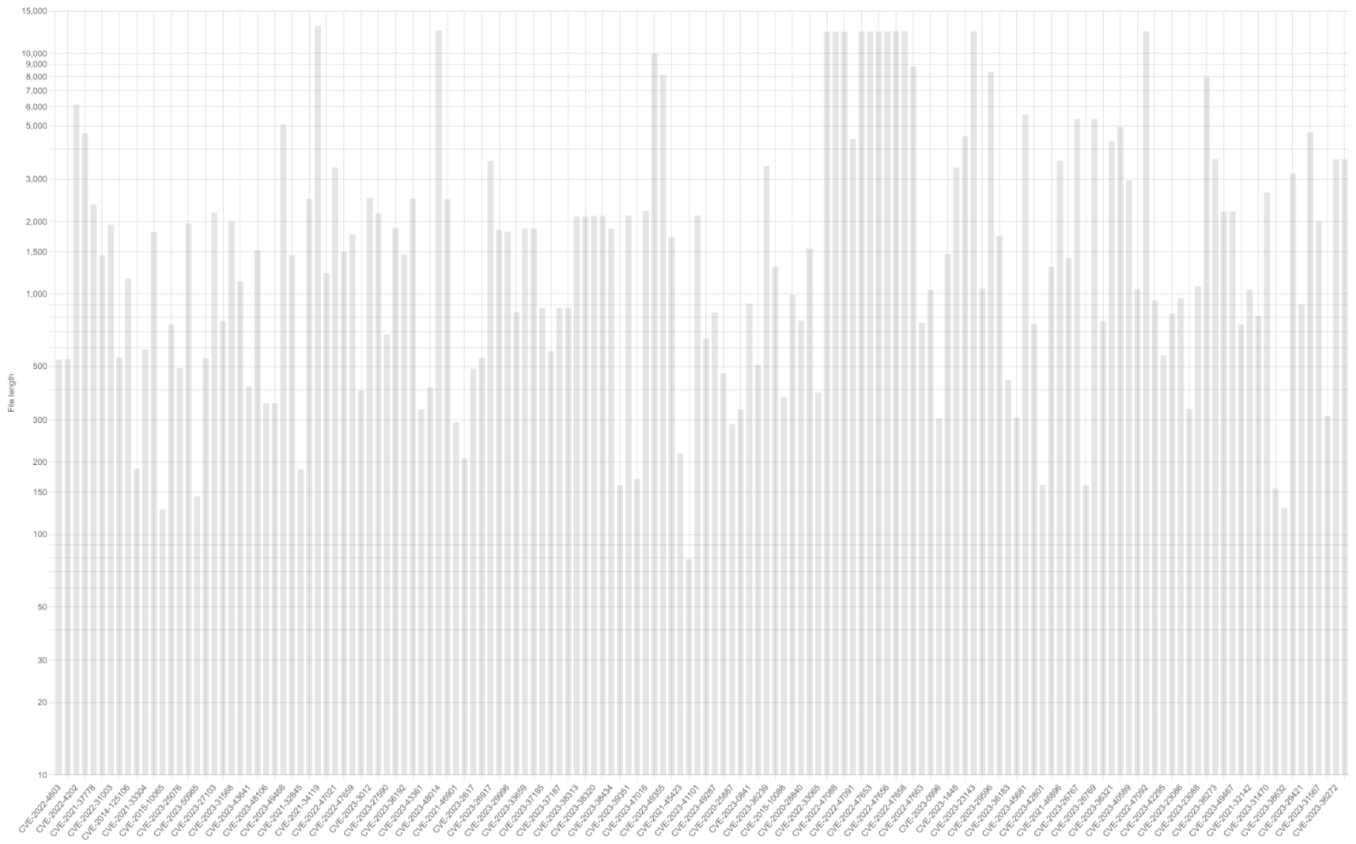

Figure 7. Length of files containing vulnerability

4. Static Code Analysis of CVEs

This section presents static code analysis results of previously selected CVEs. In addition, the section explains how vulnerabilities were analyzed, what tools were utilized, and what configuration was used for each tool.

4.1 Tools from the Literature Review

Tools reviewed in the literature review section, although accessible, were not used for vulnerability analysis in this research. The reason is simply the author's inability to configure the reviewed tools. All the tool configuration steps taken can be found in Appendix 4, with a summary of each tool's errors also present in Table 11. Tools configuration was attempted on a fresh install of a Windows Subsystem for Linux (WSL) image of Ubuntu 22.04 that has preinstalled Python version 3.10.12, as well as a WSL image of Ubuntu 20.04 that has preinstalled Python version 3.8.10. In both cases, before trying to configure programs, the following steps were taken:

- apt package lists were updated (apt update);
- pip3 was downloaded (apt install python3-pip);
- the build-essential package was installed (apt install build-essential).

Table 11. Reviewed tools configuration errors

Tool	Error
CorCA [58]	Runtime error
IncreLux [59]	Build errors
Lightweight Shape Analysis Based on Physical Types [60]	Can not consistently analyze test cases and randomly throws errors
DeepWukong [61]	Python dependencies conflict
SyzRisk [62]	Utilizes fuzzing as well as requires per-project configuration
NDI [63]	Tool cannot detect even the most basic bugs
IPPO [64]	Runtime error
MVD [65]	Absence of documentation for tool configuration

4.2 Tools Utilized for Static Code Analysis

The main criterion for selecting static code analyzers was for them to detect memory corruption vulnerabilities and support both C and C++ source code. In total, six static code analysis tools were used during program analysis. All tools used and their respective versions are presented in Table 12.

Table 12. Tools used for vulnerability analysis

Static code analyzer	Version
Clang [71]	19.0.0
Cppcheck [72]	2.7
Ikos [73]	3.3.0
Infer [74]	1.2.0
GCC [75]	14.2.0
Symbiotic [76]	8.0.0

Only five of these tools were used to analyze files in real-world projects. The reason is that Symbiotic needs an entry point in terms of a main function in C/C++ programs. Because the analysis was not done on the whole project but only on a single file that contains vulnerability, the main function was mostly not present, making Symbiotic analysis impossible.

A single file analysis instead of a complete project analysis was performed, as only a single vulnerability in the project was reviewed. The whole project analysis would be justified in case the goal is to find vulnerabilities in a project. As this research was only interested in a single function, there was no need to analyze the whole project. While single file analysis may have incurred a higher number of diagnostics produced by the tools, such a decision was still made mainly to simplify the analysis process and reduce time consumption.

4.2.1 Tools Configuration

When deciding what flags to use with a particular static code analyzer, the following two aspects were prioritized:

- Static code analyzers should analyze only file with vulnerability and not the whole project;
- Analyzers should be configured so that all checkers relevant to memory safety and integer overflow are turned on.

Other than the abovementioned aspects, some flags were also added to reduce the number of diagnostics given by the static code analyzer. For instance, Symbiotic flag '--malloc-never-fails' or Infer flag '--pulse-unsafe-malloc' are used to silence diagnostics about the malloc function potentially failing to allocate memory and returning a NULL pointer. The addition of such flags reduced the number of diagnostics that may not be worth looking at, making the diagnostics results more manageable. All used commands for each static code analysis tool can be found in Appendix 3 of this thesis.

4.2.2 Tools Produced Diagnostics Classification

Different tools for static code analysis have slightly different classifications of diagnostics they produce. In this thesis, all diagnostics were classified only into three groups: errors, warnings, and notes. While most of the selected static code analyzers had the same output diagnostics classification, Cppcheck had its own diagnostics classification. Because diagnostics from different tools were gathered and analyzed, there was a need to standardize the diagnostics that Cppcheck produced to match the chosen classification schema. Such standardization was done by mapping categories that Cppcheck outputs to the chosen classification. Cppcheck diagnostics classification mapping can be found in Table 13.

Table 13. Mapping of Cppcheck diagnostic

Chosen diagnostic classification	Cppcheck diagnostic classification
Error	- Error - Portability
Warning	- Warning
Note	- Style notes - Information - Performance

4.3 Real-world Projects Analysis

Before presenting the concrete numbers of the analysis, it would be appropriate first to talk about how the analysis was conducted. Before analyzing a project’s source code containing a vulnerability, it was downloaded and compiled. Afterwards, a file that contained a memory corruption vulnerability was tested with five static code analyzers. In case a memory corruption vulnerability required multiple file analysis, then two or more files were analyzed together using a preprocessor include directive that was added into the files under analysis. Cases where

it was utilized are, for example, a use-after-free vulnerability, where one function frees a pointer and another function dereferences it. If two functions were in different files, they were analyzed together to ensure static code analyzers had enough context to find every vulnerability. In addition, as Ikos required an entry point, a vulnerable function was used as the program's entry point. In case vulnerability could be detected only by starting analysis from a higher call stack function, that function was given to Ikos instead of the function where the memory vulnerability occurs. The most common pattern was a single file analysis, as usually a single file had enough context to detect a memory corruption vulnerability. For every exception that required multi file analysis, there is a note in the GitHub repository [66] where the analysis was conducted, specifying what has been added and in which file. In addition, the repository also contains the exact commands used for each file analysis.

While the number of CVEs that have been reviewed is 150, only 147 have been analyzed using static code analysis tools. The reason for that comes from research limitations and, more specifically, from the fact that all analyses were conducted not on a native Linux machine but rather on a Windows subsystem for Linux (WSL). While it is still possible that those three projects could have been compiled, it was considered an excessive waste of time to continue trying to configure and compile those projects.

4.3.1 Results

Arguably, the most critical metric in this research is the number of successfully detected vulnerabilities. While it is indeed important, there are also some other metrics that should be considered. For instance, the total number of diagnostics that a static code analysis tool outputs. While it is undeniably good if a static code analyzer manages to find a vulnerability, it is also vital that it does not output too many diagnostics. Otherwise, output reviewers can miss important diagnostics about vulnerability. All metrics from analyzing real-world projects can be found in Table 14. The formula used to calculate the True Positive Rate (TPR) is TP / (TP + FN).

Table 14. Real-world projects analysis results

Tool	Failed	Missed (FN)	Found (TP)	Found in the first 5 diagnostics	Total notes	Total warnings	Total errors	TPR
Clang [71]	1	141	5	4	38	1342	17	0.0342
Cppcheck [72]	4	139	4	4	455	328	191	0.0280
Ikos [73]	19	39	89	6	0	36229	66	0.6953
Infer [74]	2	140	5	4	0	22	1424	0.0345
GCC [75]	1	143	3	2	1257	2380	13	0.0205

Table 14 shows that the number of vulnerabilities found in projects does not exceed 5 for every tool except for Ikos. While Ikos' performance may seem impressive, it must also be noted that Ikos outputted the greatest number of diagnostics. In fact, the number of diagnostics outputted by Ikos has far surpassed that of all other tools combined and is roughly equivalent to the sum of diagnostics outputted by all other tools (7467) multiplied by five (37335). In the table, the True Positive Rate was calculated without considering the total number of diagnostics. This comes from the fact that multiple diagnostics could refer to the same vulnerability. In addition, not all diagnostics that do not refer to the analyzed vulnerability can be considered false positives. Because conventional metrics like TPR operate on Boolean result values and because fair analysis should also consider the number of metrics generated, a Diagnostic Efficiency (DE) metric is introduced as part of this study. DE is computed by taking all diagnostics that refer to vulnerabilities and dividing this number by the total number of diagnostics produced during analysis (TP / (total notes + total warnings + total errors)). Although DE cannot be considered completely trustworthy due to the previously mentioned reasons, it can still be useful, as it indicates the percentage of all diagnostics that refer to actual vulnerabilities. For Ikos DE equals only to 0.0025 (89 / 36295), meaning that only 0.25% of all produced diagnostics referred to actual vulnerabilities. While this is not the lowest DE of reviewed tools, as GCC scored 0.08%, other tools achieved higher scores, with the highest score of 0.41% achieved by Cppcheck.

One question that could also be answered based on gathered statistics is whether there is a correlation between the number of lines of code in a function and the successful identification of a vulnerability. Such a correlation is shown in Figure 8, and from the data gathered, it is impossible to conclude whether such a correlation exists. The slight drawback of the whole

research and the correlation graph that could not be avoided or predicted is that the analysis result data is very much influenced by a single tool. Because of such high dependency on a single tool, the correlation results may not be very objective. Nevertheless, the study found no connection between the number of lines in a function and the vulnerability detection rate.

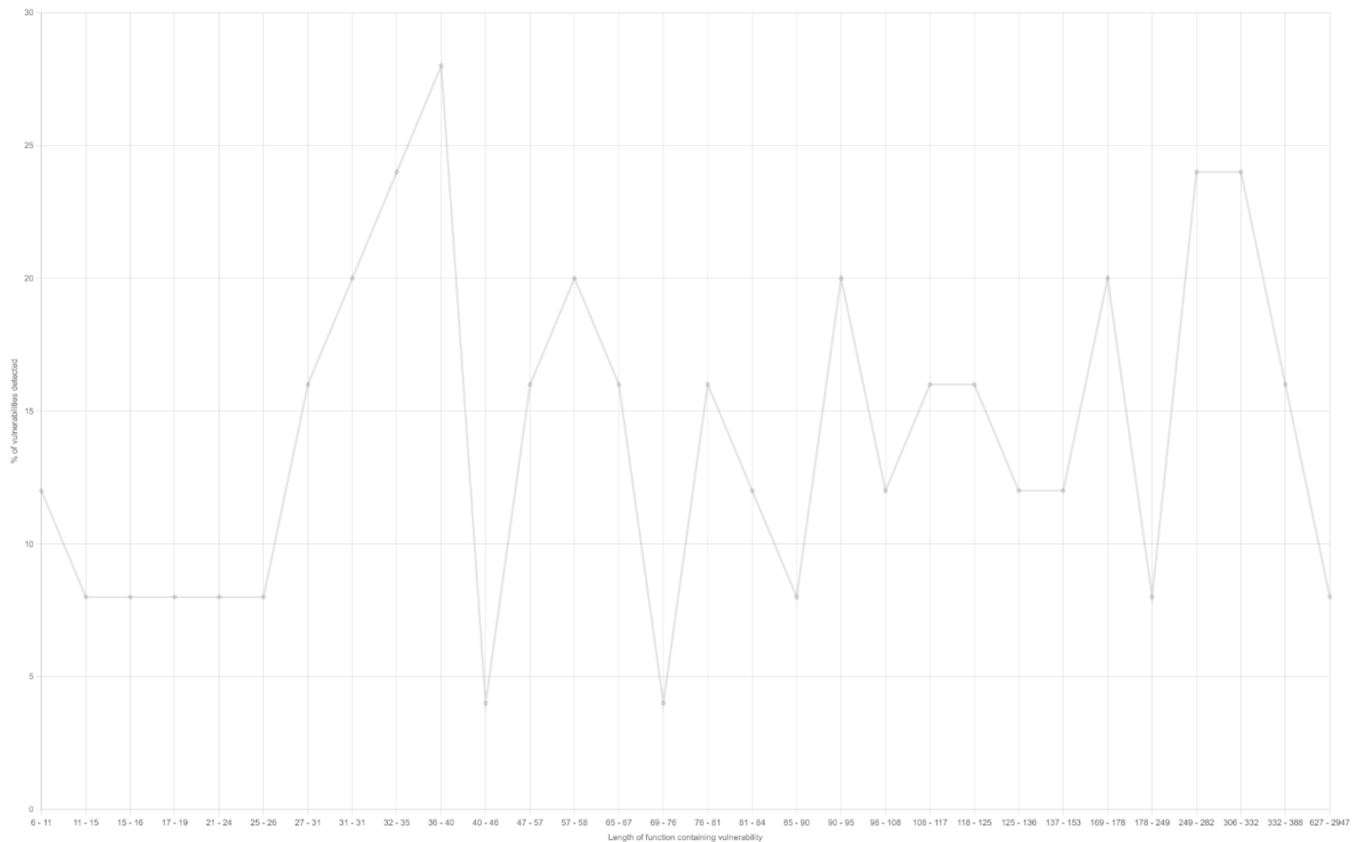

Figure 8. Percentage of successfully detected vulnerabilities relative to function length in real projects

5. Static Code Analysis of the Test Suite

This section is a logical continuation of the previous section, and it contains analysis results of the test suite created as part of this research. The same tools were utilized for test suite analysis with the same configuration as those used for real-world project analysis. During the analysis, both Ikos and Symbiotic were given the main function as a program entry point.

5.1 Results

The test suite also includes patched test cases that were analyzed, with the results categorized as either True Negatives (TN) when no vulnerability was detected and vulnerability does not exist, or false positives (FP) when the test was incorrectly flagged as vulnerable. It is worth mentioning that the result classification was done based on whether the result analysis refers to the line where the vulnerability was initially happening before the patch was applied. This means that if, after patching a vulnerability, a static code analyzer still threw diagnostics that the previously vulnerable line (and after patching, the benign line of code) still contains vulnerability, the result was categorized as a false positive.

Overall, the presence of patched versions makes it possible to assess the tools' accuracy when detecting vulnerabilities. While it is possible to calculate such widespread relative numbers as sensitivity, precision, accuracy, and many more using gathered statistics, it is not done as part of this research. The reason is that the test suite does not have vulnerable and patched test cases in equal numbers. Such a skewed dataset would result in inappropriately calculated relative numbers that would have to be interpreted differently from their standard meaning. To avoid confusion in this research, percentages are used to reason about the tools' performance. All gathered metrics from analyzing a test suite can be found in Tables 15 and 16.

Table 15. Test suite analysis results

Tool	Failed	Missed (FN)	Found (TP)	False (FP)	True (TN)	Found in the first 5 diagnostics	Total notes	Total warnings	Total errors
Clang [71]	0	85	30	1	35	30	1	77	2
Cppcheck [72]	0	111	4	0	36	4	4	5	1
Ikos [73]	0	25	90	22	14	44	0	2935	22
Infer [74]	0	98	17	1	35	17	0	0	95
GCC [75]	0	95	20	1	35	19	18	151	0
Symbiotic [76]	16	47	53	0	35	53	0	0	61

Table 16. Test suite analysis results

Tool	DE	TPR (TP / (TP + FN))	FPR (FP / (FP + TN))	TPR - FPR + 1
Clang [71]	0.4054	0.2609	0.0278	1.2331
Cppcheck [72]	0.4000	0.0348	0	1.0348
Ikos [73]	0.0402	0.7826	0.6111	1.1715
Infer [74]	0.2237	0.1478	0.0556	1.0923
GCC [75]	0.1563	0.1739	0.0278	1.1461
Symbiotic [76]	0.8689	0.5300	0	1.5300

While the True Positive Rate shows a percentage of total vulnerable test cases successfully identified, the False Positive Rate (FPR) shows a percentage of test cases incorrectly flagged as vulnerable. In real project analysis, the usage of TPR was inevitable as it is probably the only measure that could be calculated using the gathered data. In test suite result analysis, TPR is again used to avoid mixing statistics from vulnerable and patched versions. TPR operates only on data from vulnerable test cases, and FPR operates on data from patched test cases. Such percentages can accurately reflect research results while avoiding confusion related to different number of test cases with and without vulnerability. Finally, another score combines both TPR and FPR into a single number that allows for the comparison of tools with each other. The score formula utilized in this research is TPR – FPR + 1, and its possible range of values is [2, 0], with 2 being the best result and zero being the worst. A score of 2 means that TPR is 1, or that 100% of vulnerable test cases were correctly identified, and FPR is 0, meaning that none of the patched test cases were incorrectly flagged as vulnerable.

From the numbers, it can be concluded that while the vulnerability detection rate (TPR) has improved for all tools compared to real-world project analysis, it is still relatively low. Ikos managed to find an impressive 78.26% of all vulnerabilities, and Symbiotic identified 53%. Other static code analysis tools found considerably fewer vulnerabilities, with the lowest percentage being only 3.48%. While Ikos has identified the most vulnerabilities, it once again has produced the most diagnostics as well as the greatest number of false positives. In fact, only 4.02% of Ikos produced diagnostics when analyzing vulnerable test cases referred to actual vulnerabilities. Without regard to the number of diagnostics produced, judging purely by the number of identified vulnerabilities and false positives, Symbiotic can be considered the best-performing tool of this research. With 26.09% of correctly identified vulnerabilities and only 2.78% of false positives, Clang is second. Finally, with an impressive 78.26% of correctly

identified vulnerabilities and a disappointing 61.11% of falsely identified benign test cases, Ikos can be considered third.

In addition to analysis results, from Table 16, it can also be seen that despite the best efforts to preserve the vulnerability essence and context, the percentage of found vulnerabilities in the test suite is greater than that in real projects. While it may not be possible to give a complete list of reasons why it happened, some of those reasons could be:

- Complete program analysis (tools had more context);
- Smaller program leading to a reduced number of states and variable constraints;
- Hardcoded malicious input.

The author tends to believe that the main reason is hardcoded malicious inputs. There are various ways in which test cases receive information for processing. Test cases may receive information from the command line, by reading a file, or from a network socket. When analyzing real projects, there were a lot of cases when vulnerability occurred during information processing. In those cases, the source of information was not important as it did not influence vulnerability in any way. While extracting such vulnerabilities, the data acquisition was often omitted, mainly to simplify the test case code and keep only the code related to the vulnerability. Because of this simplification, many test cases directly contain a malicious input, making it easier to find a vulnerability.

In the end, the correlation between lines of code in a function and a successful vulnerability identification can also be studied in the extracted test suite. While a single tool heavily influenced the real project correlation of function length with vulnerability discovery rate, the test suite results are different. In test suite analysis results, the distribution is more uniform, which makes the graph more dependent on each tool. The graph itself can be seen in Figure 9. Again, there is no clear answer that could be derived from the graph, so the successful vulnerability detection correlation with function length will remain unanswered in this research.

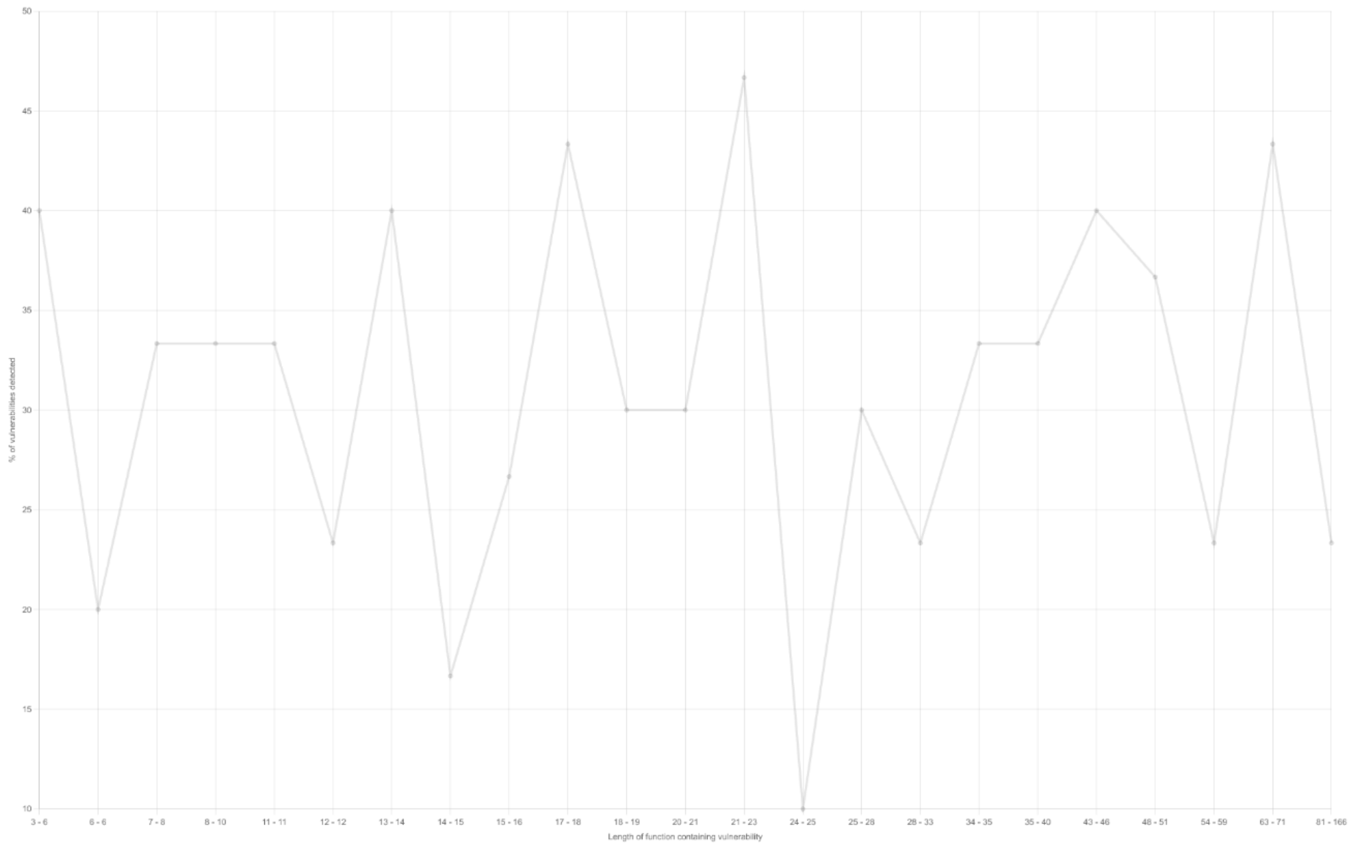

Figure 9. Percentage of successfully detected vulnerabilities relative to function length in a test suite

5.2 SV-COMP

Before the contribution, all 113 test cases were tested with 10 different static code analyzers that previously participated in the SV-COMP competition. All tools used have not been built from source code, but rather each tool's binary was used, which was submitted to SV-COMP [77]. Moreover, each tool's configuration and flags were the same as those used during the SV-COMP competition. Tools used, as well as their respective versions and execution flags, can be found in Table 17.

Table 17. SV-COMP test suite analysis tools

Tool	Version	Execution flags
CPAchecker [78]	4.0	--svcomp24 –benchmark –heap 10000m –timelimit 900 s
PredatorHP [79]	3.1415	
Ultimate Automizer [80]	0.3.0-dev-d790fec	--full-output
Ultimate Kojak [80]	0.3.0-dev-d790fec	--full-output
Ultimate Taipan [80]	0.3.0-dev-d790fec	--full-output
Bubaak [81]	0.9.2-0d2926	--sv-comp -timeout 120 -sv-comp-witness witness.graphml
CBMC [82]	cbmc-5.70.0-121-g4f69955d00	--graphml-witness witness.graphml
cpa-bam-smg [78]	svcomp23	-svcomp22-bam-smg -disable-java-assertions -heap 10000m -setprop counterexample.export.graphml=witness.graphml -setprop cpa.arg.proofWitness=witness.graphml -setprop termination.violation.witness=witness.graphml -setprop counterexample.export.compressWitness=false -setprop overflow.config=/usr/svcomp/svcomp25--overflow.properties
ESBMC-kind [83]	7.7.0 64-bit x86_64	-s kinduction
Symbiotic [76]	10.0.0-rc1	--sv-comp

5.2.1 Results

Contrary to how extracted test cases were each manually tested, the SV-COMP test suite was automatically verified utilizing the benchexec [6] framework. In addition, unlike previous test suite analyses, modified SV-COMP test suite verification always produced a single diagnostic. As such, the number of diagnostics and their efficiency are not applicable to this section. In total, two categories were contributed to SV-COMP, but only results from analyzing the invalid dereference category were recorded for PredatorHP [79] and cpa-bam-smg [78]. This was done because these tools do not officially participate in the integer overflow verification as part of SV-COMP. This led to a decision to exclude them from analyzing that category in this research as well. The test suite analysis results can be found in Table 18.

Table 18. SV-COMP test suite analysis results

Tool	All results	Correct results	Incorrect results	Unknown/Error
CPAchecker [78]	113	12 (10.62%)	2 (1.77%)	99 (87.61%)
PredatorHP [79]	107	12 (11.21%)	1 (0.93%)	94 (87.85%)
Ultimate Automizer [80]	113	22 (19.47%)	5 (4.42%)	86 (76.11%)
Ultimate Kojak [80]	113	22 (19.47%)	5 (4.42%)	86 (76.11%)
Ultimate Taipan [80]	113	22 (19.47%)	4 (3.54%)	87 (76.99%)
Bubaak [81]	113	50 (44.25%)	2 (1.77%)	61 (53.98%)
CBMC [82]	113	50 (44.25%)	0 (0%)	63 (55.75%)
cpa-bam-smg [78]	107	12 (11.21%)	3 (2.80%)	92 (85.98%)
ESBMC-kind [83]	113	46 (40.71%)	0 (0%)	67 (59.29%)
Symbiotic [76]	113	44 (38.94%)	2 (1.77%)	67 (59.29%)
Average		25.96%	2.14%	71.90%

A unique trait of the composed test suite is that it is based on vulnerabilities from real-world projects. As such, it may be interesting to compare the analysis results of the created test suite with the results that the tools achieved when analyzing the official SV-COMP test suite. In Table 19, the official results of selected SV-COMP tools are presented [84]. As with the created test suite analysis, most tool results are presented as a combination of two categories. The only tools whose results are taken from a single category are PredatorHP [79] and cpa-bam-smg [78]. This is because these two tools did not participate in integer overflow verification in SV-COMP.

Table 19. SV-COMP official competition verification results

Tool	All results	Correct results	Incorrect results	Unknown/Error
CPAchecker [78]	10323	7354 (71.24%)	6 (0.06%)	2767 (26.80%)
PredatorHP [79]	2135	1823 (85.39%)	3 (0.14%)	306 (14.33%)
Ultimate Automizer [80]	10323	6169 (59.76%)	0 (0%)	4148 (40.18%)
Ultimate Kojak [80]	10323	4993 (48.37%)	1 (0.01%)	5324 (51.57%)
Ultimate Taipan [80]	10323	6143 (59.51%)	1 (0.01%)	4166 (40.36%)
Bubaak [81]	10323	6470 (62.68%)	3 (0.03%)	3693 (35.77%)
CBMC [82]	10213	6977 (68.31%)	27 (0.26%)	1076 (10.54%)
cpa-bam-smg [78]	2080	1680 (80.77%)	0 (0%)	398 (19.13%)
ESBMC-kind [83]	10323	8040 (77.88%)	6 (0.06%)	1305 (12.64%)
Symbiotic [76]	10323	6850 (66.36%)	7 (0.07%)	3286 (31.83%)
Average		68.03%	0.06%	28.31%

Worth noting is that while extracting information from official SV-COMP competition results, an additional category that was not seen when analyzing the composed test suite, called correct-unconfirmed, was present. This category was ultimately excluded from review and is not present in Table 19 results. Such a decision was made because it could not be decided whether to consider these results as unknown or concrete, and it was decided to eliminate them from the comparison completely. Because of that, when summing percentages in Table 19, the total may be slightly lower than 100%.

While the number of test cases in the created test suite does not nearly match the number in the official SV-COMP verification repository, some conclusions can still be made. For instance, one immediately noticeable aspect is how much lower the correct result percentage is between the analysis of the two test cases. Such a big difference may hint that test cases derived from real-world projects are more complex than mostly synthetic test cases. The difference may also be caused by the fact that the created test suite features new test cases that have never been seen before. While verification tool maintainers saw and could adjust their tool for test cases in the SV-COMP repository, the created test cases are entirely new. As such, the tools developers did not have a chance to account for new test cases. Regardless of the actual reason, the massive difference in both correct and incorrect results shows that even competition-grade tools struggle with certain test case analysis. Such results completely align with this thesis's goal, which was not only to determine the capabilities of static code analyzers but also to create a test suite that would facilitate further development of static code analyzers.

6. Conclusion

In the end, it can be concluded that while static code analysis can detect some vulnerabilities, it is not capable of being the ultimate solution to the memory corruption problem. Such a conclusion can be drawn from the overall vulnerability detection rate of both competition-grade tools as well as chosen widespread static analyzers. Nevertheless, static code analysis is still a major field that has its place in vulnerability detection. Proof of that is the fact that, even as part of this research, a couple of vulnerabilities were found utilizing static code analysis in open-source software.

Conducted research despite not proving a correlation between function length and vulnerability detection rate, still provides evidence that analysis of smaller chunks of code yields more precise results. Some studied approaches in the literature review have already utilized the source code partition in their analysis, and such a measure seems justified based on the research analysis results. Inspection of smaller pieces of code makes it possible to avoid known limitations of existing approaches, such as symbolic execution path explosion. Additionally, such analysis makes it possible to use various static code analysis tools, including competition-grade tools whose performance is superior to conventional static code analysis tools. One of the potential research continuation ideas is to develop a static code analyzer capable of identifying all vulnerabilities in the created test suite. Furthermore, another potential contribution that would bring considerable value to the field of static code analysis is to create a tool that would split the source code into smaller pieces for analysis.

This research has made several contributions to the field of static code analysis. The first theoretical contribution is an overview of techniques for detecting memory corruption vulnerabilities. After reviewing techniques, an analysis of modern vulnerabilities was carried out. During analysis, vulnerabilities were not only analyzed using a variety of static code analyzers, but their logic was also extracted, which led to the creation of a test suite based on real-world vulnerabilities. In addition, patched versions of many test cases were developed, utilizing the original project's patching strategies whenever applicable. As a final contribution, most created test cases were submitted to the International Competition on Software Verification (SV-COMP), where they will be analyzed as part of the contest. Together, these efforts provide both theoretical and practical results to support future research and tool development in the field of static code analysis.

References

[1] B. Lord, "The Urgent Need for Memory Safety in Software Products," 20 September 2023. [Online]. Available: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products. [Accessed 3 March 2024].

[2] T. W. House, "BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE," The White House, Washington, 2024.

[3] J. Bartlett, "Computer Architecture," in *Programming from the Ground Up*, 2003, pp. 7-19.

[4] S. Laszlo, M. Payer, T. Wei and D. Song, "SoK: Eternal War in Memory," University of California, Berkeley, 2013.

[5] "cvedetails," SecurityScorecard, [Online]. Available: https://www.cvedetails.com/. [Accessed 10 March 2024].

[6] sosy-lab, "benchexec," sosy-lab, 15 March 2025. [Online]. Available: https://github.com/sosy-lab/benchexec. [Accessed 15 March 2025].

[7] sosy-lab, "Competition on Software Verification (SV-COMP)," sosy-lab, [Online]. Available: https://sv-comp.sosy-lab.org/. [Accessed 15 March 2025].

[8] I. Xplore, "IEEE Xplore," IEEE, 7 May 2025. [Online]. Available: https://ieeexplore.ieee.org/. [Accessed 7 May 2025].

[9] Elsevier, "sciencedirect," Elsevier, 7 May 2025. [Online]. Available: https://www.sciencedirect.com/. [Accessed 7 May 2025].

[10] Elsevier, "scopus," Elsevier, 7 May 2025. [Online]. Available: https://www.scopus.com/. [Accessed 7 May 2025].

[11] A. f. C. Machinery, "ACM Digital Library," Association for Computing Machinery, 7 May 2025. [Online]. Available: https://dl.acm.org/. [Accessed 7 May 2025].

[12] metaxis, "International Database of Education Systematic Reviews," metaxis, 7 May 2025. [Online]. Available: https://idesr.org/. [Accessed 7 May 2025].

[13] ResearchGate, "ResearchGate," ResearchGate, 7 May 2025. [Online]. Available: https://www.researchgate.net/. [Accessed 7 May 2025].

[14] D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević and S. Ristić, "STATIC CODE ANALYSIS TOOLS: A SYSTEMATIC LITERATURE REVIEW," in *Proceedings of the 31st DAAAM*, Vienna, 2020.

[15] N. C. f. A. Software, "Juliet C/C++ 1.3," NIST, 1 October 2017. [Online]. Available: https://samate.nist.gov/SARD/test-suites/112. [Accessed 13 October 2024].

[16] NIST, "NIST Software Assurance Reference Dataset," NIST, [Online]. Available: https://samate.nist.gov/SARD. [Accessed 13 October 2024].

[17] D. S. Lab@ISU, "BugBench," Data Storage Lab@ISU, [Online]. Available: https://github.com/data-storage-lab/BugBench. [Accessed 13 October 2024].

[18] Y. Chen, Z. Ding, L. Alowain, X. Chen and D. Wagner, "DiverseVul: A New Vulnerable Source Code Dataset for Deep," RAID ’23, Hong Kong, 2023.

[19] Devign, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," Devign, [Online]. Available: https://sites.google.com/view/devign. [Accessed 13 October 2024].

[20] S. Lipp, S. Banescu and A. Pretschner, "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection," Association for Computing Machinery, South Korea, 2022.

[21] K. Goseva-Popstojanova and A. Perhinschi , "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology,* vol. 68, pp. 18-33, 2015.

[22] A. K. Kaur and R. Nayyar, "A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code," in *Procedia Computer Science 171*, New Delhi, 2020.

[23] D. V. J. J. Vishruti V. Desai, "Comprehensive Empirical Study of Static Code Analysis Tools for C Language," *International Journal of Intelligent Systems and Applications in Engineering,* vol. 10, no. 4, pp. 695-700, 2022.

[24] J. Malm, E. Enoiu, M. A. Naser, B. Lisper, Z. Porkolab and S. Eldh, "An Evaluation of General-Purpose Static Analysis Tools on C/C++ Test Code," in *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2022.

[25] F. S. Foundation, "Program Instrumentation Options," Free Software Foundation, [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#index-fsanitize_003daddress. [Accessed 29 October 2024].

[26] F.-J. Gao, Y. Wang, L.-Z. Wang, Z. Yang and X.-D. Li, "Automatic Buffer Overflow Warning Validation," *Journal of Computer Science and Technology,* vol. 35, pp. 1406-1427, 2020.

[27] A. Wilkerson , R. Muniz and P. Machado, "Towards a Technique to Detect Weaknesses in C Programs," in *Brazilian Symposium on Software Engineering (SBES '21)*, New York, 2021.

[28] S. Hong, J. Lee, J. Lee and H. Oh, "SAVER: Scalable, Precise, and Safe Memory-Error Repair," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, Seoul, Association for Computing Machinery, 2020, pp. 271-283.

[29] T. Murray, P. Yan and G. E. Ernst, "Compositional Vulnerability Detection with Insecurity Separation Logic," in *Formal Methods and Software Engineering*, Singapore, Springer, 2023, pp. 65-82.

[30] J. Inacio and I. Medeiros, "Effectiveness on C Flaws Checking and Removal," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, IEEE, 2022, pp. 33-34.

[31] A. Rocha, Z. Altahat, S. Alnaeli and B. Barcaskey, "Enhancing the Quality and Security of IoT Software Systems Using Cloud-Based Vulnerability Detector," in *Advances in Information and Communication*, Springer, Cham, 2022, pp. 919-927.

[32] W. Yaxin, L. Xiaoqing, W. Gaofei, T. Shijian, Z. Yajie and D. Ting, "Detecting use-after-free bugs in embedded C programs," *Xi'an Dianzi Keji Daxue Xuebao/Journal of Xidian University,* vol. 48, no. 1, pp. 124-132, 2021.

[33] D. Steinhöfel, "Symbolic Execution: Foundations, Techniques, Applications, and Future Perspectives," Springer Nature, Switzerland, 2022.

[34] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *IEEE Symposium on Security and Privacy*, San Jose, 2014.

[35] A. P. L. Ferreira, "Model Checking," in *2011 Workshop-School on Theoretical Computer Science*, Pelotas, IEEE, 2011, pp. 9-14.

[36] C. K. Roy and J. R. Cordy, A Survey on Software Clone Detection Research, Ontario: Queen’s University, 2007.

[37] M. I. Schwartzbach and A. Møller, "Dataflow Analysis with Monotone Frameworks," in *Static Program Analysis*, Aarhus, Aarhus University, Denmark, 2024, pp. 51-76.

[38] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou and W. Shi, "MVP: detecting vulnerabilities using patch-enhanced vulnerability signatures," in *29th USENIX Conference on Security Symposium (SEC'20)*, USA, 2020.

[39] C. Du, Y. Guo, Y. Feng and S. Zheng, "HotCFuzz: Enhancing Vulnerability Detection through Fuzzing and Hotspot Code Coverage Analysis," *Electronics,* vol. 13, no. 10, 2024.

[40] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, New York, 2020.

[41] G. Lee, D. Xu, S. Salimi, B. Lee and M. Payer, "SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer," in *ACM Asia Conference on Computer and Communications Security (ASIA CCS ’24)*, Singapore, 2024.

[42] N. Xiao, J. Zeng, Q. Yao and X. Huang, "A Method of Firmware Vulnerability Mining and Verification Based on Code Property Graph," in *Advances in Artificial Intelligence and Security*, Beijing, Springer, Cham, 2022, pp. 543-556.

[43] A. Piran, C.-P. Chang and A. M. Fard, "Vulnerability Analysis of Similar Code," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Hainan, 2021.

[44] S. Priya, Y. Su, Y. Bao, X. Zhou, Y. Vizel and A. Gurfinkel, "Bounded Model Checking for LLVM," in *2022 Formal Methods in Computer-Aided Design (FMCAD)*, Trento, 2022, pp. 214-224.

[45] H. Wei, L. Chen, X. Nie, Z. Zhang, Y. Zhang and G. Shi, "An Efficient Metric-Based Approach for Static Use-After-Free Detection," in *2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, Melbourne, 2022.

[46] Y. Iqbal, M. A. Sindhu and M. H. Arif, "Enhancement in Buffer Overflow (BOF) Detection Capability of Cppcheck Static Analysis Tool," in *2021 International Conference on Cyber Warfare and Security (ICCWS)*, Islamabad, 2021.

[47] L. Ferreirinha and I. Medeiros, "On the Path to Buffer Overflow Detection by Model Checking the Stack of Binary Programs," in *19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, Angers, 2024.

[48] J. Inacio and I. Medeiros , "CorCA: An Automatic Program Repair Tool for Checking and Removing Effectively C Flaws," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Dublin, 2023.

[49] Y. Zhai, Y. Hao, Z. Zhang, W. Chen, G. Li, Z. Qian, C. Song, M. Sridharan, S. V. Krishnamurthy, T. Jaeger and P. Yu, "Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel," in *Network and Distributed Systems Security (NDSS) Symposium*, San Diego, 2022.

[50] O. Nicole, M. Lemerre and X. Rival, "Lightweight Shape Analysis Based on Physical Types," in *Lightweight Shape Analysis Based on Physical Types*, Paris, Springer, Cham, 2022, p. 219–241.

[51] X. Cheng, H. Wang, J. Hua and G. Xu, "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network," *ACM Transactions on Software Engineering and Methodology,* vol. 30 (3), pp. 1-33, 2021.

[52] F. Gao, Y. Wang, T. Chen and L. Situ, "Static Checking of Array Index Out-of-Bounds Defects in C Programs Based on Taint Analysis," *International Journal of Software and Informatics,* vol. 11(2), pp. 121-147, 2021.

[53] Q. Zhou, Q. Wu, D. Liu, S. Ji and K. Lu, "Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Los Angeles, 2022.

[54] D. Liu, Z. Lu, S. Ji, K. Lu, J. Chen, Z. Liu, D. Liu, R. Cai and Q. He, "Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences," in *Proceedings of the 33rd USENIX Security Symposium*, Philadelphia, 2024.

[55] D. Liu, Q. Wu, S. Ji, K. Lu, Z. Liu, J. Chen and Q. He, "Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*, New York, 2021.

[56] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, X. Wu, C. Tao, T. Zhang and W. Liu, "Learning to Detect Memory-related Vulnerabilities," *ACM Transactions on Software Engineering and Methodology,* vol. 33 (2), p. 35, 2023.

[57] D. Hin, A. Kan, H. Chen and A. M. Babar, "LineVD: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th International Conference on Mining Software Repositories*, New York, Association for Computing Machinery, 2022, p. 596–607.

[58] iberiam, "CorCA," 11 November 2024. [Online]. Available: https://github.com/iberiam/CorCA. [Accessed 11 November 2024].

[59] seclab-ucr, "IncreLux," 11 November 2024. [Online]. Available: https://github.com/seclab-ucr/IncreLux. [Accessed 11 November 2024].

[60] O. Nicole, M. Lemerre and X. Rival, "Lightweight Shape Analysis based on Physical Types -- article and artifact," 16 September 2021. [Online]. Available: https://zenodo.org/records/5589489. [Accessed 11 November 2024].

[61] jumormt, "DeepWukong," 11 November 2024. [Online]. Available: https://github.com/jumormt/DeepWukong. [Accessed 11 November 2024].

[62] HexHive, "SyzRisk," 11 November 2024. [Online]. Available: https://github.com/HexHive/SyzRisk. [Accessed 11 November 2024].

[63] umnsec, "ndi," 11 November 2024. [Online]. Available: https://github.com/umnsec/ndi. [Accessed 11 November 2024].

[64] dinghaoliu, "IPPO," 11 November 2024. [Online]. Available: https://github.com/dinghaoliu/IPPO. [Accessed 11 November 2024].

[65] MVDetection, "MVD," 11 November 2024. [Online]. Available: https://github.com/MVDetection/MVD. [Accessed 11 November 2024].

[66] R. Amjaga, "memory-corruption-examples," 28 March 2025. [Online]. Available: https://github.com/DiRaltvein/memory-corruption-examples. [Accessed 28 March 2025].

[67] openrazer, "openrazer," openrazer, 21 April 2025. [Online]. Available: https://github.com/openrazer/openrazer. [Accessed 21 April 2025].

[68] gpac, "gpac," gpac, 21 April 2025. [Online]. Available: https://github.com/gpac/gpac. [Accessed 21 April 2025].

[69] DiRaltvein, "Heap-buffer-overflow in function gf_m2ts_process_sdt of media_tools/mpegts.c:795," gpac, 21 April 2025. [Online]. Available: https://github.com/gpac/gpac/issues/3180. [Accessed 21 April 2025].

[70] R. Amjaga, "Added 84 (+30) new tests related to memory safety," 12 May 2025. [Online]. Available: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1616. [Accessed 12 May 2025].

[71] "Clang: a C language family frontend for LLVM," [Online]. Available: https://clang.llvm.org/. [Accessed 15 March 2025].

[72] "Cppcheck," [Online]. Available: https://cppcheck.sourceforge.io/. [Accessed 15 March 2025].

[73] NASA, "ikos," NASA, [Online]. Available: https://github.com/NASA-SW-VnV/ikos. [Accessed 15 March 2025].

[74] Facebook, "A tool to detect bugs in Java and C/C++/Objective-C code before it ships," Facebook, [Online]. Available: https://fbinfer.com/. [Accessed 15 March 2025].

[75] "GCC, the GNU Compiler Collection," [Online]. Available: https://gcc.gnu.org/. [Accessed 15 March 2025].

[76] "symbiotic," symbiotic, [Online]. Available: https://github.com/staticafi/symbiotic. [Accessed 15 March 2025].

[77] sosy-lab, "Tools for Formal Methods: Tools," sosy-lab, 18 April 2025. [Online]. Available: https://fm-tools.sosy-lab.org/. [Accessed 18 April 2025].

[78] D. Beyer, P. Wendler and E. Keremoglu, "CPAchecker," [Online]. Available: https://cpachecker.sosy-lab.org/. [Accessed 15 March 2025].

[79] T. Vojnar, K. Dudka, P. Peringer, P. Muller, V. Šoková, M. Kotoun and O. Kinst, "Predator Hunting Party," [Online]. Available: https://www.fit.vut.cz/research/group/verifit/public/tools/predatorhp/. [Accessed 15 March 2025].

[80] D. Dietsch, D. Klumpp, F. Schüssele and M. Heizmann, "Ultimate," Ultimate, [Online]. Available: https://ultimate-pa.org/. [Accessed 15 March 2025].

[81] M. Chalupa, "Bubaak," 26 November 2021. [Online]. Available: https://gitlab.com/mchalupa/bubaak. [Accessed 15 March 2025].

[82] cprover.org, "CBMC," cprover.org, [Online]. Available: https://www.cprover.org/cbmc/. [Accessed 15 March 2025].

[83] S. &. S. V. Laboratory, "ESBMC," Systems & Software Verification Laboratory, [Online]. Available: https://ssvlab.github.io/esbmc/. [Accessed 15 March 2025].

[84] sosy-lab, "Results of the Competition," sosy-lab, 18 April 2025. [Online]. Available: https://sv-comp.sosy-lab.org/2024/results/results-verified/. [Accessed 18 April 2025].

[85] pyenv, "pyenv," pyenv, 14 February 2025. [Online]. Available: https://github.com/pyenv/pyenv. [Accessed 14 February 2025].

Appendix

I. Data Extraction Form

Author(s)	
Title and year	
Study objective	What is the problem the paper tries to solve or the goal.
Method name	
Method type(s) of memory corruption	What type(s) of memory corruption is the method capable of finding. Example: buffer/integer overflow, user-after-free, etc.
Method approach	Is the method static/dynamic, and a more specific approach classification like symbolic execution, fuzzing, or Abstract Syntax Tree (AST) analysis.
Method working principles	Description of how the method works.
Method tools used	What tools are used to employ the method. Example: Clang, GCC, KLEE, etc.
Method advantages	
Method limitations	
Method effectiveness	Any metrics on the method. Example: Precision, detection rate, false-positive rate, etc.
Method validation and evaluation	If the method is validated through experiments, mathematically, or using a test suite.
Method implementation availability	Whether the implementation is publicly accessible
Method notes	
Key findings	What are the main findings and contributions of the analyzed paper to the systematic literature review.

II. Systematic Literature Review Paper Summaries

An Efficient Metric-Based Approach for Static Use-After-Free Detection [45]

The paper presents an approach for finding use after free vulnerabilities. The main idea of the approach is to utilize a Code Property Graph (CPG) to perform control and data-sensitive analysis. Using such analysis, all feasible paths are found that can lead to a use-after-free vulnerability. Essentially, the algorithm traverses CPG and looks for a pair of statements that first free a memory region and then later use it (such a pair of statements is called a UAF pair). In addition to finding all use-after-free paths, each instance of vulnerability is given a controllability score that shows how likely the vulnerability is to be triggered in a real-world scenario.

Enhancement in Buffer Overflow (BOF) Detection Capability of Cppcheck Static Analysis Tool [46]

The paper presents an approach to increase the efficiency of detecting buffer overflow vulnerabilities. The idea is to check all potential places where buffer overflow can happen by utilizing a control flow sensitive reverse traversal of the Data Flow Graph (DFG). Source code statements that are checked are all lines in the source code that can cause a buffer overflow, such as dangerous API functions, array read/write operations, etc. For each potentially vulnerable statement, a reverse DFG traversal is executed to find the values of variables used in that statement. If DFG traversal can yield specific values, they are checked, and an error is generated if the values can lead to a buffer overflow. Otherwise, if traversal could not produce a concrete value, then a warning is generated.

Bounded Model Checking for LLVM [44]

The paper presents a novel approach for bounded model checking of LLVM-based programming languages. The novelty of this approach is that the program's source code goes through a series of transformations where each intermediate representation can be verified by using different verification conditions. In addition, to verify memory safety, the checker transforms pointers into fat pointers and extends memory with a shadow memory to store metadata. The approach was tested on the aws-c-common library, and the results showed that it is effective as it outperformed competitor tools in most of the tested categories.

On the Path to Buffer Overflow Detection by Model Checking the Stack of Binary Programs [47]

The paper presents a method for creating a model of a binary stack memory and validating it using a model-checking approach with predefined as well as user-provided security properties. The analysis starts by extracting the x86-64 assembly and control flow graph from the executable file. When data is extracted, a model of a program stack space is created and verified against the security properties. If any security property is violated, the vulnerability class can be deduced by correlating the failed security property with the vulnerability classes. In addition, when a security property is violated, the exact place in the binary code is found and reported. The exact vulnerable place is found by utilizing reverse-flow analysis of a counterexample generated as a response to a failed security property.

CorCA: An Automatic Program Repair Tool for Checking and Removing Effectively C Flaws [48]

The paper presents a way to resolve buffer overflow vulnerabilities in C/C++ code automatically. The whole process is divided into steps, and firstly, potential vulnerable places are found using static code analysis. The paper describes a new type of analysis that finds potentially vulnerable places by looking for specific C/C++ functions that are not memory safe (gets, scanf, sprintf, strcpy, memcpy, strcat, etc.). All occurrences of memory-unsafe functions are flagged as potentially vulnerable. Next, all potential vulnerable statements are extracted into separate executables, and the vulnerability presence is verified using fuzz testing. Extraction is done from the buffer declaration up to the potentially vulnerable line of code. Suppose fuzz testing confirms the existence of a vulnerability. In that case, that vulnerability gets automatically patched, and the patched code gets tested by fuzzing again to verify that the automatic fix has been generated correctly.

Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel [49]

The paper describes a bottom-up summary-based incremental analysis tool called IncreLux, whose primary purpose is to analyze the Linux kernel. The analysis is a summary-based analysis where a so-called summary is created for each function, specifying the caller-callee relationships. If the function did not change after new changes were introduced, then it is safe to reuse the summary that was previously generated for that function. When two functions whose summaries do not align are found, symbolic execution is utilized to find a feasible path to trigger a vulnerability.

Lightweight Shape Analysis Based on Physical Types [50]

The paper presents a way to detect memory violations by analyzing the program variable types and their interactions throughout the execution. The analysis relies on an abstract flow-insensitive heap representation that models the relationship between memory locations and pointers based on their physical types. After the heap abstract representation is constructed, all statements related to pointers are checked against it to determine whether interaction may potentially result in a memory violation. To reduce the number of false positives, flow-insensitive method points-to analysis is also used to get additional information about a memory region.

DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network [51]

The paper presents an approach to detect the top ten most common C/C++ Weakness Enumeration (CWE) vulnerabilities from 1. January 2017 to 20 July 2019. The approach relies on a deep graph neural network that utilizes the latest recently proposed graph neural architecture, which consists of a graph convolutional layer, a graph pooling layer, and a graph readout layer. The whole analysis is performed using a Program Dependency Graph (PDG) that is constructed using a Control-Flow Graph (CFG) and a Value-Flow Graph (VFG). First, the Program Dependency Graph (PDG) is sliced into smaller graphs starting from the point of interest until a fixed point is reached. Next, each slice node (that represents tokens of each statement) is vectorized using Doc2Vec, and vectors along with graph edges (that represent data- or control-dependence relation between two nodes) are provided to a deep graph neural network that then says whether code contains a vulnerability or not.

Vulnerability Analysis of Similar Code [43]

The paper describes a theoretical approach to vulnerability detection based on the similarity between two functions' source code. Two functions are compared, one containing a known vulnerability, and another that needs to be analyzed. If the two functions' code is similar, then there is a probability that the analyzed function contains the same vulnerability as the function it was compared to. The research analyzed vulnerabilities grouped by Common Weakness Enumeration (CWE) and their similarities. The similarities were calculated within CWE groups as well as within functions that belong to the same domain/category. Ultimately, it was concluded that 8 CWE categories out of 23 analyzed had function mean and median similarities above 50%.

Static Checking of Array Index out of Bounds Defects in C Programs Based on Taint Analysis [52]

The paper presents an approach to detect array out-of-bound access in an application using static code analysis and, more specifically, a tainted analysis. The approach first looks for all buffers accessed using a tainted index (an index not known during program compilation). After taint buffer access is found, the buffer's overall size is determined by retrieving it from the Abstract Syntax Tree (in case the buffer is stored on the stack) or performing a pointer analysis (in case the buffer is stored on the heap). When buffer length is known, backwards data flow analysis is executed with simple matching and constraint solving strategies to verify whether the index used to access the buffer is within the range of the buffer or not.

SyzRisk: A Change-Pattern-Based Continuous Kernel Regression Fuzzer [41]

The paper presents an approach to regression fuzzing of large applications (like kernels). The presented fuzzing utility, called SyzRisk, utilizes 23 different patterns to assign risk weights to modified functions. A function's weight level results from multiplying all risk levels of patterns that matched that function. The higher the weight of a function, the greater the chance that the function contains a vulnerability. After the modified functions are assigned their weights, fuzzing starts, where each input is also assigned a weight. The weight of each input is the average weight of all the functions visited during program invocation with that input. The higher the input's weight, the more likely it will be prioritized for testing in the following iterations and/or mutations.

A Method of Firmware Vulnerability Mining and Verification Based on Code Property Graph [42]

The paper presents an approach for detecting vulnerabilities in firmware using four different types of graphs. First Code Property Graph (CPG) is generated from the source code. Next, Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Dependency Graph (DDG) are extracted from CPG. The vulnerabilities can be discovered manually by analyzing source code and the traversal results on three extracted graphs (AST, CFG, and DDG).

Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs [53]

The article describes an approach to bug detection that utilizes different program path inconsistencies. The approach first constructs a control-flow graph and, using it, finds all paths that a function may have. Next, all distinct paths are compared to find if there are paths that result in a different state of variables after merging. For example, if the function has two distinct paths, and in one path a certain pointer is freed while in another path that same pointer is not freed. If an algorithm finds such inconsistencies between two paths in a function, it tries to find variables that could identify which path has been taken. If no such variable exists, then the function is reported as vulnerable. If a variable exists that can help distinguish between paths taken, then the algorithm looks further to see if the program recovers from inconsistencies before using an inconsistent variable.

Detecting Kernel Memory Bugs through Inconsistent Memory Management Intention Inferences [54]

The article presented an approach to finding use-after-free as well as memory leakage bugs through inconsistent memory management of caller and callee functions. Research outlines two types of memory management strategies: caller-based or callee-based. In caller-based memory management, the caller is responsible for the memory allocated in an invoked function. In callee-based memory management, the callee is responsible for freeing all allocated resources. When the caller and callee memory management strategies do not match, either a use-after-free vulnerability or a memory leak occurs. The paper utilizes Control Flow Graphs (CFG) along with Call Graph (CG) to detect what memory management strategies functions are using. For complex cases, an LLM is also used to determine which memory management strategy function is used. After detecting memory management strategies of all functions, the algorithm looks for inconsistencies between caller and callee functions in a program.

Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities [40]

The paper presents the development of coverage-based fuzzing tools to better detect use-after-free vulnerabilities. The working principles of the extended approach are as follows. First, the Control Flow Graph (CFG) gets generated, and all node sequences that may lead to use-after-free are extracted. Next, using extracted control sequences, fuzzing is executed where inputs are prioritized based on how many edges of a path they have visited. The more edges that belong to a path input visits, the more likely it is to be mutated. Eventually, the program tries

to find such input that would visit all the nodes in a path that would confirm the existence of the use-after-free bug.

Detecting Missed Security Operations Through Differential Checking of Object-based Similar Paths [55]

The paper presents an approach for detecting bugs in programs that relies on inconsistencies between two similar paths. The whole idea is to find similar paths with respect to some critical variable and analyze whether similar paths share the same security operations. If two paths are deemed similar, but one path, for instance, allocates/releases a pointer or locks/unlocks some resources while another path does not do that, then it is deemed a bug. Path similarity is calculated based on the Control Flow Graph (CFG), which is traversed to find different paths in a function that are similar with respect to some object.

MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures [38]

The paper develops further existing clone-based and function-matching-based approaches. Research introduces a novel approach for determining two function similarities that relies on signatures. Signatures can be categorized into three categories: function, vulnerability, and patch. The function signature is generated by taking into consideration not only function syntax but also semantics based on control and data dependencies in a function. Vulnerability and path signatures are further found based on the function signature and the patch function slices. When a vulnerability and patch signature are generated, all program functions are scanned to determine whether the function contains the same vulnerability. Function is deemed vulnerable if it matches the vulnerability signature but does not match the patch signature.

Learning to Detect Memory-related Vulnerabilities [56]

The paper presents a novel approach for the detection of memory-related vulnerabilities based on Graph Neural Networks (GNN). First, the Abstract Syntax Tree (AST) and System Dependence Graph (SDG) are generated from the source program. Based on the SDG, potential places with memory vulnerabilities are identified, and program slicing is performed to reduce possible noise from irrelevant code statements. Generated slices are then vectorized using a syntax-aware statement encoder that utilizes an AST for encoding the slice nodes and statements. Lastly, generated vectors that represent SDG slice nodes with slice vectors are fed to the Flow-sensitive Graph Neural Network (FS-GNN) for analysis.

LineVD: Statement-level Vulnerability Detection using Graph Neural Networks [57]

The paper presents an approach to detecting vulnerabilities in C/C++ programs using artificial intelligence. The main difference of the proposed approach is the utilization of both function and statement-level information to predict vulnerabilities in the source code. The function-level information comes directly from tokenizing a function. Statement-level information comes from tokenizing each function statement and then converting the tokenized statements into a Program Dependency Graph (PDG) that is then converted into vectors using Graph Attention Network (GAN).

HotCFuzz: Enhancing Vulnerability Detection through Fuzzing and Hotspot Code Coverage Analysis [39]

The study presents an approach for guided fuzz testing based on hotspot code coverage. Hotspots are detected in a source code by static code analysis, specifically by traversing the Abstract Syntax Tree (AST) and the Program Dependence Graph (PDG). AST is traversed to locate all potentially vulnerable places using a comparison to a known vulnerable snippet. PDG traversal is then done to find all anomalies between functions based on control and data dependencies. Gathered information about hotspots is then used to guide the fuzzing process in prioritizing inputs that have larger hotspot code coverage.

III. Commands Used for Static Code Analyzers

Clang

- clang --analyze -Xclang -analyzer-checker=core,alpha.core,security,alpha.security,optin,unix,alpha.unix,nullability -Xclang -analyzer-disable-checker=security.insecureAPI -ferror-limit=0 <filename>
- clang++ --analyze -Xclang -analyzer-checker=core,alpha.core,security,alpha.security,optin,unix,alpha.unix,nullability,cplusplus,alpha.cplusplus -Xclang -analyzer-disable-checker=security.insecureAPI -Xclang -analyzer-config -Xclang aggressive-binary-operation-simplification=true -ferror-limit=0 <filename>

Cppcheck

- cppcheck --enable=warning,portability --force --inconclusive <filename>

Ikos

- ikos -w --globals-init=all -a "boa, dbz, nullity, prover, uva, sio, uio, poa, shc, pcmp, sound, fca, dfa" -f text --rm-db --entry-points=<function> <filename>

Infer

- infer run --default-checkers --headers --biabduction --biabduction-unsafe-malloc --bufferoverrun --pulse-unsafe-malloc --keep-going -- gcc -c <filename>

GCC

- gcc/g++ -fanalyzer -Wall -Wextra -Wformat=2 -c <filename>

Symbiotic

- symbiotic --prp=memsafety --malloc-never-fails <filename>

IV. Reviewed Paper Tools Configuration

CorCA [58]

Steps taken:

- Repository cloned locally (`git clone https://github.com/iberiam/CorCA.git && cd ./CorCA`)
- All files unzipped:
    - `apt install unzip`
    - `unzip ./tool/CorCA-0.1-withExamples.zip`
    - `unzip ./SourceCode.zip`
- afl-gcc installed (`apt install afl++`)
- Required Python modules installed
    - `pip3 install flawfinder==2.0.10`
    - pycparser installed from source because tool needs access to files inside fake_libc_include directory that does not come when installing pycparser using pip.
        - `wget https://github.com/eliben/pycparser/archive/refs/tags/release_v2.20.zip`
        - `unzip release_v2.20.zip`
        - `cd pycparser-release_v2.20`
        - `python3 setup.py install`
- Downloaded, unzipped, and built an AFL fuzzing application that is needed as a dependency
    - `wget https://github.com/google/AFL/archive/refs/tags/v2.57b.zip`
    - `unzip v2.57b.zip`
    - `cd ./AFL-2.57b && make && make install`
- Word 'core' added to file /proc/sys/kernel/core_pattern as per installation manual (`echo core > /proc/sys/kernel/core_pattern`)
- config.ini file configured with the required values

- `librariesPath = /usr/CorCA/pycparser-release_v2.20/utils`
- `fuzzerPath = /usr/local/bin/afl-fuzz`
- `compilerPath = /usr/bin/afl-gcc`
- `execTime = 30`

After configuration, the tool can be used to analyze files using the following command: `python3 CorCA.py -i <file.i>`. Nevertheless, the tool does not work completely. When analyzing files, it can run Vulnerability Identifier, which, if it does not detect any vulnerability, outputs `No potential vulnerabilities found. Terminating execution`. However, when the tool detects a vulnerability, it tries to extract it into a separate file for fuzzing, and at this step, a certain error always occurs `linker input file unused because linking not done`.

IncreLux [59]

Steps taken:

- Repository cloned locally (`git clone https://github.com/seclab-ucr/IncreLux.git && cd ./IncreLux`)
- setup script ran (`sh setup.sh`)

The setup.sh script fails with various generic errors, and no executable is produced under the build/bin folder.

Lightweight Shape Analysis Based on Physical Types [60]

This tool is unique as it provides an image with a precompiled binary of the program. Image itself can be imported into VirtualBox, and existing tests can be executed (`cd /home/vmcai/Desktop/shape_benchmarks/c_liSemanticDirected2017 && rm -f *.csv && make`). From the Makefile command that is executed on files can also be extracted, and it is:

- `frama-c -kernel-debug 2 -codex-msg-key evaluating -load-module /home/vmcai/Desktop/codex/frama-c/top/CodexPlugin -codex -codex-msg-key evaluating -codex-verbose-terms 0 -codex-domains 6 -ulevel -1 -codex-exp-dump test.dump -no-allow-duplication -codex-focusing <file.c> -codex-function <entry point>`

This command can compile existing tests that come with the provided image, but when trying to execute the tool on any other C program, various errors are often produced, like:

- `Fatal error: exception CodexPlugin.Compilation_to_term.Make(P).Analysis_type_not_found("__anonstruct_GF_M2TS_Program_3")`.

Errors are produced even though all analyzed programs can be compiled and executed without third-party dependencies.

DeepWukong [61]

Steps taken:

- Repository cloned locally (`git clone https://github.com/jumormt/DeepWukong.git && cd DeepWukong`)
- Python required packages installed from requirements.txt file (`sh ./env.sh`)
    - Because DeepWukong requires Python version 3.6. Python version management software called pyenv [85] was used during installation on both Ubuntu 20.04 and Ubuntu 22.04. After the required version of Python was installed, the required dependencies could not be installed. During installation, there were conflicts among dependencies that the author could not resolve.

SyzRisk [62]

To work properly, the tool requires pattern risks and pattern weight estimation per project. Moreover, the tool utilizes guided fuzzing, which is not very suitable for the current research scope.

NDI [63]

Steps taken:

- Repository cloned locally (`git clone https://github.com/umnsec/ndi.git && cd ndi`)
- LLVM 15.0.0 cloned and built:
    - `cd llvm`
    - `wget https://github.com/llvm/llvm-project/archive/refs/tags/llvmorg-15.0.0.zip`
    - `apt install unzip`
    - `unzip llvmorg-15.0.0.zip`
    - `mv llvm-project-llvmorg-15.0.0 llvm-project`
    - `sh build.sh`

- The tool is built
    - `cd ../analyzer`
    - `apt install libz3-dev`
    - `make`

Now the executable is available at `./build/lib/kanalyzer` but it does not output errors on any analyzed test cases. The command executed considering the project source code directory was cloned inside the usr directory is as follows:

```
- /usr/ndi/llvm/llvm-project/build/bin/clang -emit-llvm -c <file.c> -o ndi.bc
  && /usr/ndi/analyzer/build/lib/kanalyzer ./ndi.bc
```

IPPO [64]

Steps taken:

- Repository cloned locally (`git clone https://github.com/dinghaoliu/IPPO.git && cd IPPO`)
- LLVM 9.0.0 cloned and built:
    - During this step, the $OSTYPE environment variable had to be manually set for the build-llvm.sh script to work properly
        - `export OSTYPE=linux-gnu`
    - `cd llvm && sh build-llvm.sh`
- tool built
    - Before installation, also install some required undocumented dependencies
        - `apt install cmake libncurses5 libtinfo-dev`
    - `cd ../ && make`

Now the analyzer executable is available in /usr/IPPO/build/lib, considering the source code was cloned inside the usr folder. When trying to analyze an object file with the command:

```
- /usr/IPPO/llvm/build/bin/clang -fno-inline -g -O2 <file.c> -o ippo.bc &&
  /usr/IPPO/build/lib/analyzer ./ippo.bc
```

The following error is produced:

```
- /usr/IPPO/build/lib/analyzer: error while loading shared libraries:
  libomp.so: cannot open shared object file: No such file or directory
```

Installation of the libomp library (`apt install libomp5 libomp-dev`) as well as other attempts could not resolve this runtime problem.

MVD [65]

The tool utilizes neural networks and requires training. Model training documentation is to be released later, so this tool was not tested or considered.

V. License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Roman Amjaga ,
(author´s name)

1. grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

   Static Analysis to Detect Memory Corruption Vulnerabilities ,
   (title of thesis)

   supervised by Vesal Vojdani ;
   (supervisor´s name)

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Roman Amjaga
15/05/2025