UNIVERSITY OF CAGLIARI

FACULTY OF ENGINEERING AND ARCHITECTURE

Master's Degree in
Computer Engineering, Cybersecurity and Artificial Intelligence

Effective Graphical Visualization of Vulnerabilities in C and C++ Programs

Supervisor:
Prof. Davide Maiorca
Co-Advisor:
Alessandro Sanna

Thesis by:
Matteo Asuni matr.
70/90/00330

Academic Year 2023-2024

Contents

Chapter 1

Introduction

Software is more present than it used to be; sometimes, its presence might not be evident. From household appliances to our vehicles, nowadays, almost every piece of tech relies on software to perform its designated tasks, but also critical infrastructures such as the power grid: in literature, there are many examples of how the power grid is transforming into the Smart Grid. In [1], the authors explain the different technologies employed to make the power grid "smarter."

DataReportal [2] shows that 5.16 billion people (64.4% of the population) are online. Figure 1.1 shows how this number changed from 1990 to 2023. Being connected enriches our lives in ways we could not have thought of during the days of ARPANET [1]: we can easily communicate in real-time with people far from us, exchange knowledge through different means, and more.

Unfortunately, this increasing number of connected users entails that the attack surface available to cyber criminals will become more prominent. Malicious actors can cause damage by exploiting hardware or software vulnerabilities. If properly used, vulnerabilities can harm the final users of a product, both consumers and businesses. A cyber attack might impact the vendors as well. As stated by Anwar et al. in [3], an attack can lead to the loss of trust by the customers, ruined brand reputation, and loss of customers. It is also essential to consider the cost of patching a system after discovering a vulnerability, including time and money.

The focus of this thesis is the visualization of Software Vulnerabilities. To visualize the vulnerabilities, they must be found, and regarding this, there are multiple approaches that software developers could use to find these vulnerabilities. The most common ones are static analysis and dynamic analysis, which rely on the source code of a program (eventually transformed into intermediate representations). In contrast, dynamic analysis requires the execution of the program.

[1] Advanced Research Projects Agency Network, the technical foundation behind the Internet. This project, first implemented in universities, allowed resource sharing between computers.

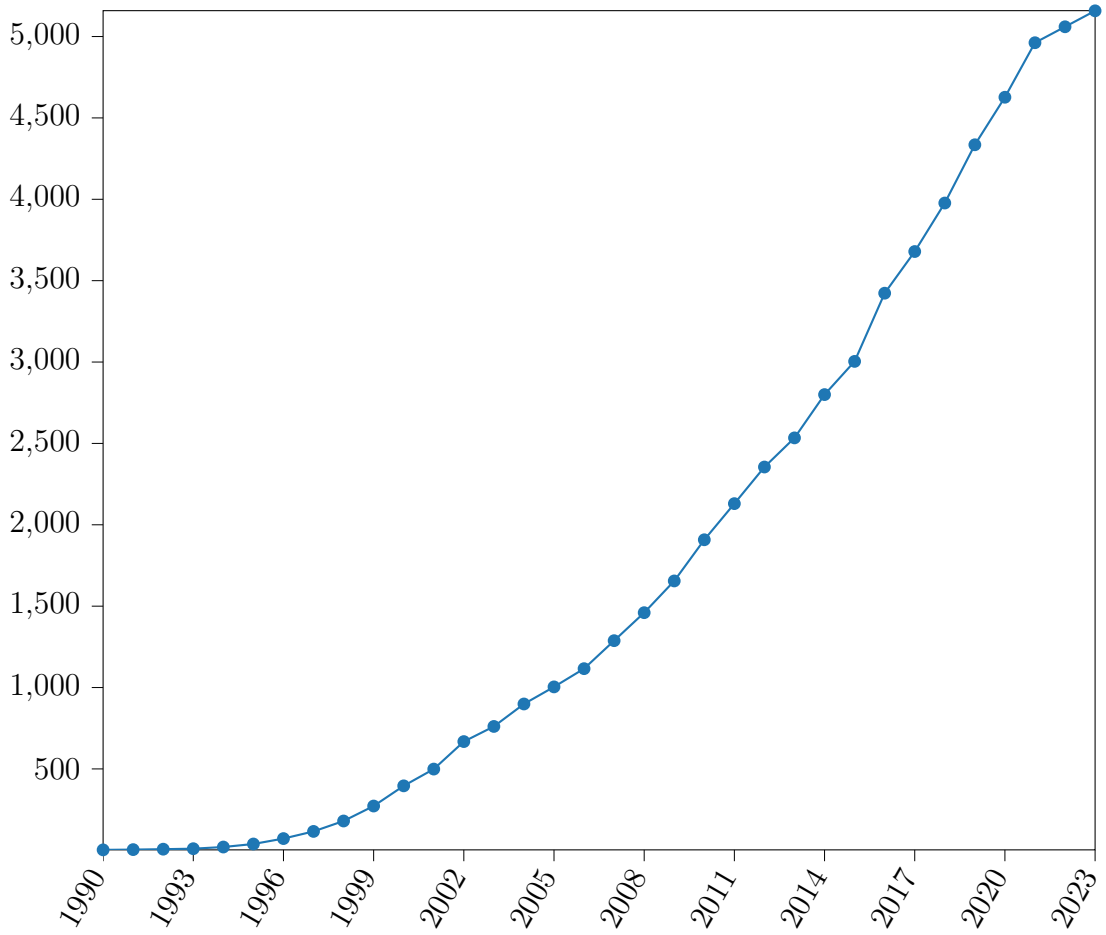

Figure 1.1: Number (Millions) of Users that are online [2].

These approaches can be more or less automated, with pros and cons and multiple implementations available. Also, nowadays, software is more complex than it used to be, and human error is always around the corner, so software is expected to contain some error that leads to a potential vulnerability. Note that external libraries that perform specific tasks can also introduce potential vulnerabilities. A great example is the Log4Shell vulnerability[2], a zero-day vulnerability [3] disclosed in 2021 that affected the famous logging framework Log4j [4]. When exploited, this vulnerability allowed for remote code execution, and due to this, it received a CVSS severity score of 10: this corresponds to critical. Multiple actors exploited this

[2] `https://nvd.nist.gov/vuln/detail/CVE-2021-44228`
[3] A vulnerability that is unknown to anyone capable of fixing the impacted system.
[4] `https://logging.apache.org/log4j/2.x/`

vulnerability, as reported by Bitdefender [4] and others, to expand botnets [5](such as the Muhstik Botnet), run crypto miners [6](like the XMRIG miner), or deploy ransomware[7](such as Khonsari).

Although tools exist to detect vulnerabilities, solving this problem and preventing most of them is more complex than it may seem.

Multiple examples exist in the literature of why developers do not use these tools. Reynolds et al. [5], for instance, explain that the output of the automatic analysis tool could often be more straightforward for developers. They also talk about one of the main issues that comes from using static analysis tools: the presence of false positives. A recent review [6] discusses the current state of static analysis and potential solutions to mitigate the false positives problem. However, how can the analysis output be made less harsh for the developers? One viable solution consists of implementing some visualization for this data. produced a review of visualization tools for cybersecurity-related problems. At the beginning of the review, they also remember why visualization can be helpful. A well-implemented visualization can help developers quickly answer questions about a specific matter, gain new insights, and, most importantly, process this information better. Assal et al. provided a great example of vulnerability visualization in [7]. Here, they talk about Cesar, a tool for visualizing the output of FindBugs [8], a static analysis tool for Java programs. In the paper, they also show the results of the user interface evaluation.

Figure 1.2: From [7]: "Cesar's visualization, details, and source code panes"

[5]A trojan that takes control of several computers to allow malicious actors to perform all kinds of actions, such as Distributed Denial of Service attacks.

[6]A software that can be used to validate blocks of a blockchain, by validating a block the user receives a monetary reward. Attackers use these to generate revenue by utilizing the victim's computing power.

[7]Ransomware is a kind of computer virus that encrypts the content of a computer and asks it to pay a ransom to get the decryption key to retrieve the files, with no guarantee of getting the files back.

[8]`https://findbugs.sourceforge.net/`

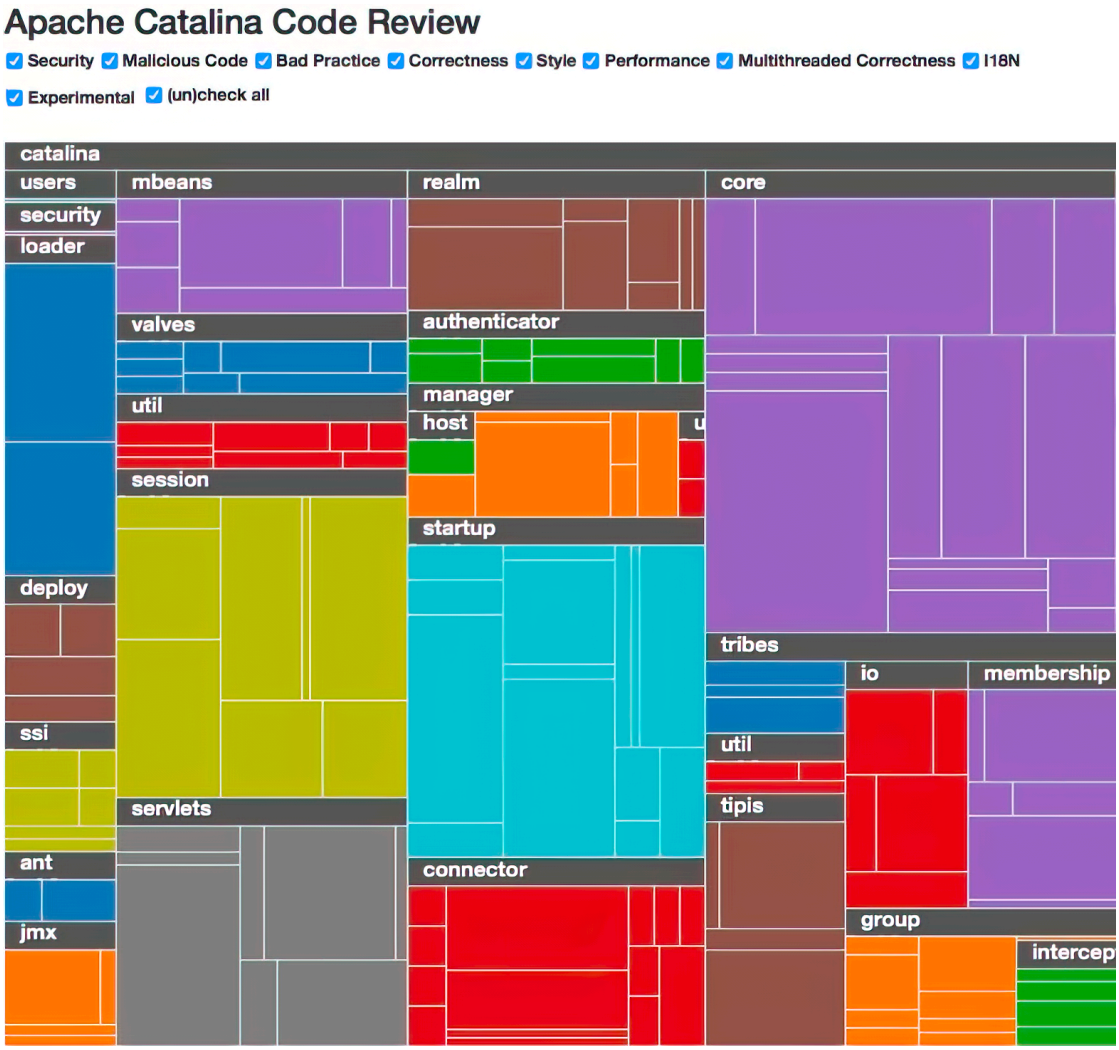

Figure 1.3: From [7]: "Cesar's' treemap showing the distribution of selected vulnerability categories in package Catalina"

In 2002, the National Institute of Standards and Technology (NIST) estimated an annual economic loss of approximately 60 billion dollars in the United States due to expenses associated with patch development, redistribution, system redeployment, and direct productivity loss resulting from vulnerabilities [3], [8]. Multiple factors influence the economic impact of a vulnerability; these may include the kind of vulnerability, the industry (for example, pharmaceutical or financial), the potential users, and how they may perceive a potential vulnerability [3]. Canalys, a market analysis firm, reported that global spending on cybersecurity-related products or services reached $71.1 billion in 2022, with expected growth to $458.9 billion by 2025 [9].

Over the years, multiple cyber attacks have caused significant economic damage; one of the most recent ones is the WannaCry ransomware (2017). Several news outlets report that at least 150 countries were affected by it [10], [11]. A press release about a collaboration between the insurance firm Allianz and the cyber risk analytics and modeling firm Cyence [12] reports an estimated loss of 8 billion dollars, including the cost of the ransom, potential profits that were lost, and expenses to restore the systems. Figure 1.4 represents data coming from an elaboration of the statistics platform Statista about worldwide spending for Information Security products and Services between 2017 and 2023 divided by category, along with an estimation for 2024[9]. This elaboration is based on Gartner data[10], a technological research and consulting firm.

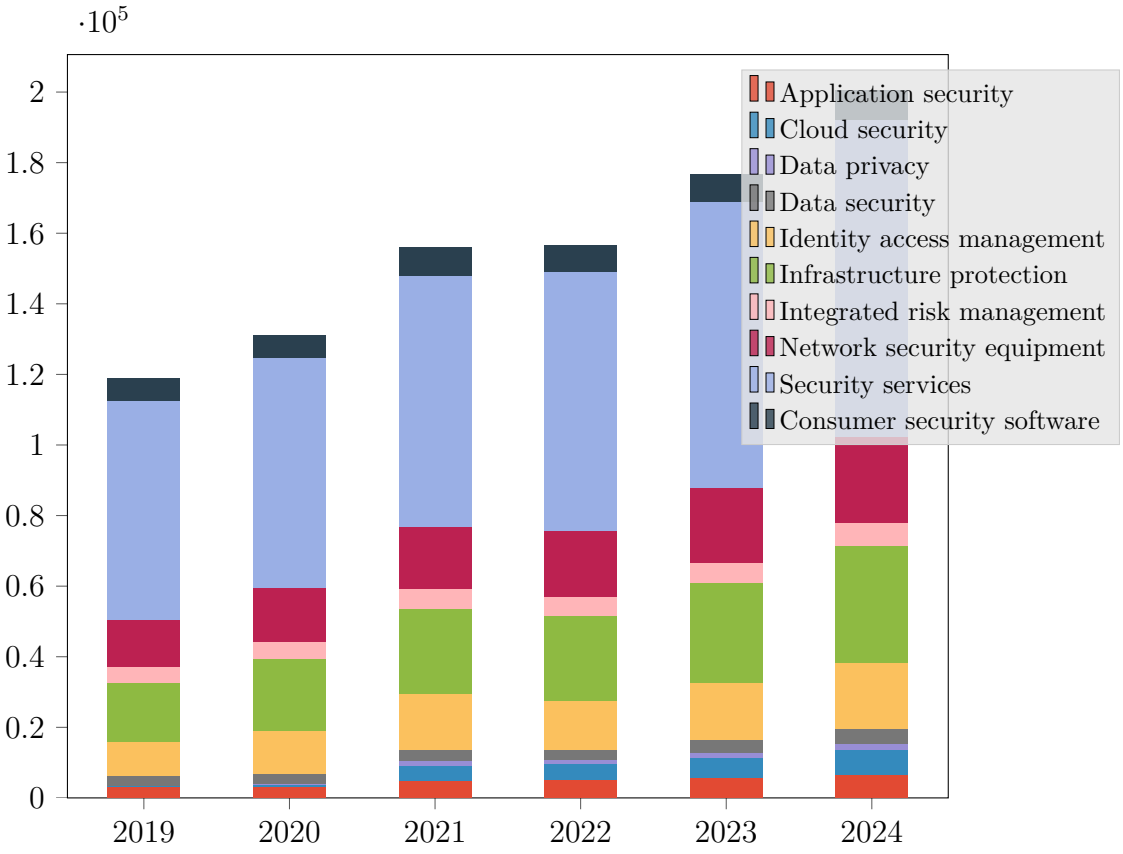

Figure 1.4: Information security spending worldwide from 2019 to 2024, by segment (in million U.S. dollars). Data source[10], elaboration[9]

[9] https://www.statista.com/statistics/790834/spending-global-security-technology-and-services-market-by-segment/

[10] https://www.gartner.com/en/newsroom/press-releases/2023-09-28-gartner-forecasts-global-security-and-risk-management-spending-to-grow-14-percent-in-2024

According to this, it is possible to see that most spending is on the "Security services" category.

The annual report of the United States' Internet Crime Complaint Center (IC3) [11] reports a total damage of 10300 Billion dollars due to Cyber attacks in 2022. Figure 1.5 shows how the monetary damage increased throughout the years. This has been somewhat steady until 2017, and from that year, there has been exponential growth.

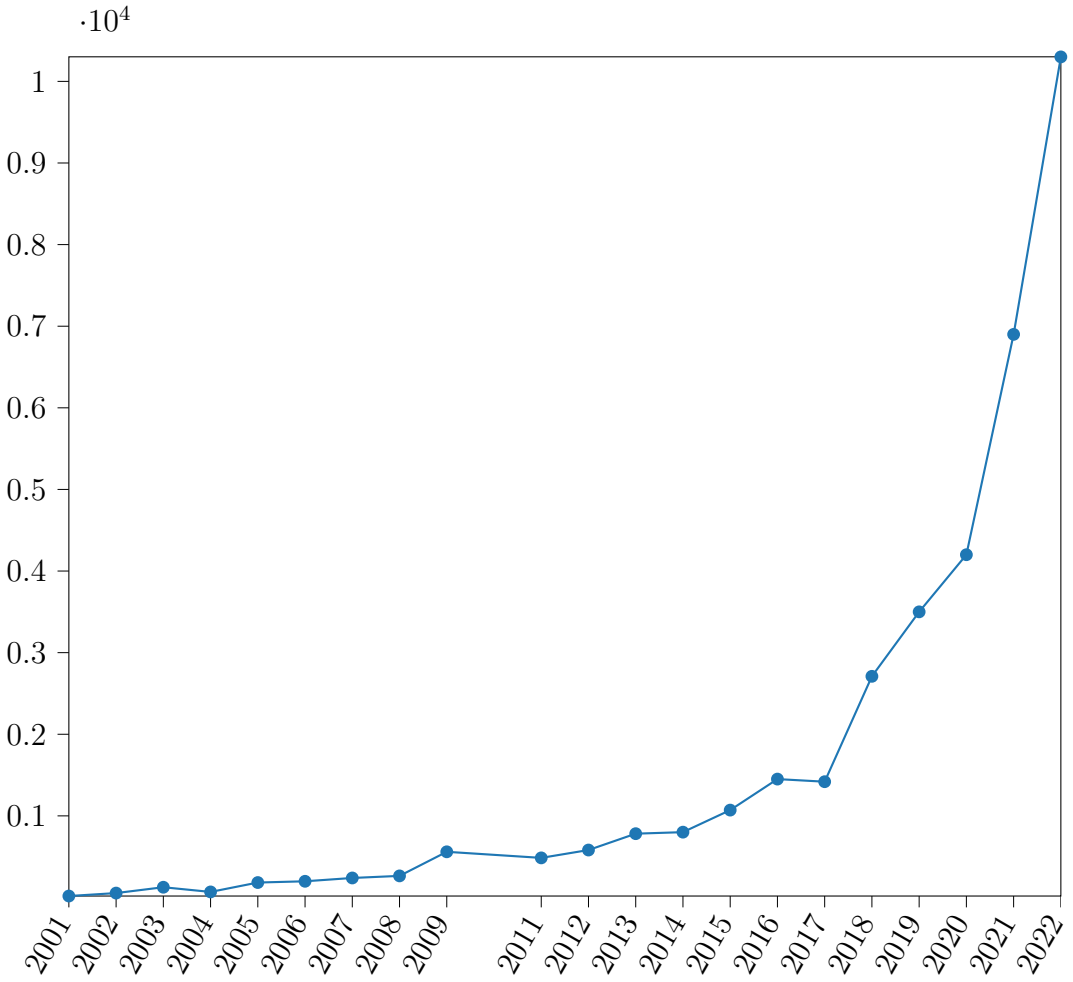

Figure 1.5: Monetary damage caused by reported cybercrime in the US from 2001 to 2022. 2010 is not present due to missing data point.

This number is expected to increase due to the increasing number of vulnerabilities that are out in the public: an excellent resource for tracking publicly available

[11] https://www.ic3.gov/Media/PDF/AnnualReport/2022_IC3Report.pdf

vulnerabilities is the National Vulnerabilities Database (NVD), maintained by the United States government. According to their metrics [12], in 2023, 28961 vulnerabilities (identified by a CVE ID) were reported, with 2023 also being the year with the all-time high number of reports. It is crucial to consider that many systems might remain unpatched for years, so old vulnerabilities can be exploited. Figure 1.6 shows how this number changed from 1999 to 2023.

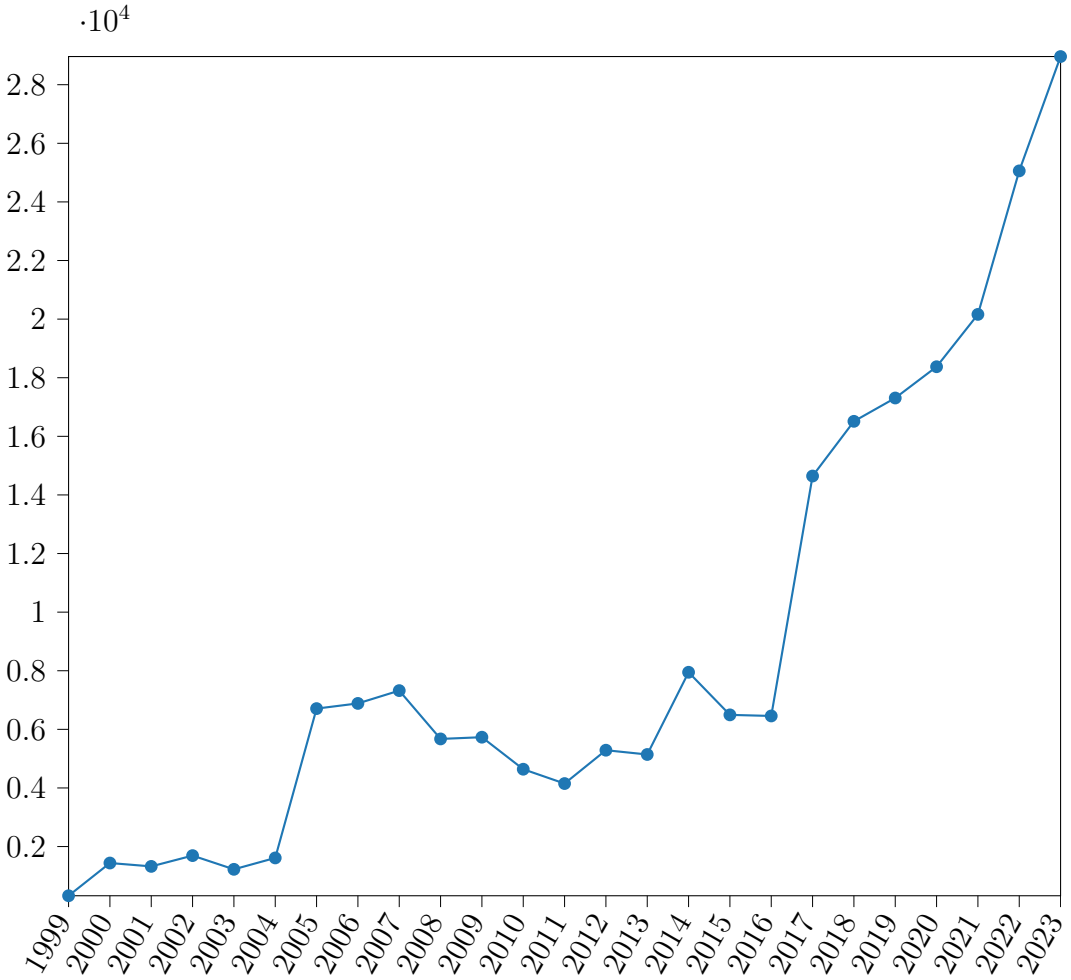

Figure 1.6: Number of published CVEs per year from 1999 to 2023.

[12] https://www.cve.org/About/Metrics

1.1 Goals and Contributions

This first chapter serves as an introduction to the primary goal of this work, expanding an existing tool for software architecture visualization by embedding vulnerability information from different analysis tools.

The following chapter will provide the background needed to understand the work better:

- An introduction to the structure of C and C++ programs.
- How vulnerabilities can be detected inside a program.
- What is a vulnerability, and how can they be cataloged?
- The use of graphs for code representation.

The next chapter reviews the state-of-the-art tools that can perform vulnerability analysis in C and C++ programs, focusing strongly on graph-based approaches. Each tool will be given a small description focusing on the peculiarities and how they perform the analysis.

The methodology chapter will explain the contributions, focusing on Rascal, the metaprogramming language used to implement the knowledge model, and the visualization tool ClassViz[13]. Finally, a discussion about the results The result chapter will focus on explaining the visualization tool's user interface by showing some examples and comparing it with a similar tool. The last chapter will focus instead on discussing the results, limitations, and potential future works.

The following key points can summarize the contribution:

- Implementation of vulnerability visualization in ClassViz.
- Implementation of the knowledge model, defined by Rukmono and Chaudron [13] for C and C++ programs.

[13] `https://github.com/rsatrioadi/classviz`

Chapter 2

Background

Nowadays, applications are more complex than ever; they implement many functionalities, interact with more data and external services, and, most importantly, have a tighter release schedule that may force developers to make some compromises. All of this inevitably leads to applications that are more prone to vulnerabilities.

Since the implementation of this thesis involves the visualization of vulnerabilities in C and C++ projects, this chapter will describe the structure of C and C++ applications, along with the various vulnerability analysis techniques and the main kinds of graphs used to represent code.

2.1 Architecture of C and C++ programs

The C programming language is one of the most popular high-level programming languages. It is defined as general-purpose since it is not specialized for a specific use case (one example of a highly specialized programming language is R). C and C++ are compiled programming languages, which means that programs written with one of the two languages must be compiled using a compiler to obtain an executable binary compatible with our machine. Note that C and C++ do not impose a folder structure for the projects. A single source code file could contain a complete program. However, while this is possible, it is not recommended for modularity and readability reasons.

The whole process is divided into multiple steps:

- Preprocessing: In this phase, a preprocessor analyzes the source code to find the preprocessor directives. For example, the `#include` directive indicates that the content of an external file must be copied inside the source code of our program. This is usually done when macros or external libraries are used.

- Compilation: In this phase, the source code written in C is translated into machine language (object code). During this phase, the compiler notifies about compilation errors, such as language rule violations or "syntax errors."
- Linking: Here, the object code is linked to the external libraries that might be used throughout the program to obtain an executable containing all the pieces necessary to run the program.

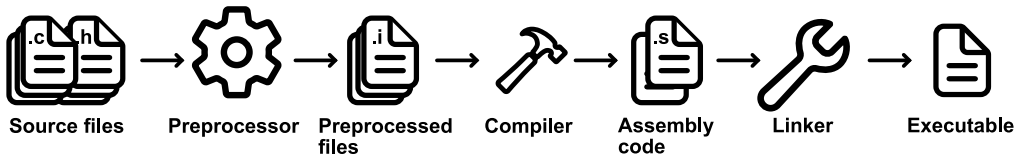

Figure 2.1: Compilation process for C/C++ programs

The compilation process must be repeated for every architecture and operating system one wants to target. Being able to run the same program under different kinds of machines is one of the main advantages of C/C++. Another advantage of these two programming languages is performance. It is not uncommon for Operating Systems to have portions of it written using C/C++. Likewise, embedded systems rely heavily on these two languages due to their low resource consumption (e.g., memory-wise). As compiled languages, interpretation is unnecessary at run-time, and several optimizations are applied at compile-time.

2.2 Detection techniques

In recent years, there has been a spike in interest in software vulnerabilities and tools to find them. Hanif et al. presented in 2021 a taxonomy [14] that describes the main techniques that can be used for detecting vulnerabilities, along with some insight into the current research trends. This section will refer to this categorization to describe the current landscape. The authors define three principal methodologies in this taxonomy: manual analysis, conventional techniques, and machine learning approaches.

Manual analysis relies on a human expert to perform the testing and debugging of software during its development phase to detect vulnerabilities. As one can expect, this method is time-consuming, especially for large-scale projects that

nowadays are more common than before. Besides requiring much time, it is also prone to human error, and the quality of it depends on the specific subjects chosen to perform the task, as some have different levels of knowledge. Thus, it can not be a reliable way to perform vulnerability analysis.

Conventional techniques refer to all the methods used (and still in use) before the advent of newer ones based on machine learning. This category includes static and dynamic analysis and fuzz testing. Often, these have a low detection performance and may produce many false positives. False positives consist of all cases where a non-vulnerable sample is classified as vulnerable. A higher false-positive rate can lead to an increase in time cost. Static analysis relies on analyzing a program's source code without executing it. Dynamic analysis instead requires the program to be compiled and executed, usually with some test inputs designed to reach all the possible execution states of a program. Without a proper input data set, some parts of the program might not be reachable and remain untested.

However, nowadays, most researchers' efforts are directed toward machine learning-based approaches. Given the rapidly evolving nature of software vulnerabilities, research efforts for these new approaches are more important than ever. Machine Learning approaches are very efficient and cost-effective but still need improvement. The main techniques are based on supervised learning, semi-supervised learning, ensemble learning, and deep learning. Supervised learning works best when paired with a well-labeled dataset, which can be used to train models after preprocessing and cleaning. Semi-supervised approaches instead expand the datasets with unlabeled data that gets labeled by performing predictions. This approach is helpful for those scenarios when obtaining a giant mole of data is challenging. Ensemble learning instead combines multiple classifiers into a single one to decrease the error rate. Instead, Deep Learning relies on a network architecture of various layers of neurons that perform calculations based on the input.

2.3 Vulnerabilities and Weaknesses

To better understand this thesis, it is necessary to define a vulnerability. A good definition of vulnerability comes from NIST (National Institute of Standards and Technologies) Special Publication 800-53 revision 5 [15]. In this and many other documents, they define a vulnerability as: "Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.".

One of the primary sources of information about vulnerabilities is the National

Vulnerabilities Database (also known as NVD)[1], created by the U.S. government in 1999. NVD performs analysis on the Common Vulnerability and Exposures (CVE) records. A CVE record refers to a specific vulnerability found in a software or library. Each CVE is identified by a unique ID that MITRE or other authorized authorities called CVE Numbering Authorities (CNAs) can issue. Each record contains a description that usually includes the type of vulnerability, the software vendor, and the affected code base, if available. Some references might be added to give more context. Once a CVE is published, the NVD reviews the reference material and assigns a Common Weakness Enumeration (CWE) used to specify the class of vulnerability. After this, exploitability and impact metrics are calculated to classify the vulnerability better, including a base, a temporal, and an environmental score. NVD gives the base score following the Common Vulnerability Scoring System (CVSS)[2]. It is used to indicate the severity of the vulnerability according to its intrinsic characteristics, such as the attack vector and complexity, the required privileges, and which area is impacted; the temporal score, instead, can be used to upgrade the severity of the vulnerability if, for example, a new exploit came out. The environmental score instead is used to indicate the impact of the vulnerability in a specific environment.

2.4 Most popular weaknesses

CWE publishes a series of insights regarding weaknesses, such as a list of the top 25 weaknesses that appeared in the NVD. CWE creates this list by giving a score based on the frequency of appearance and the severity of it. A weakness that is both popular and can cause significant harm will receive the highest score. In Figure 2.2, a graph represents how the ranking changed from 2022 to 2023. Blue lines represent an equal rank between the years, a green one a higher rank, and red ones a lower rank.

[1] `https://nvd.nist.gov/`
[2] `https://www.first.org/cvss/specification-document`

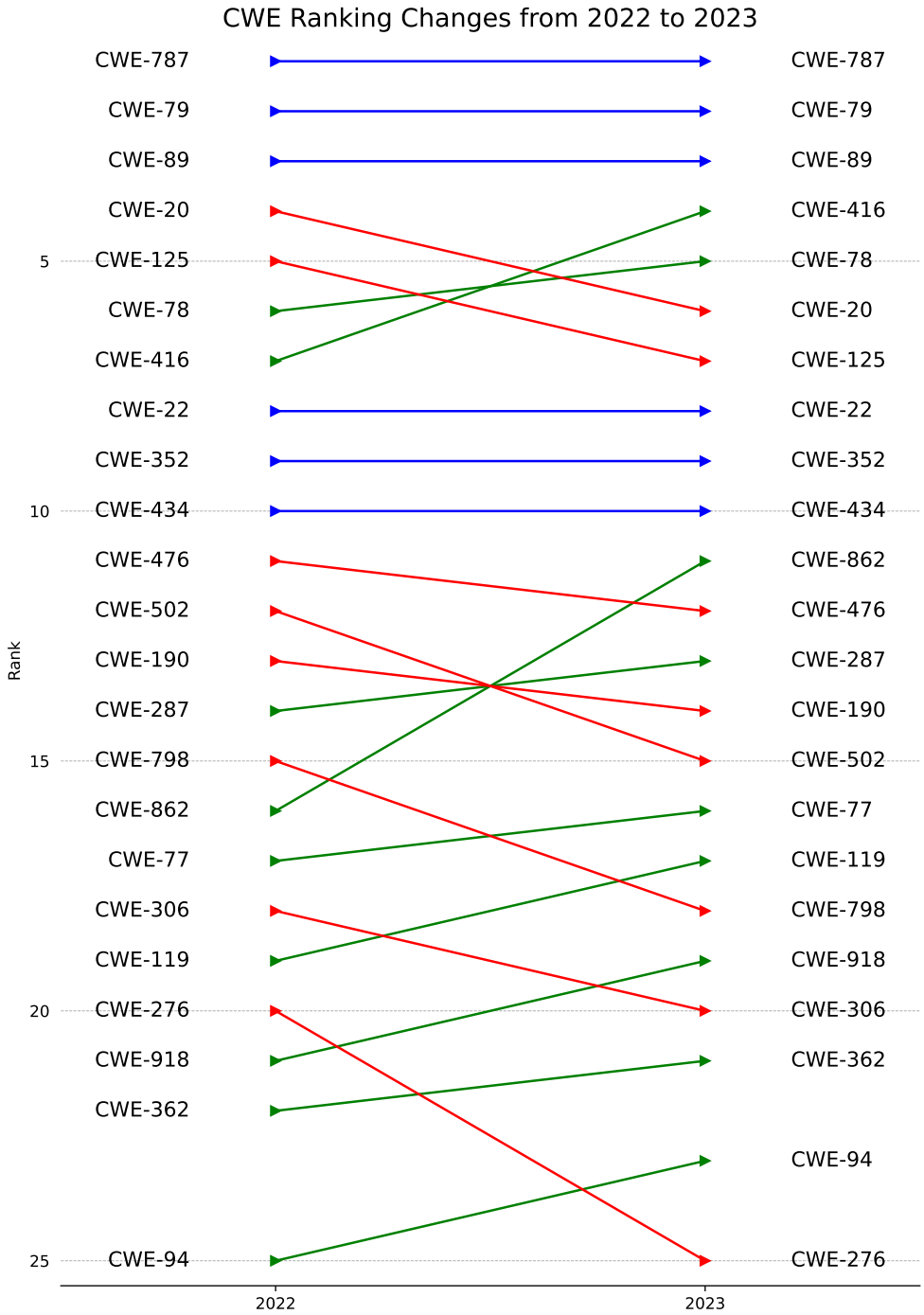

Figure 2.2: CWE Ranking Changes from 2022 to 2023

Here is an extract of that list with some of the most popular weaknesses based on data gathered between 2021 and 2023.

- CWE-787: Out-of-bounds Write

  This weakness occurs whenever there is an attempt to write any form of data past the end or before the beginning of the designated memory location. The consequences of this weakness are data corruption, crashes, or code execution.

- CWE-416: Use After Free

  This weakness occurs whenever there is an attempt to access a memory location after this has been marked as "free´´ by using the specific functions provided by the programming language. When a memory location is marked as free by a program, it gets removed by the set of legal memory locations that can used to execute the program. The consequences of this weakness are data corruption, crashes, and execution of arbitrary code.

- CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

  This weakness occurs whenever a program uses functions that can execute commands in the operating system before checking the given command. This scenario becomes even worse when this command is an arbitrary user input. Not checking or performing any neutralization on the command means the program provides an entry point to steal sensible information or modify a system's configuration by executing a malicious command.

- CWE-125: Out-of-bounds Read

  This weakness occurs whenever there is an attempt to read any form of data past the end or before the beginning of the designated memory location. The consequences are the possibility of accessing memory locations without proper authorizations and program crashes.

- CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

  This weakness occurs whenever a program uses external input to reconstruct a path that will be used by a program to access a resource without neutralizing the input. This input might contain special characters that can be used to reach other paths without having the proper authorizations. An example could be using `../../test` as an input. Without normalization, the program will try to access the resource test located two levels before the

current one. The consequence of this is access to files and resources without authorization.

- CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

  This weakness occurs whenever a program has one or more code sequences that can run concurrently and require temporary, exclusive access to a shared resource that another part of the code might use. Imagine that one of these code sequences handles user authentication. What could happen if another portion of the code accessed the resources used for authenticating a user? The consequences are a lack of availability, confidentiality, and integrity according to the task done by the involved code portion.

2.5 Code as a Graph

Modern codebases can easily reach millions of lines of code and are scattered in multiple files. For example, according to the stats provided by GitHub, the Linux Kernel has ∼54 thousand C source files [16]. Analyzing a similar project would require reviewing each file; this is extremely time-consuming. However, there are alternative methods in the literature that utilize graphs to perform codebase analysis: Neamtiu et al. [17] used Abstract Syntax Trees for comparing the source code of C projects across multiple versions, Wang et al. [18] instead used them for detecting code clones. The next Chapter will provide different approaches that employ graphs to detect vulnerabilities. Thus, exploring the main types of graph representation regarding source code is crucial. For all the examples, I will use this simple C program:

```c
int main(int argc, char *argv[]) {
  int c = 42;
  if (argc > 1)
  {
    exit(c);
  }
  exit(0);
}
```

Listing 1: Sample program written in C

- Abstract Syntax Tree(AST), as the name suggests, is a tree that represents the syntax of a program in an abstract way. It can also be defined as a simplified version of the parse tree [19]. From this structure, it is possible to reconstruct the syntax of a program by seeing all the parameters involved with a function, conditional statements, return values, arguments, and variables. Take, for example, the simple program in Listing 1; its AST will be:

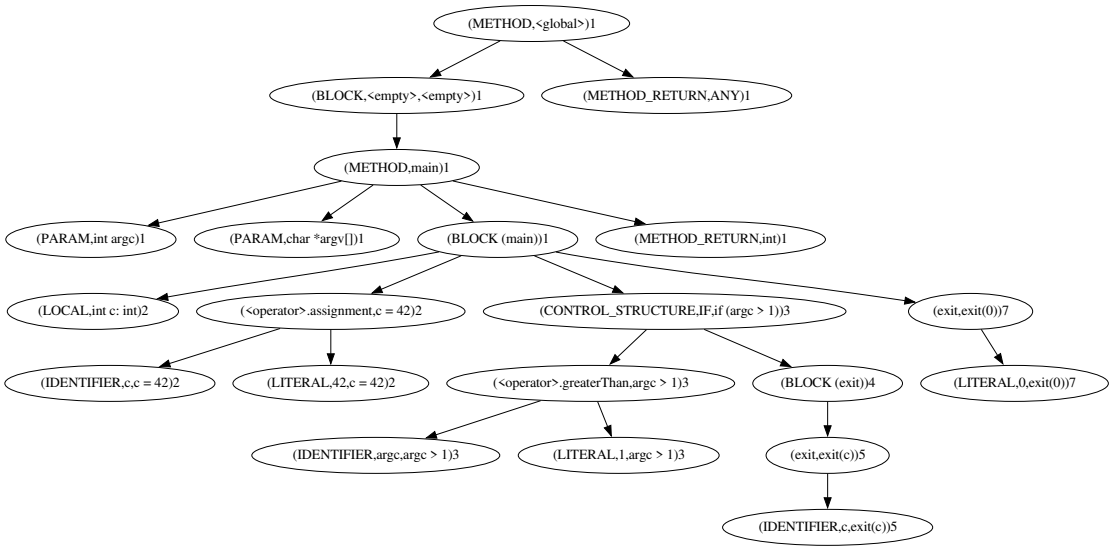

Figure 2.3: Example of an Abstract Syntax Tree for Listing 1

- Control-Flow Graph is a directed graph that shows all the reachable paths inside a program. Each graph node represents a basic block, a linear sequence of instructions with an entry point and an exit [20]. The edges instead represent the control flow paths between the various blocks.

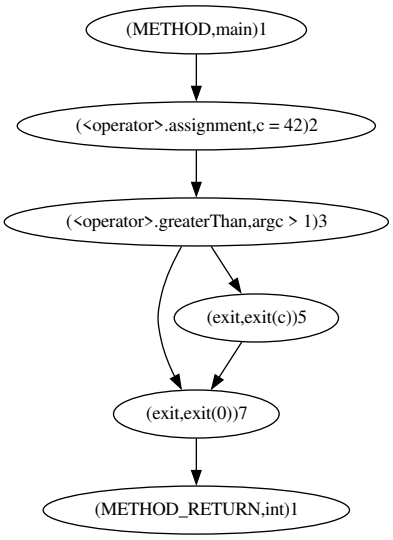

Figure 2.4: Example of an Control Flow Graph for Listing 1

- Data Dependence Graph is a directed graph that can be used to see where a value is defined and where it is used. The nodes of this graph represent operations; most operations contain both the definition and the use of a value [19]. The edges instead represent the use of that specific operation.

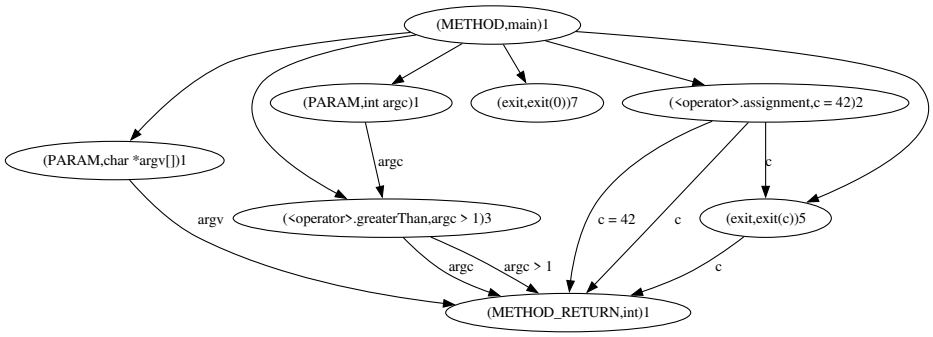

Figure 2.5: Example of an Data Dependence Graph for Listing 1

Chapter 3

State of The Art

Vulnerability analysis is a topic that has gained popularity in recent years, and multiple surveys that try to describe the current state of the art have been published and define a taxonomy to classify all the different methods. Some of these reviews focus on new approaches, such as Deep Neural Networks [21], while others try to give a broader view of the state of the art [14], [22], [23]. This chapter will focus on publications from conferences in the first three tiers of the "Computer Security Conference Ranking and Statistics" by Guofei Gu[1], plus some others from different sources, such as related papers or the surveys listed before. Given that in recent years, most of the approaches relied on graphs, the different approaches will be divided into those that use a graph to represent the code and use it to perform the actual analysis using various means and those that do not rely on a graph.

3.1 Non Graph-based approaches

In 2013, Meta (formerly Facebook) acquired the verification startup Monoidics. With this, they acquired Infer[2], a static program analyzer compatible with Java, C, C++, Objective-C, and C#. Infer is written in OCaml and is currently used by Meta to analyze the source code of their mobile apps, such as Facebook and Instagram. It can easily be integrated into a continuous integration scenario and supports the main build systems such as Cmake, Gradle, Make, and Maven.

In 2015, Shuai et al. published [24] a paper where they tried to solve the problems of traditional fuzz testing. Fuzz testing is a technique where an application is tested with malformed inputs. One of the main limitations comes from the input generation. If the program immediately fails, the testing ends up in a scenario where deeper code is never tested. In this paper, the authors propose

[1] `https://people.engr.tamu.edu/guofei/sec_conf_stat.htm`
[2] `https://fbinfer.com/`

an approach to produce better test cases by using static analysis, code coverage, and a genetic algorithm: a global optimization algorithm. Experiments performed on actual software show that this improved fuzzing approach can obtain higher vulnerability detection when compared to traditional ones.

In 2018, Li et al. published VulDeePecker [25], a system based on deep learning for detecting vulnerabilities and a dataset that can be used to evaluate similar approaches. In the paper, the authors recognize the potential benefits of a deep learning-based approach, and one is the absence of human intervention in picking the features needed for the system to work. Still, at the same time, they realize that programs can not be fed directly into a deep learning architecture, and they present some principles to make things work. VulDeePecker was tested on three software products (Xen[3], Seamoneky[4], and Libav[5]), and four vulnerabilities that were not reported to the NVD were successfully found. In 2021, this work was expanded by presenting SySeVR [26], where the authors overcome some of the limitations found during the development of VulDeePecker, such as

- "It considers only vulnerabilities related to library/API function calls."
- "It leverages only the semantic information induced by data dependency."
- "It considers only a particular RNN known as Bidirectional Long Short-Term Memory(BLSTM)."
- "It makes no effort to explain the cause of false positives and negatives."

This new framework found 15 vulnerabilities not reported to the NVD and seven unknown to the respective software owner.

In 2022, Liu et al. presented Acquirer [27], an approach to detect Algorithmic Complexity (AC) Denial-of-Service attacks in Java programs. This kind of attack involves crafting an input vector capable of triggering the worst-case logic of a running program to increase resource consumption and trigger performance degradation. Potentially vulnerable structures are detected using static analysis, and then these are tested dynamically to see if two different execution paths exist with a noticeable cost difference.

In 2023, Woo et al. introduced V1SCAN [28], an approach for discovering 1-day vulnerabilities in reused C/C++ open-source software. A 1-day vulnerability is known by the software developers and patched. V1SCAN process can be described in three phases:

[3] `https://xenproject.org/`
[4] `https://www.seamonkey-project.org/`
[5] `https://github.com/libav/libav`

1. Component identification: The authors try to find OSS components in a program. This process is done using an existing tool called CENTRIS since it can precisely identify them. Once a list of OSS components is found, V1SCAN tries to determine the version of each component by comparing the functions with those included in the target program: the most frequently identified version is marked as prevalent. A function can be classified as exactly reused if it has not been modified, changed if it has a similar function to the original one, and unused if it is not present inside the program.

2. Detection: The authors made a database that maps the various CVEs to the affected OSS versions. Thanks to this, V1SCAN can identify the CVE vulnerabilities included in the prevalent version of the OSS component.

3. In consolidation, the result of the previous phases is consolidated to cover false negatives, and finally, the list of vulnerabilities in the target programs is reported.

The authors report that testing V1SCAN on popular C/C++ projects from GitHub showed that the tool is capable of detecting 50% more vulnerabilities when compared to state-of-the-art tools. More precisely, the authors picked 100 random C/C++ repositories with more than 1000 starts on GitHub. In these, V1SCAN found 73 vulnerabilities that were successfully reproduced.

Author	Year	Compatible programming languages	Source Code
Monoidics, and Meta	2013	C/C++, C#, Java	Yes
Shuai et al. [24]	2015	Not specified	No
Li et al. [25]	2018	C/C++	Yes
Li et al. [26]	2021	C/C++	Yes
Liu et al. [27]	2022	Java	No
Woo et al. [28]	2023	C/C++	Yes

Table 3.1: Non Graph-based approaches

3.2 Graph-based approaches

When discussing graph-based approaches, it is crucial to recognize the contributions made during the years by Yamaguchi et al. with the introduction of Joern.

In 2014, Yamaguchi et al. introduced Joern [29], a tool used extensively in current vulnerability detection approaches. In the paper, the authors present the concept of Code Property Graphs (CPG), a single graph that combines Abstract Syntax Trees(AST), Control Flow Graphs(CFG), and Program Dependence Graphs(PDG). This graph combination was created to characterize vulnerabilities correctly. By leveraging graph traversal on the CPG, it is possible to identify security flaws. For example, the authors tested one of Joern's first versions on the Linux kernel. From testing, the authors found ten out of twelve vulnerability types reported as present in the Linux source code from 2012, just by employing the CPG. Overall, the authors identified 18 previously unknown vulnerabilities. Joern is still actively developed and supports multiple programming languages, such as C/C++, Java, Kotlin, and Python. Recent versions also include a code scanner called Joern Scan[6], which can run queries on the CPG to find vulnerabilities more quickly; these queries come from a public database[7] where everyone is open to contributing.

In 2018, Galois Inc.[8], Trail of Bits[9], and Dr. Stephen Chong's lab at Harvard University [10] started working on MATE: Merged Analysis To prevent Exploits.[11] Initially developed as part of the US DARPA CHESS program[12], and released to the public in 2022, it is a series of tools to perform interactive program analysis in C and C++ programs using Code Property Graphs (CPGs). MATE also provides a web app to visualize the vulnerabilities.

In 2019, Zhou et al. presented Devign [30], an approach based on a graph neural network and a manually labeled dataset[13] that includes code from the Linux Kernel[14], QEMU[15], Wireshark[16] and FFmpeg[17]. To do this, the authors use a composite code representation based on graphs, precisely the Abstract Syntax Tree(AST), the Control Flow Graph(CFG), and the Data Flow Graph(DFG), all

[6] `https://docs.joern.io/scan/`
[7] `https://queries.joern.io/`
[8] `https://galois.com`
[9] `https://www.trailofbits.com/`
[10] `https://people.seas.harvard.edu/~chong/`
[11] `https://galoisinc.github.io/MATE/`
[12] Air Force and Defense Advanced Researched Research Project Agency
[13] `https://sites.google.com/view/devign`
[14] `https://github.com/torvalds/linux`
[15] `https://www.qemu.org/`
[16] `https://www.wireshark.org/`
[17] `https://www.ffmpeg.org/`

of which are extracted using Joern. To see if Devign can discover new vulnerabilities, the authors took ten of the latest CVEs for each test project. Looking at the corresponding commits that fix them, 112 vulnerable functions were counted. Devign achieved an average accuracy of 74,11%, indicating the potentiality of discovering new vulnerabilities in practical applications.

In 2020, Zhuang et al. (IBM Research Labs) did something similar [21], but rather than using a composite graph, they analyzed the different graphs (AST, CFG, and DFG) separately. The result of this is a model called 3GNN that was evaluated on the Draper [31] dataset[18] and QEMU[15]+FFmpeg[17]. As in the previous attempt, Joern was used to generate the graph representation of the code. The authors found that the proposed model outperforms several existing models, with a 6,9% better F1 score than GGNN in the Draper dataset[18].

In 2020, Bilgin et al. proposed an approach for detecting vulnerabilities as a binary classification task that relies only on the Abstract Syntax Tree [32]. The authors tested this approach on the Draper dataset[18], different functions from open-source projects such as Debian Linux[19], and more. The abstract syntax tree was generated using Pycparser[20], while the classification algorithm was implemented using Scikit-learn[21] and Tensorflow[22].

In 2021, Zhou et al. presented GraphEye [33], an approach for detecting vulnerabilities in C and C++ functions using Code Property Graphs (CPG) and a deep learning model. The Code Property Graph is a property graph that combines the Abstract Syntax Tree (AST), the Control Flow Graph (CFG), the Program Dependence Graph (PDG), and the Data Dependency Graph (DDG). Even in this case, Joern was a crucial tool for making things work as it was used to generate the Code Property Graph. The authors tested the approach on the SARD dataset[23], focusing mainly on three CWEs: Stack-based Buffer Overflow(121), Divide-Zero(369), and Null Pointer Deference(476). Two additional CWEs were selected for comparison with other models: Buffer Error(199) and Resource Management Error(399). For CWE 199, GraphEye had a false positive rate(FPR) of 9,4%, the lowest compared to other approaches such as Flawfinder, RATS, and a system based on natural language processing. It also outperformed those in other metrics such as the false negative rate(FNR), true positive rate(TPR), precision, and F1. The same applies to CWE 399, where the FPR is 0,4% with a precision of 99,3%.

In 2021, Haojie et al. presented Vulmg [34], an approach for detecting vul-

[18] `https://osf.io/d45bw/`
[19] `https://www.debian.org/`
[20] `https://github.com/eliben/pycparser/tree/master`
[21] `https://scikit-learn.org/stable/`
[22] `https://www.tensorflow.org/`
[23] `https://samate.nist.gov/SARD/`

nerabilities based on graph classification. The main difference from other approaches is that the authors realized that the vulnerabilities are not isolated in a single function but can also exist in the function that calls the vulnerable one. As usual, the Code Property Graph is generated using Joern. This time, it gets enhanced by adding the Function Call Graph(FCG) to associate the calling function with the called function to detect cross-function vulnerabilities. Vulmg was tested on the Juliet dataset[24], focusing on CWE 369 (Divide by Zero) and CWE 476(Null Pointer Dereference) with the usual metrics as false-positive rate(FPR), false-negative rate(FNR), true-positive rate(TPR), precision and F1 score. The proposed model, MGGAT, produced excellent results compared to similar approaches such as FlawFinder, RATS, BiLSTM, SVM, and CGN, with all metrics having the highest values on both test classes.

In 2021, Zou et al. introduced μVulDeePecker [35], an approach based on Bidirectional Long-Short Time Memory(BLSTM) that is capable of telling if a piece of code contains a vulnerability and, if yes, what the type, inspired by VulDeePecker [25]. The main difference between the two is that VulDeePecker cannot tell the type of a detected vulnerability since it is based on a binary classifier. μVulDeePecker introduces the concept of code attention, which takes inspiration from the image processing world. Essentially, some aspects of the code may reveal more information regarding potential vulnerabilities, which is exploited to collect more "localized" information. As usual, the graph representation of the various code snippets is generated using Joern. The authors use a new kind of graph called the System Dependency Graph(SDG), which derives from the Programm Dependence Graph(PDG). When tested with actual software (Libav, Seamonkey, Xen), μVulDeePecker managed to detect sixteen vulnerabilities, fourteen of which correspond to patterns of known vulnerabilities.

In 2022, Ding et al.(IBM Research Labs) presented VELVET [36], a novel ensemble learning approach to detect vulnerabilities that combine graph-based neural networks with sequence-based ones to have a better view of the local and global context of a program graph. This model was first trained with synthetic data from the Juliet Test Suite. Synthetic code is code written with the sole purpose of showing vulnerable patterns. The model was then finetuned with data from the D2A dataset containing real-world vulnerable code from different open-source projects. As usual, Joern was used to generate the Code Property Graphs. According to the tests performed by the authors, VELVET achieves 4,5x better performance than the baseline static analyzers on real-world data from the D2A dataset. The model also achieves 99,6% and 43,6% top-1 accuracy over synthetic and real-world data.

In 2022, Wu et al. presented VulCNN [37], an approach for detecting vul-

[24] `https://samate.nist.gov/SARD/test-suites/112`

nerabilities inspired by image classification models, which is done by converting the source code of a function into an image. This conversion is done by taking Joern's Program Dependency Graph(PDG) and treating it as a social network to apply social network centrality analysis. These images are then used to train the CNN model. The authors tested this using synthetic and real-world code from the SARD[23] and the NVD. From experimental results, the authors report that VulCNN can achieve better accuracy than eight state-of-the-art tools (both commercial and non), with higher scalability when compared to VulDeePecker [25] and SySeVR [26].

In 2023, Islam et al. [38] presented an approach to detecting vulnerabilities based on a semantic vulnerability graph (SVG). The authors decided to use an SVG since it integrates sequential flow to understand the syntactic of a program, the data flow graph to know how the data flows inside the program, and the control flow graph. Other than this, the authors introduce a new kind of edge called poacher flow that is supposed to provide more information for capturing vulnerabilities. These are meant to identify boundaries, corner cases, and external checkpoints, and the authors define them as a bridge between static and dynamic analysis. The authors ran experiments to determine how much the model can classify vulnerabilities, how the model can handle a biased dataset, and the detection of N-day and zero-day programs. For the dataset, authors have used a mix of synthetic and real-world code, along with VulF[25], a dataset created by the authors that combines source code from GitHub and the NVD. Experimental results report that this new approach outperformed state-of-the-art tools with fewer false negatives and false positives, with an improvement of at least 2,41% (18,75% in the best-case scenario).

In 2023, Dong et al. presented DeKeDVer [39], a multi-type vulnerability classifier that combines vulnerability descriptions and source code. The vulnerability descriptions are preprocessed and then used with TextRCNN to extract knowledge and generate representation vectors. The code instead is converted into a Code Property Graph (CPG) generated with Joern and then processed with RGAT, and the output of these two neural networks is combined to create the dataset. The motivation behind this approach is that the vulnerability description can enrich the source code and vice versa; for example, the description can be used to narrow the analysis scope. This model was trained on a custom dataset of 5089 samples that contains vulnerabilities stored in the NVD from 2004 to 2019. The authors discarded the vulnerabilities without a CWE and those with a RESERVED, DISPUTED, or REJECT state. From the author's testing, this model achieves 84,49% in weighted F1-measure, proving to be more effective when compared to other ap-

[25] `https://drive.google.com/drive/folders/1d00kfEX6k1MhpxJtuFv5JqtlQTJfg03N?usp=sharing`

proaches.

In 2023, Mirksy et al. introduced VulChecker [40], a deep-learning approach for detecting the precise manifestation point of vulnerabilities in source code. The analysis is done using a custom LLVM[26] compiler to generate a GNN-optimized representation of the program. The authors refer to this as an Enriched Program Dependency Graph (ePDG). VulChecker performs the analysis at the Intermediate Representation (IR) layer. This means that this approach could potentially be compatible not only with C/C++ but also with other languages, such as Objective-C and Rust since LLVM supports them. Vulchecker uses a specific model for each CWE since they require different manifestation points. The pipeline is composed as follows:

- Generation of the ePDG, done by compiling the source code into the LLVM IR using a custom plugin.
- Sampling, the ePDG is scanned to locate potential manifestation points, so instructions that might lead to that specific vulnerability.
- Feature extraction.
- Training/Execution of the model.

Using VulChecker on 19 C++ projects, the authors found 24 vulnerabilities (reported CVEs).

[26] `https://llvm.org/`

Author	Year	Compatible programming languages	Employed Graphs	Source Code
Galois Inc. et al	2018	C/C++	CPG	Yes
Zhou et al. [30]	2019	C/C++	AST, CFG, DFG	Yes
Zhuang et al. [21]	2020	C/C++	AST, CFG, DFG	No
Bilgin et al. [32]	2020	C/C++	AST	No
Zhou et al. [33]	2021	C/C++	CPG	No
Haojie et al. [34]	2021	C/C++	CPG, FCG	No
Zou et al. [35]	2021	C/C++	SDG	No
Ding et al. [36]	2022	C/C++	CPG	Yes
Wu et al. [37]	2022	C/C++	PDG	Yes
Islam et al [38]	2023	C/C++	SVG	Yes
Dong et al. [39]	2023	C/C++	CPG	No
Mirsky et al. [40]	2023	C/C++, potentially Objective-C, Fortran, Rust	ePDG	Yes

Table 3.2: Graph-based approaches

Chapter 4

Methodology

4.1 Meta-Programming

Meta-programming indicates the development of meta-programs, a family of programs that uses other programs as input data. Meta-programs can be used to analyze, transform, and generate other programs [41] in a desired object language [42]. Meta-programs include tools for analyzing programs written in common programming languages like C or Java and migrating large-scale C/C++ legacy test code [43].

This chapter focuses on Rascal[1], a meta-programming language developed by UseTheSource, an organization from the Centrum Wiskunde & Informatica(CWI) [2].

4.2 Rascal

In 2009, Klint et al. [44] introduced Rascal, a domain-specific meta-language for source code analysis and manipulation. The authors define Rascal as a language that "covers the range of applications from pure analyses to pure transformations and everything in between.". [44] The authors developed Rascal to achieve the following goals:

- Remove the overhead needed for integrating source code analysis tools and manipulation ones.
- Provide a safe and interactive environment capable of working with large code bases.
- Be easily understandable to a large group of computer programming experts.

[1] `https://www.rascal-mpl.org/`
[2] `https://www.cwi.nl/en/`

Rascal allows users to write programs that perform static analysis, generate new code, and design new domain-specific languages. Since this thesis does not focus on the latter two use cases, here are some examples from the Rascal documentation.

Listing 2 shows an example of a Rascal function that can transform the "style" of a Java code snippet. In particular, this improves the readability of if statements by removing negated conditionals (line 7), introducing braces for the if block (line 10), and using a single return statement (line 18) by using pattern matching on a parse tree. In Listing 3, it is possible to see an output example of the function `idiomatic`.

```
1  module Idiomatic
2
3  import lang::java::\syntax::Java15;
4
5  CompilationUnit idiomatic(CompilationUnit unit) = innermost
6  visit(unit) {
7     case (Stm) `if (!<Expr cond>) <Stm a> else <Stm b>` =>
8          (Stm) `if (<Expr cond>)  <Stm b> else <Stm a>`
9
10    case (Stm) `if (<Expr cond>) <Stm a>` =>
11         (Stm) `if (<Expr cond>) { <Stm a> }`
12      when (Stm) `<Block _>` !:= a
13
14    case (Stm) `if (<Expr cond>) <Stm a> else <Stm b>` =>
15         (Stm) `if (<Expr cond>) { <Stm a> } else { <Stm b> }`
16      when (Stm) `<Block _>` !:= a
17
18    case (Stm) `if (<Expr cond>) { return true; } else { return
      ↪  false; }` =>
19         (Stm) `return <Expr cond>;`
20 };
```

Listing 2: Source-to-source transformation, from `https://www.rascal-mpl.org/docs/WhyRascal/UseCases/SourceToSource/`

```
class MyClass {
    int m() {
        if (!x)
            println("x");
        else
            println("y");
        if (x)
            return true;
        else
            return false;
    }
}
```

```
class MyClass {
  int m() {
    if (x) {
      println("y");
    } else {
      println("x");
    }
    return x;
  }
}
```

Listing 3: Example input of the function `idiomatic` defined in listing 2 with the relative output

Listing 4 shows an example of how Rascal can be used to define the syntax for a Domain-Specific Language(DSL). The DSL is used to model a state machine in this specific example. Lines 6 to 8 define a simple state machine of states and transitions. In listing 5, there is an example of a function that can be used to check the semantics of a state machine; in particular, this function can allow one to find all unreachable states. This is done by creating binary relations between states using comprehension.

```
module Syntax

extend lang::std::Layout;
extend lang::std::Id;

start syntax Machine = machine: State+ states;
syntax State = state: "state" Id name Trans* out;
syntax Trans = trans: Id event ":" Id to;
```

Listing 4: Definition of a syntax for a state machine, from `https://www.rascal-mpl.org/docs/WhyRascal/UseCases/DomainSpecificLanguages`

One of the advantages of Rascal stands in its modularity. Over the years, the authors expanded the language functionalities by introducing new modules, one of them being Clair, which enables Rascal to perform analysis of C and C++ programs.

```
module Analyze

import Syntax;

set[str] unreachable(Machine m) {
  r = { <"<q1>","<q2>"> | (State)`state <Id q1> <Trans* ts>` <-
   ↪ m.states,
                  (Trans)`<Id _>: <Id q2>` <- ts }+;
  qs = [ "<q.name>" | q <- m.states ];
  return { q | q <- qs, q notin r[qs[0]] } - qs[0];
}
```

Listing 5: Example of a function used to process parse trees from `https://www.rascal-mpl.org/docs/WhyRascal/UseCases/DomainSpecificLanguages`

4.3 M^3 model

In 2013, Izmaylova et al. [45], later refined in 2015 by Basten et al. [46] introduced the M^3 model used by Rascal, an essential piece of this work. Inspired by other models such as FAMIX [47], RSF [48], GXL [49], KDM [50], and ATerms [51], what is the main difference? M^3 is composed of purely immutable, typed data, something that does not happen in the other models used as inspiration. It introduces URI literals to identify source code artifacts in a language-agnostic way and support fully structured type symbols. The model comprises a hierarchical layer containing the Abstract Syntax tree and a relational (flat) layer, the relationship which is explained in table 4.1. In the paper, the authors focus on several design aspects. One is that the M^3 model must be fully typed and serializable as readable text (the same thing applies to all Rascal data). This requirement does not apply to the input source code. Another factor relates to linking the information stored in the model to the source of information: the source code. By taking inspiration from the semantic web, the authors use URIs to identify any artifact. These can refer to a physical or logical code location. With physical code locations, they refer to the actual storage location of the source code file. This can be represented with a relative or absolute path. For example, `|file:////tmp/Hello.java|`. Logical code locations instead are modeled with a specific naming scheme for each programming language. For example, `|java+class://myProject/java/util/List|`. The M^3 model uses binary relations between locations, as shown in Table 4.1. It annotates AST nodes with these locations to link to the source code whenever possible. The M^3 model is both layered and compositional. It can be combined ("linked") and

extended ("annotated"). This model was developed to be easy to construct, extend to include language specifics, and consume to produce metrics or perform analysis on top of it. When the model was developed, the authors wanted the model to be generic enough to work with multiple programming languages without losing accuracy: to achieve this, the M^3 standard prescribes language-specific ASTs and then language-agnostic relational models that can be more abstract. This core model can then be expanded to support the modeling of language-specific information by adding extra relations that model key aspects of the static semantics of a programming language.

Layer	`rel[loc,loc]`	Description
Generic Core	containment	Logical containment of code entities
Generic Core	declarations	Map logical `loc` to physical `loc`
Generic Core	uses	Map physical `loc` to logical `loc`
OO	extends	Who extends who (logical `loc`)
OO	implements	Who implements who (logical `loc`)
OO	methodInvocation	Who invokes who (logical `loc`)
OO	methodOverrides	Who overrides who (logical `loc`)
OO	fieldAccess	Who accesses who (logical `loc`)

Table 4.1: Core binary relations of M^3

The extraction of the M^3 models is done incrementally, file by file. These can be joined into one by performing a set union of all the model's relations. Rascal provides a function to do this called `composeM3`.

4.4 Clair

As mentioned, Rascal is a modular language that can be expanded using different modules. One of them is Clair [52], the source of code of which is publicly available on GitHub[3]. Clair allows Rascal to process C and C++ programs. This module uses Eclipse CDT[4] to parse the code and produce an Abstract Syntax Tree [43]. Clair then converts this AST into a format processable with Rascal. The AST is also used to enrich the M^3 model. Clair introduces a specialized version of the M^3

[3] `https://github.com/usethesource/clair`
[4] `https://projects.eclipse.org/projects/tools.cdt`

model for C and C++ source code that expands the core model explained in the previous section. This specialized model is illustrated in table 4.2. One of these new fields is `declaredType`. This one is used to retrieve functions, methods, and variables along with their associated type. Another one is `functionDefinitions`, which is used to retrieve the location of methods and functions.

Table 4.2: Relations of M^3 provided by Clair

Field	**Data Structure**	**Description**
implicitDelcarations	`set[loc]`	A set of locations where a implicit declaration (`new`) is present
typeDependency	`rel[loc caller, loc typeName]`	Relation used to indicate the usage of type literals
macroExpansions	`rel[loc file, loc macro]`	Relation used to indicate the location where a macro is used, so the macro invocation gets substituted with the actual value.
macroDefinitions	`rel[loc macro, loc src]`	Relation used to indicate the location where a macro is defined.
includeDirectives	`rel[loc directive, loc occurrence]`	Relation used to indicate where (occurrence) an include (directive) is used.
inactiveIncludes	`rel[loc directive, loc occurrence]`	Relation used to indicate include Directives that might be inactive.
includeResolution	`rel[loc directive, loc resolved]`	Relation used to indicate the resolved path of an include directive.

Continued on next page

Table 4.2 – continued from previous page

Field	Data Structure	Description
declaredType	`rel[loc decl, TypeSymbol typ]`	Relation used to map a declaration (functions, methods, variables...) to the respective type. This will include the return and parameter types for functions and methods.
memberAccessModifiers	`rel[loc decl, loc visiblity]`	Relation used to map a declaration to the respective access modifier.
functionDefinitions	`rel[loc name, loc src]`	Maps the logical name of a function to the location where it is declared.
cFunctionsToNoArgs	`rel[loc decl, loc src]`	Relation that maps a function declaration to a version with no arguments.
declarationToDefinition	`rel[loc decl, loc impl]`	Relation used to map a declaration to its definition.
unresolvedIncludes	`rel[loc directive, loc file]`	Relation used to indicate all those include directives whose locations could not be resolved.
comments	`list[loc]`	A list of locations that includes a comment.
requires	`rel[loc includer, loc includee]`	Maps the location of a source file (includer) to the location of the header file (included) included in the source file.

Continued on next page

Table 4.2 – continued from previous page

Field	Data Structure	Description
provides	`rel[loc provider, loc providee]`	Relation that maps a provider(definition) to a provided(declaration).
partOf	`rel[loc decl, loc file]`	Relation used to map a declaration to the file that contains it.
callGraph	`rel[loc caller, loc callee]`	Maps the location where a function (this also includes C++ methods and constructors) is called (caller) to the location in which the function is provided (callee).

4.5 ClassViz, the visualization tool

This thesis project is based on Satrio Adi Rukmono's [5] Ph.D. project, a system that automatically answers software developers' questions using only architectural knowledge from the source code.

The project is composed of different building blocks. I worked on the knowledge extractors and knowledge presenters. Figure 4.1 shows a schematized version of the complete workflow.

4.5.1 Knowledge extractors.

In [13], Rukmono and Chaudron define the knowledge representation model used in the various tools. This representation needs to satisfy the following requirements:

1. **Abstract & Concrete.** The knowledge representation must support an abstract view free from the implementation details. This means that the representation may not be convertible into source code. At the same time, this representation should be connected to the concrete implementation.

[5] `https://github.com/rsatrioadi/phd`

2. **Rich In relational information.** This knowledge representation should contain more information than the typical dependency graph, such as specialization and composition.

3. **Language-agnostic.** This knowledge representation must be language-agnostic, so it should not work only for a specific programming language. Note that the work present in this paper included only class-based object-oriented software systems.

4. **Handles partial and evolving knowledge.** This representation captures partial knowledge from models that may be incomplete. A representation may be incomplete for different reasons, one of which might be related to the capabilities of the chosen knowledge extractor.

5. **Supports multiple knowledge sources.** The knowledge may come from different sources, such as the software source code, UML models, and architecture documents. All of these sources should be combined in a single representation.

6. **Enrichable.** This knowledge representation must be flexible enough to be enriched with new types of knowledge. This requirement made it possible to add vulnerability information.

This results in a labeled property graph(LPG) with two levels of abstraction used according to different needs. The first level is the detailed one, which contains concepts such as types, classes, fields, and methods. All of these are represented as graph nodes. The second one uses a higher abstraction level and only considers containers and types (this includes classes). Inside the first level of abstraction, there are the following kinds of nodes:

- *Structure:* a class type; under this node, there are elements known as structs, records, and enumerations.

- *Container:* nodes representing any element from the language that can contain structures or allow their organization, such as packages, directories, modules, or namespaces.

- *Variable:* Self-explanatory can include fields, attributes, properties, function/method, and parameters.

- *Script:* Code expression or a block of statements.

- *Operation:* A Script that can be identified with a signature (name and parameters) and has a return type.

- *Constructor:* An Operation used to create an instance of a Structure.

Regarding the edges, there are:

- *Specializes,* this edge represents inheritance between two nodes. This edge also includes the implementation of an interface.
- *Invokes,* this edge represents invocations to a node that has a signature.
- *Instantiates,* this edge is a convenience edge that can be derived from other elements in the graph.

Inside the LPG are also edge types such as *contains*, *type*, *returnType*, *variable*, *hasScript*, and *hasParameter*. These should be self-explanatory, given the name.

A tool called Javapers [53] was developed to implement this model to extract knowledge from Java programs. This tool performs static analysis over the source code using the Spoon library [54]. Javapers can export the LPG in different formats, including XML, CSV, and JSON. The JSON LPG complies with the Cytoscape JS [55] graph structure; Listing 6 shows a minimal example. One of the goals of this thesis is to recreate this LPG but for the C/C++ source code. This goal led to developing a tool for converting the M^3 model produced by Rascal and the Clair library needed for parsing C/C++ source code.

The tool is written in Python, and it performs these tasks:

- **Generation of the M^3 model.** The generation is done by executing a script written using Rascal. An example script can be seen in the Appendix A.1. This script makes use of tree modules: `IO,lang::json::IO`, which comes from the Rascal standard library, and `lang::cpp::M3`, which comes from the Clair library. The script is composed of only one function called `main`, responsible for all the work, structured in a way that allows passing arguments when executing the script through the terminal. In particular, it takes the following arguments:
  - `cpp`, a boolean flag used to indicate whether the script will work with a C or C++ program,
  - `srcPath`, a variable of kind `loc` used to indicate the path of the folder containing the source files,
  - `names`, a list of strings containing the name of the source files that need to be analyzed. The generation process of the M^3 model requires the analysis of each file involved in the program,
  - `stdLib`, a list of `loc` used to indicate where the standard libraries are located. This changes along the host operating system that is used during the execution of the script so it can not be hardcoded,

```
{
    "elements": {
        "nodes": [
            {
                "data": {
                    "id": 1,
                    "labels": [
                        "label1",
                        "..."
                    ],
                    "properties": {
                        "amazing_key": "value"
                    }
                }
            }
        ],
        "edges": [
            {
                "data": {
                    "id": 123,
                    "source": 0,
                    "target": 0,
                    "label": "<edge_label>",
                    "properties": {
                        "amazing_key": "value"
                    }
                }
            }
        ]
    }
}
```

Listing 6: Example of an LPG compliant with Cytoscape

– `includeDir`, a list of `loc` used to indicate the location of files that might be included in the source code.

The `cpp` flag is used to determine which function the script needs to use to generate the M3 models: `createM3FromCppFile` for C++ programs, and `createM3FromCFile` for programs written in C. Note that it is possible to

generate an M^3 model for C programs by using the function reserved for C++. At the same time, this does not guarantee an accurate result. Given the early state of Clair, there might be a lack of information in the M^3 models generated using `createM3FromCFile`. For example, while writing this thesis, it was noticed that information about the return type and the parameters used in C functions was missing. These and other issues have been reported to the developers, who quickly responded to the GitHub issues.

The script uses map comprehensions to generate a model for each file, `name:tup|name <-names,tup:=createM3FromCppFile`.

With one line of code, the script can generate the M^3 model for each file, indicated by its name. The models are stored in a variable of kind `Map`, where each name is mapped to the respective model. The models are then composed into a single one by using the `composeCppM3` function, which takes in input two parameters: a `loc` that is used to identify one composed model to another one and a set of $M^3 models$. The composition considers each model field as a set to perform a union operation. After this, the script exports the composed model as a JSON file using `writeJSON` function from the `lang::json::IO` module.

- **Vulnerability extraction** The goal of this project is to develop a graphical visualization tool for potential vulnerabilities that may be present in a program. To do this, it is necessary to extract the output from code analyzers to add this information to the knowledge graph. This is done by implementing an "extraction" function specific to a code analyzer. For testing purposes, tree extractors have been implemented for the following analyzers: MATE, Joern Scan, and Infer.

  - Extraction from MATE: this tool provides a REST API to extract all information. In particular, the endpoint `/api/v1/pois/build/{build_id}` has been used to retrieve all the Points of Interest(POIs) for a specific build. In case of a successful request, this endpoint returns a list of the potential vulnerabilities in that program. For each vulnerability, it provides a name along with a brief textual description, plus the source and the target that are involved. A POI example is available in Appendix B.1.
  - Extraction from Joern Scan: this tool does not provide any API to retrieve the analysis output but instead saves the complete output (including other output messages not strictly related to the analysis) to a temporary file. To overcome this, the user must redirect the scan output to a text file. This output is then cleaned and parsed. An output example is available in Appendix C.1.

– Extraction from Infer: this tool automatically saves the result of the analysis in a JSON file. An output example is available in Appendix D.1.

In all cases, the output is parsed and processed to comply with the vulnerability structure, as shown in Listing 7.

```
{
    "analysis_name": "name",
    "target": {
        "file": "file_name",
        "location": "line:col",
        "script": "name of the target"
    },
    "description": "text description"
}
```

Listing 7: Example of the vulnerability structure

- **Conversion of the M^3 model.** The JSON file containing the composed M^3 model is then parsed with a Python script that extracts the knowledge from it and converts it into a format compliant with the one accepted by Cytoscape. The process is almost the same for C and C++ models, and the only difference comes from the structures that are not present in C due to not being an object-oriented programming language.

  The conversion process starts by cleaning the exported model by removing elements that include error messages. This is done to prevent failures in the conversion process. Error messages usually come from not having the proper values for the `stdLib` or `includeDir` variables, but also from problems that may arise from Clair itself. The cleaned model is saved to a new file so the user can compare it, see what is missing, and evaluate the resulting information loss.

  Once the model has been cleaned, the conversion process can start. The tool will create an instance of a class that is appropriate to the programming language of the program that the user is trying to analyze and visualize. For both C and C++, the class constructor takes as input the output path, used to store the converted model, the parsed JSON file that contains the cleaned model, and an optional array of vulnerabilities. In this class, there is a list of primitive types that will be rendered as nodes among the other types and a dictionary used to store the converted model; this will have the same structure shown in the Listing 6.

Each node contains:

- an ID that is understandable by a human; this might be the name of the element or a combination of more names (e.g., "name of the function.name of the variable")
- a series of labels that are used to identify the node in line with what is defined in [13], plus other information according to the application needs,
- of properties used to modify how a node is rendered. This usually includes the name shown in the visualization tool plus a kind field more specific than the one defined in the labels.

Edges instead are characterized by:

- a numeric ID generated using a hash function with different properties of the element as input,
- a source node, identified by its ID,
- a target node, identified by its ID,
- a dictionary of properties, which usually includes the weight of the edge,
- a set of labels that are used to identify the edge in line with what is defined in [13].

The conversion process starts by creating a `Container` node for each file used to generate the model. Container nodes are particular nodes rendered differently by the visualization tool used to aggregate different nodes on their inside. Files are retrieved from the `provides` field of the M^3 model.

Classes are extracted from the field `containment` with the help of `functionDefinitions` to retrieve the location(file, line, and column). Along these two, the field `extends` determines whether a class expands another; if so, a `specializes` edge will be added to link the two nodes.

The script uses different fields of the M^3 model to extract functions and methods to retrieve all the information about them. From `declaredType`, the script retrieves the list of functions/methods, the return type, the types of potential parameters that might be used to call the function/method, and the variables. Functions/methods location is retrieved from the field `functionDefinitions`, while the names of the parameters are retrieved from `declarations`. After this, a node is created for each function/method, along with the nodes for the parameters and the variables. These are linked using the edges `hasParameter, variable, returnType`; methods also use an additional edge `hasScript` to connect them to the respective class.

When creating these new nodes, a `contains` edge is collocated in the correct container node. The list of all the functions and methods is then used to reconstruct a call graph with the help of the field `callGraph`. The script also checks for vulnerabilities in the provided analysis result while creating the nodes for functions, methods, or classes. If a match is found, these get added to the vulnerabilities array stored in the node's properties, along with the `vulnerable` label, used by the visualization tool to render the impacted nodes differently.

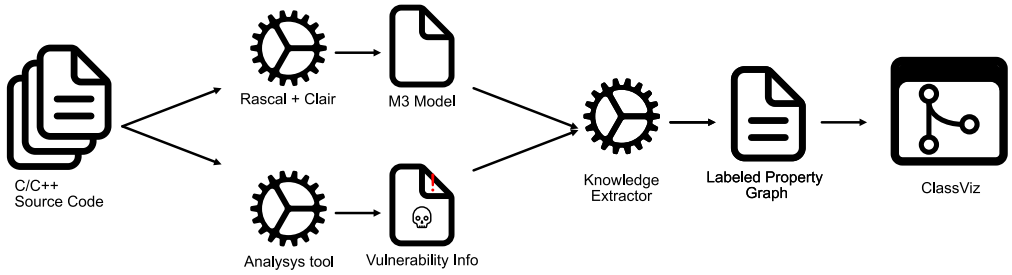

Figure 4.1: Complete workflow, from start to finish.

4.5.2 Knowledge presenters

To visualize the labeled property graphs produced in Section 4.5.1, Satrio Adi Rukmono developed an interactive web app [6] called ClassViz. The initial purpose of this visualization tool was to visualize software classes, hence the name, but nowadays, it offers more functionalities. ClassViz uses the javascript library Cytoscape JS to render the graph, so the labeled LPG must comply with its graph structure. The original version of the tool can be used to visualize Java projects along with some added information, such as role stereotypes [7], trace information from dynamic analysis, and summaries about a specific class generated using Large Language Models(LLMs) [56]. Given this, the tool has been modified to visualize the potential vulnerabilities.

[6] `https://github.com/rsatrioadi/classviz`

[7] Role stereotypes are a way to classify classes with an abstract characterization of their responsibility.

Chapter 5

Results

When users open ClassViz, they must upload the previously produced JSON file to visualize it. File upload can be done via the OS file picker accessible from the navbar. The User Interface(UI) is composed of four main components:

- **Navbar**, from here, the user can open a new file, save the visualization as an SVG file, or open it in a new tab.
- **Sidebar** comprises two tabs: General and Vulnerabilities.
  - **General**, from here, the user will be able to turn on or off nodes that represent primitives or packages, turn on or off the edges that represent the relationships defined in the previous chapter, and change how those edges should be rendered by choosing between Orthogonal or Bezier edges. Additionally, it is possible to choose a different layout algorithm, highlight specific nodes, and expand or collapse container nodes. Figure 5.1 shows an example.
  - **Vulnerabilities**, from here, the user can filter the nodes by the vulnerability type. Each vulnerability type is associated with a color, which will be used as a background color for the impacted nodes. Figure 5.2 shows an example of this. If a node contains more than one vulnerability kind, a gradient made of the associated colors will be used as a background. Figure 5.5 shows an example of the coloring system in action.
- **Infobox** is a fixed information box showing information about a node. It contains the name and what kind of element the node represents, and if marked as vulnerable. It also includes a list of the found vulnerabilities along with a counter that describes how many times it has been found. The infobox content is updated every time a user hovers in a node and can be hidden by clicking on it. Figure 5.3 shows an example of this.

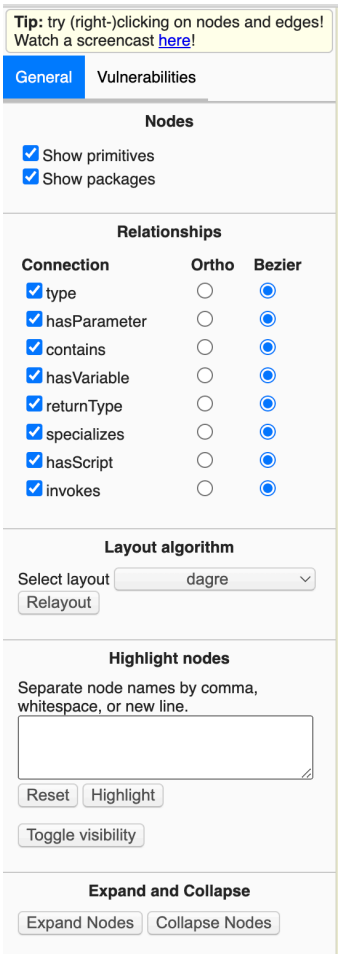

Figure 5.1: General tab of the sidebar.

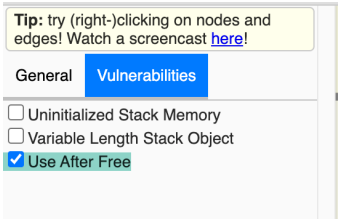

Figure 5.2: Vulnerabilities tab of the sidebar.

- **Canvas** is a crucial component of the UI. Implemented using Cytoscape JS allows the user to interact with the knowledge graph. The user can pan and zoom through the graph, move a node in a different location, or hide it by simply clicking on it with the mouse's right button. It is also possible to hide edges. Nodes are rendered with different shapes according

Figure 5.3: Infobox used to show information about the function `importaSalvataggio` from a C project.

to their kind: Variables (defined in 4.5.1) are shaped as an ellipse, and Scripts are shaped as rectangles instead. A node labeled as vulnerable will be rendered with a thick red border to alert the user. When hovering in one of these nodes, it is also possible to see a tooltip with a textual description of the vulnerabilities. Figure 5.4 shows an example of the canvas, while Figure 5.5 shows an example of the tooltip and the corresponding vulnerable code snippet from a project written in C that emulates a tabletop game.

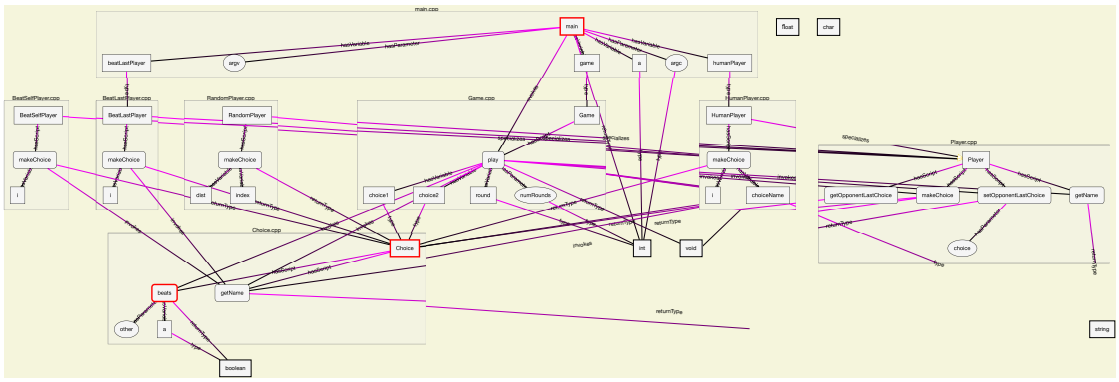

Figure 5.4: Example of a simple C++ project inside ClassViz.

```
...
    if (app->next == NULL) {
        prev = app->prev;
        prev->next = NULL;
        free(app);
    } else {
        prev = app->prev;
        prev->next = app->next;
        prev->next->prev = prev;
        free(app);
    }
}
app->c.idCarta=10;
```

Figure 5.5: Example of a tooltip inside ClassViz along with the corresponding C code snippet for the function `recuperaCarta`. The code snippet shows the code for the first "Use After Free" vulnerability.

The tool can be used with smaller projects, as shown in Figure 5.4, but also with bigger ones. In Figure 5.6, it is possible to see Curl as a graph.

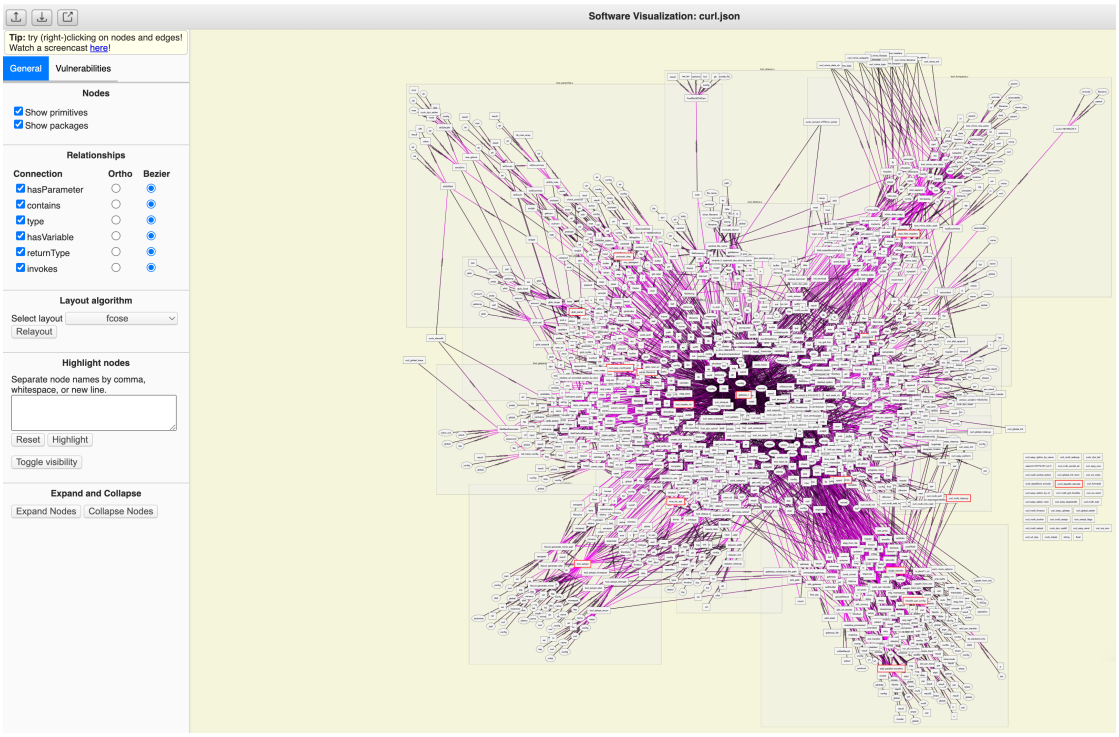

Figure 5.6: Curl rendered inside ClassViz

5.1 Comparison with MATE

As mentioned in Section 3.2, MATE also provides a visualization tool. It can be used to visualize the whole program as a graph, better understand how the different components are connected, and visualize the subgraph containing the nodes involved in a potential vulnerability, which MATE calls a Point Of Interest (POI). For the first use case, the user must ”construct” the graph by manually indicating the wanted nodes. As shown in Figure 5.7, the user can start choosing a function as the primary node, then expand by adding other nodes manually or using the menu accessible via a right-click on the node. The edges are color-coded to distinguish the different kinds of relations. A legend is placed on top of the graph to understand them. The POI visualization uses the same interface but to visualize the portion of the program involved in a specific POI. In addition, the UI also provides some insights that contain a brief textual description of the vulnerability, along with some background information. The downsides of this view are mainly two: at a glance, it is not possible to see the vulnerable nodes inside the global context of the program; also, by default, MATE shows a highly detailed graph based on the LLVM bytecode generated from the source code. Figure 5.8

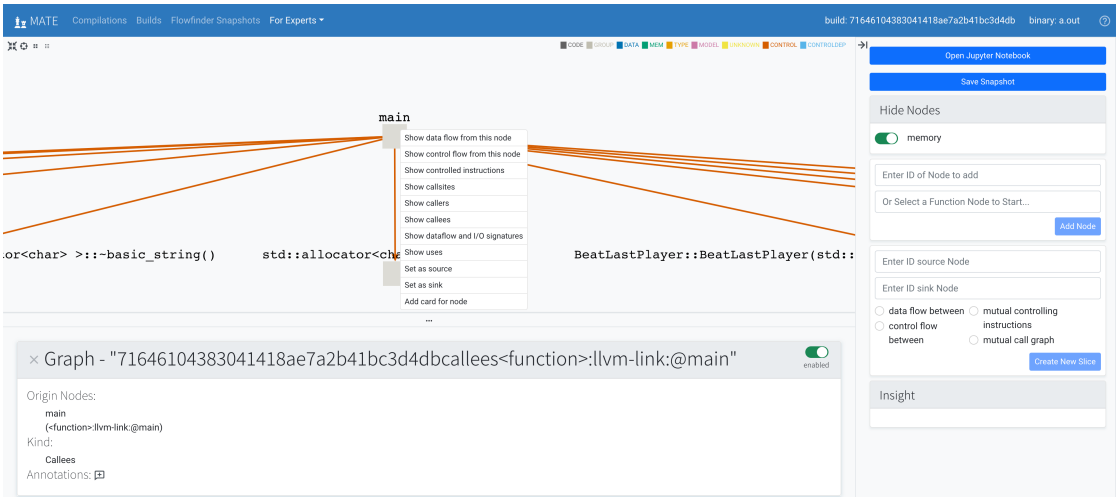

Figure 5.7: MATE Flowfinder UI

shows an example of this. While this can be useful, it might not be understandable for those who lack the required skills. MATE also provides a compact version of the subgraph; this example can be seen in figure 5.9.

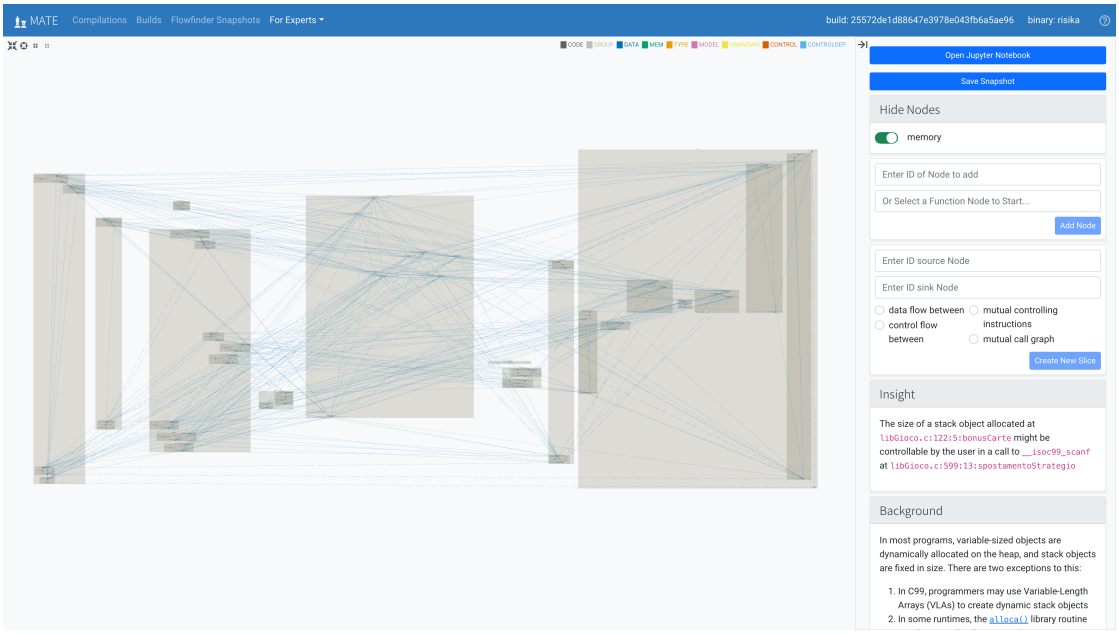

Figure 5.8: MATE Flowfinder UI, showing a potential vulnerability in a C program.

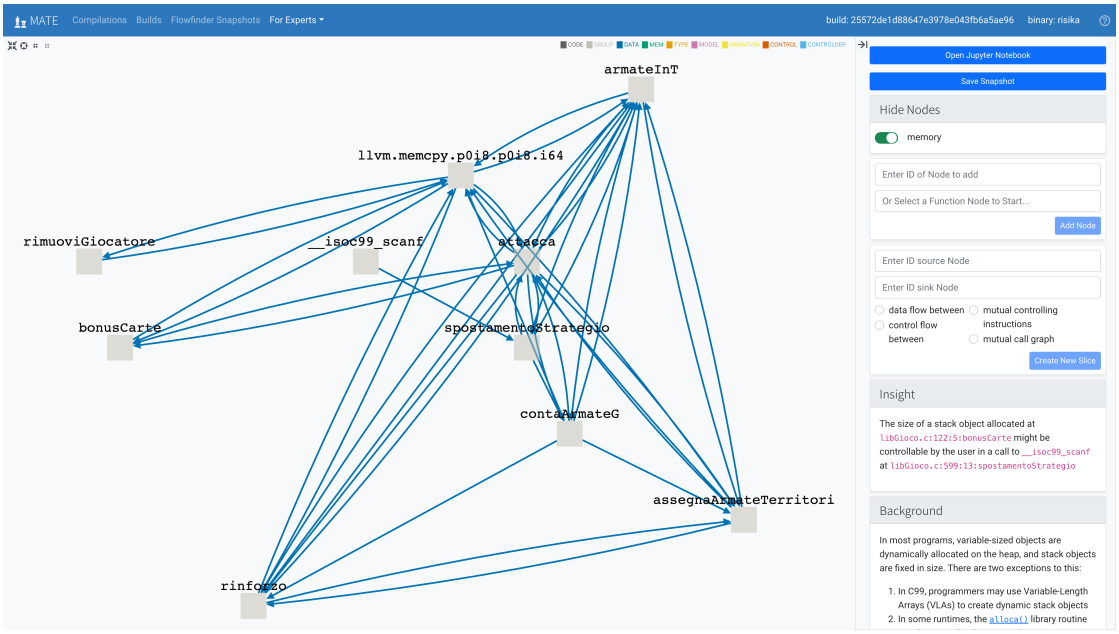

Figure 5.9: MATE Flowfinder UI, showing a compact version of the subgraph present in figure 5.8

Chapter 6

Conclusions

The importance of caring about vulnerability analysis and improving its usability aspects should be evident. This thesis introduces an initial effort to solve this problem.

Some of the more recent tools listed in Chapter 3 were considered for running some experiments. Unfortunately, it was not always possible, mainly for two reasons: lack of publicly available source code or lack of proper instructions for usage. Trying the tool on external programs outside the training or testing dataset was sometimes impossible.

Regarding ClassViz, there is still room for improvement, particularly regarding the layout algorithms: current ones do work, but they do not produce a decent result when used with large graphs, like the one in figure 5.6. A possible solution could be to develop a new layout algorithm or change the web app's visualization library.

The same thing applies to the knowledge extractor. Clair is still under development, and as stated in subsection 4.5.1, there might be a lack of information when generating a M^3 model for C programs using the designated function. For example, types related to a variable or a field are not correctly recognized among several error messages from the Eclipse CDT parser that are returned to the user. The library developers are aware of this problem, which will hopefully be solved in a future release of Clair. The extractor script source code is available on GitHub [1] and is open to any contribution but also criticism that might come from others. Possible future developments include the development of additional extractors for different vulnerability analysis tools and updates to the script that converts the M3 model in case of breaking changes that might come from future versions of Clair.

[1] `https://github.com/matteasu/MSc-thesis`

6.1 Acknowledgements

Thanks to Professor Michel R.V. Chaudron and his Ph.D. student Satrio Adi Rukmono for three months of hosting at Eindhoven University of Technology (The Netherlands) as part of the Erasmus+ Traineeship program.

Thanks to the Rascal team, particularly Professor Jurgen J. Vinju, for providing help and insights on using Rascal and the Clair library.

Bibliography

[1] R. Bayindir, I. Colak, G. Fulli, and K. Demirtas, "Smart grid technologies and applications," *Renewable and Sustainable Energy Reviews*, vol. 66, pp. 499–516, Dec. 1, 2016, ISSN: 1364-0321. DOI: 10.1016/j.rser.2016.08.002.

[2] "We are social & meltwater (2023), "digital 2023 global overview report"," DataReportal – Global Digital Insights. (Jan. 26, 2023), [Online]. Available: https://datareportal.com/reports/digital-2023-global-overview-report (visited on 12/27/2023).

[3] A. Anwar, A. Khormali, J. Choi, *et al.*, "Measuring the cost of software vulnerabilities," *ICST Transactions on Security and Safety*, vol. 7, no. 23, p. 164 551, Jun. 30, 2020, ISSN: 2032-9393. DOI: 10.4108/eai.13-7-2018.164551.

[4] M. Zugec. "Technical advisory: Zero-day critical vulnerability in log4j2 exploited in the wild," Bitdefender Blog. (), [Online]. Available: https://www.bitdefender.com/blog/businessinsights/technical-advisory-zero-day-critical-vulnerability-in-log4j2-exploited-in-the-wild/ (visited on 01/15/2024).

[5] S. L. Reynolds, T. Mertz, S. Arzt, and J. Kohlhammer, "User-centered design of visualizations for software vulnerability reports," in *2021 IEEE Symposium on Visualization for Cyber Security (VizSec)*, New Orleans, LA, USA: IEEE, Oct. 2021, pp. 68–78, ISBN: 978-1-66542-085-3. DOI: 10.1109/VizSec53666.2021.00013.

[6] Z. Guo, T. Tan, S. Liu, *et al.*, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Transactions on Software Engineering*, vol. 49, no. 12, pp. 5154–5188, Dec. 2023, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3329667.

[7] H. Assal, S. Chiasson, and R. Biddle, "Cesar: Visual representation of source code vulnerabilities," in *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, Oct. 2016, pp. 1–8. DOI: 10.1109/VIZSEC.2016.7739576.

[8] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," 2002.

[9] D. Freeze. "Global cybersecurity spending to exceed $1.75 trillion from 2021-2025," Cybercrime Magazine. Section: Blogs. (Sep. 10, 2021), [Online]. Available: https://cybersecurityventures.com/cybersecurity-spending-2021-2025/ (visited on 12/29/2023).

[10] "U.s. blames north korea for 'WannaCry' cyber attack | reuters." (), [Online]. Available: https://www.reuters.com/article/idUSKBN1ED00Q/ (visited on 12/28/2023).

[11] ""WannaCry" ransomware attack losses could reach $4 billion - CBS news." (May 16, 2017), [Online]. Available: https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/ (visited on 12/28/2023).

[12] "Allianz | allianz teams up with cyence to boost cyber risk modeling and analysis," Allianz.com. (), [Online]. Available: https://www.allianz.com/en/press/news/business/insurance/170927-allianz-teams-up-with-cyence.html (visited on 01/07/2024).

[13] S. A. Rukmono and M. R. Chaudron, "Enabling analysis and reasoning on software systems through knowledge graph representation," in *2023 webIEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, ISSN: 2574-3864, May 2023, pp. 120–124. DOI: 10.1109/MSR59073.2023.00029.

[14] H. Hanif, M. H. N. Md Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103 009, Apr. 1, 2021, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2021.103009.

[15] Joint Task Force Interagency Working Group, "Security and privacy controls for information systems and organizations," National Institute of Standards and Technology, Sep. 23, 2020, Edition: Revision 5. DOI: 10.6028/NIST.SP.800-53r5.

[16] "Code search results." (), [Online]. Available: https://github.com/search?q=repo%3Atorvalds%2Flinux++language%3AC&type=code (visited on 10/23/2023).

[17] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 international workshop on Mining software repositories*, ser. MSR '05, New York, NY, USA: Association for Computing Machinery, May 17, 2005, pp. 1–5, ISBN: 978-1-59593-123-8. DOI: 10.1145/1083142.1083143.

[18] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, ISSN: 1534-5351, Feb. 2020, pp. 261–271. DOI: 10.1109/SANER48275.2020.9054857.

[19] K. D. Cooper and L. Torczon, "Intermediate representations," in *Engineering a Compiler*, Elsevier, 2023, pp. 159–207, ISBN: 978-0-12-815412-0. DOI: 10.1016/B978-0-12-815412-0.00010-3.

[20] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, Jul. 1, 1970, ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[21] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, "Software vulnerability detection via deep learning over disaggregated code graph representation,"

[22] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys*, vol. 50, no. 4, 56:1–56:36, Aug. 25, 2017, ISSN: 0360-0300. DOI: 10.1145/3092566.

[23] E. Sotos Martínez, N. M. Villanueva, and L. A. Orellana, "A survey on the state of the art of vulnerability assessment techniques," in *14th International Conference on Computational Intelligence in Security for Information Systems and 12th International Conference on European Transnational Educational (CISIS 2021 and ICEUTE 2021)*, J. J. Gude Prego, J. G. de la Puerta, P. García Bringas, H. Quintián, and E. Corchado, Eds., ser. Advances in Intelligent Systems and Computing, Cham: Springer International Publishing, 2022, pp. 203–213, ISBN: 978-3-030-87872-6. DOI: 10.1007/978-3-030-87872-6_20.

[24] B. Shuai, H. Li, L. Zhang, Q. Zhang, and C. Tang, "Software vulnerability detection based on code coverage and test cost," in *2015 11th International Conference on Computational Intelligence and Security (CIS)*, Dec. 2015, pp. 317–321. DOI: 10.1109/CIS.2015.84.

[25] Z. Li, D. Zou, S. Xu, *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. DOI: 10.14722/ndss.2018.23158. arXiv: 1801.01681[cs].

[26] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, Jul. 1, 2022, ISSN: 1545-5971, 1941-0018, 2160-9209. DOI: 10.1109/TDSC.2021.3051525. arXiv: 1807.06756[cs,stat].

[27] Y. Liu and W. Meng, "Acquirer: A hybrid approach to detecting algorithmic complexity vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, New York, NY, USA: Association for Computing Machinery, Nov. 7, 2022, pp. 2071–2084, ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3559337.

[28] S. Woo, E. Choi, H. Lee, and H. Oh, "V1SCAN: Discovering 1-day vulnerabilities in reused C/C++ open-source software components using code classification techniques," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 6541–6556, ISBN: 978-1-939133-37-3.

[29] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, ISSN: 2375-1207, May 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[30] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019.

[31] R. L. Russell, L. Kim, L. H. Hamilton, *et al.*, *Automated vulnerability detection in source code using deep representation learning*, Nov. 27, 2018. DOI: 10.48550/arXiv.1807.04320. arXiv: 1807.04320[cs,stat].

[32] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, "Vulnerability prediction from source code using machine learning," *IEEE Access*, vol. 8, pp. 150 672–150 684, 2020, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3016774.

[33] L. Zhou, M. Huang, Y. Li, Y. Nie, J. Li, and Y. Liu, "GraphEye: A novel solution for detecting vulnerable functions based on graph attention network," in *2021 IEEE Sixth International Conference on Data Science in Cyberspace (DSC)*, Oct. 2021, pp. 381–388. DOI: 10.1109/DSC53577.2021.00060.

[34] Z. Haojie, L. Yujun, L. Yiwei, and Z. Nanxin, "Vulmg: A static detection solution for source code vulnerabilities based on code property graph and graph attention network," in *2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, ISSN: 2576-8964, Dec. 2021, pp. 250–255. DOI: 10.1109/ICCWAMTIP53232.2021.9674145.

[35] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "M VulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, Sep. 2021, Conference Name: IEEE Transactions on Dependable and Secure Computing, ISSN: 1941-0018. DOI: 10.1109/TDSC.2019.2942930.

[36] Y. Ding, S. Suneja, Y. Zheng, *et al.*, *VELVET: A noVel ensemble learning approach to automatically locate VulnErable sTatements*, Jan. 12, 2022. DOI: 10.48550/arXiv.2112.10893. arXiv: 2112.10893[cs].

[37] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA: Association for Computing Machinery, Jul. 5, 2022, pp. 2365–2376, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510229.

[38] N. T. Islam, G. D. L. T. Parra, D. Manuel, E. Bou-Harb, and P. Najafirad, *An unbiased transformer source code learning with semantic vulnerability graph*, Apr. 17, 2023. DOI: 10.48550/arXiv.2304.11072. arXiv: 2304.11072[cs].

[39] Y. Dong, Y. Tang, X. Cheng, and Y. Yang, "DeKeDVer: A deep learning-based multi-type software vulnerability classification framework using vulnerability description and source code," *Information and Software Technology*, vol. 163, p. 107 290, Nov. 2023, ISSN: 09505849. DOI: 10.1016/j.infsof.2023.107290.

[40] Y. Mirsky, G. Macon, M. Brown, *et al.*, "{VulChecker}: Graph-based vulnerability localization in source code," presented at the 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6557–6574, ISBN: 978-1-939133-37-3.

[41] P. Klint, T. Van Der Storm, and J. Vinju, "EASY meta-programming with rascal," in *Generative and Transformational Techniques in Software Engineering III*, J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 6491, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–289, ISBN: 978-3-642-18022-4 978-3-642-18023-1. DOI: 10.1007/978-3-642-18023-1_6.

[42] Y. Lilis and A. Savidis, "A survey of metaprogramming languages," *ACM Computing Surveys*, vol. 52, no. 6, 113:1–113:39, Oct. 16, 2019, ISSN: 0360-0300. DOI: 10.1145/3354584.

[43] M. T. W. Schuts, R. T. A. Aarssen, P. M. Tielemans, and J. J. Vinju, "Large-scale semi-automated migration of legacy c/c++ test code," *Software: Practice and Experience*, vol. 52, no. 7, pp. 1543–1580, 2022, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3082, ISSN: 1097-024X. DOI: 10.1002/spe.3082.

[44] P. Klint, T. van der Storm, and J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, Sep. 2009, pp. 168–177. DOI: 10.1109/SCAM.2009.28.

[45] A. Izmaylova, P. Klint, A. Shahi, and J. Vinju, *M3: An open model for measuring code artifacts*, version: 1, Dec. 4, 2013. arXiv: 1312.1188[cs].

[46] B. Basten, M. Hills, P. Klint, *et al.*, "M3: A general model for code analytics in rascal," in *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, Mar. 2015, pp. 25–28. DOI: 10.1109/SWAN.2015.7070485.

[47] S. Demeyer, S. Tichelaar, and S. Ducasse, *Famix 2.1—the famoos information exchange model*, 2001.

[48] H. Muller and K. Klashinsky, "Rigi: A system for programming-in-the-large," in *Proceedings. [1989] 11th International Conference on Software Engineering*, 1988, pp. 80–86. DOI: 10.1109/ICSE.1988.93690.

[49] R. Holt, A. Winter, and A. Schurr, "Gxl: Toward a standard exchange format," in *Proceedings Seventh Working Conference on Reverse Engineering*, 2000, pp. 162–171. DOI: 10.1109/WCRE.2000.891463.

[50] "About the knowledge discovery metamodel specification version 1.4." (), [Online]. Available: https://www.omg.org/spec/KDM (visited on 12/20/2023).

[51] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier, "Efficient annotated terms," *Software: Practice and Experience*, vol. 30, no. 3, pp. 259–291, 2000. DOI: https://doi.org/10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291097-024X%28200003%2930%3A3%3C259%3A%3AAID-SPE298%3E3.0.CO%3B2-Y.

[52] Rodin Aarssen, J. J. Vinju, Ruichen-Ing, D. Landman, J. Stoel, and O. Duhaiby, *Usethesource/clair: V0.8.0*, version v0.8.0, Aug. 18, 2023. DOI: 10.5281/ZENODO.8261984.

[53] Satrio Adi Rukmono, *Rsatrioadi/javapers: Javapers 1.0*, version v1.0, Jan. 25, 2023. DOI: 10.5281/ZENODO.7568437.

[54] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. DOI: 10.1002/spe.2346.

[55] M. Franz, A. Mondal, O. Sumer, *et al.*, *Cytoscape/cytoscape.js: V3.27.0*, version v3.27.0, Oct. 30, 2023. DOI: 10.5281/ZENODO.10055424.

[56] S. A. Rukmono, L. Ochoa, and M. R. Chaudron, "Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis," in *2023 IEEE International Conference on Data and Software Engineering (ICoDSE)*, 2023, pp. 7–12. DOI: 10.1109/ICoDSE59534.2023.10292037.

Appendix A

A.1 Example of a script for M^3 generation

```
module Main

import IO;
import lang::cpp::M3;
import lang::json::IO;

public list[loc] standardLib=[
|file:///usr/include/c++/8|,
|file:///usr/include/x86_64-linux-gnu/c++/8|,
|file:///usr/include/c++/8/backward|,
|file:///usr/lib/gcc/x86_64-linux-gnu/8/include|,
|file:///usr/local/include|,
|file:///usr/lib/gcc/x86_64-linux-gnu/8/include-fixed|,
|file:///usr/include|];

public list[str] names =[
    "slist_wc","tool_binmode","tool_bname",
    "tool_cb_dbg","tool_cb_hdr","tool_cb_prg",
    "tool_cb_rea","tool_cb_see","tool_cb_wrt",
    "tool_cfgable","tool_dirhie","tool_doswin",
    "tool_easysrc","tool_filetime","tool_findfile",
    "tool_formparse","tool_getparam","tool_getpass",
    "tool_help","tool_helpers","tool_libinfo",
    "tool_listhelp","tool_main","tool_msgs",
    "tool_operate","tool_operhlp","tool_paramhlp",
    "tool_parsecfg","tool_progress","tool_setopt",
    "tool_sleep","tool_stderr","tool_strdup",
    "tool_urlglob","tool_util","tool_vms",
    "tool_writeout","tool_writeout_json",
    "tool_xattr","var"];
int main(){
    map[str,M3] curl_models =(name:tup|name <-names,tup:=
    createM3FromCFile(|file:///home/masuni/curl/src|+(name+".c"),
    stdLib=standardLib,includeStdLib=true,
```

```
    includeDirs=[|file:///home/masuni/curl/src|]));
    compose = composeCppM3(|file:///home/masuni/curl/src|,
    {model|name<-names,model:=curl_models[name]});
    writeJSON(|file:///home/masuni/curl/m3.json|,compose);
    return 0;
}
```

Appendix B

B.1 MATE POI example


```
{
   "poi_result_id": "ef33182092624ba7a0263b8031b30a54",
   "build_id": "6771aac753234ab6841254efb39633c4",
   "analysis_task_id": "20ebe8751acc48e082f4113d53c015ac",
   "analysis_name": "UninitializedStackMemory",
   "poi": {
     "use": "4641",
     "sink": "libGioco.c:103:39:rinforzo",
     "source": "libGioco.c:16:13",
     "insight": "Stack variable `sceltaIn` allocated
     at `libGioco.c:16:13` may be used at
     `libGioco.c:103:39:rinforzo` without having been
        ↪ initialized.",
     "use_contexts": ["nil"],
     "alloc_contexts": ["nil"],
     "local_variable":
        ↪ "<local>:llvm-link:1d25077bf7f1a8e93a4f:sceltaIn"
   },
   "flagged": false,
   "done": false,
   "complexity": "unknown",
   "parent_result_id": null,
   "child_result_ids": [],
   "snapshot_ids": [],
   "graph_requests": [
     {
       "build_id": "6771aac753234ab6841254efb39633c4",
       "source_id": "4243",
       "sink_id": "4641",
       "kind": "forward_control_flow",
```

```
      "avoid_node_ids": [],
      "focus_node_ids": [],
      "request_kind": "slice"
    },
    {
      "build_id": "6771aac753234ab6841254efb39633c4",
      "node_id": "4243",
      "request_kind": "node"
    },
    {
      "build_id": "6771aac753234ab6841254efb39633c4",
      "node_id": "4641",
      "request_kind": "node"
    }
  ],
  "salient_functions": [
    {
      "cpg_id": "<function>:llvm-link:@rinforzo",
      "demangled_name": "rinforzo",
      "name": "rinforzo"
    }
  ]
}
```

Appendix C

C.1 Joern Scan output example

```
Result: 3.0 : Two file operations on the same path can act on
    different files:
libRisika.c:29:gioco

Result: 3.0 : Two file operations on the same path can act on
    different files:
libRisika.c:31:gioco

Result: 3.0 : Two file operations on the same path can act on
    different files:
libRisika.c:80:gioco

Result: 3.0 : Two file operations on the same path can act on
    different files:
libRisika.c:89:gioco

Result: 3.0 : Two file operations on the same path can act on
    different files:
libRisika.c:116:gioco

Result: 3.0 : Unchecked read/recv/malloc: libPrep.c:19:nuovaPartita

Result: 3.0 : Unchecked read/recv/malloc: libRisika.c:54:gioco
```

Appendix D

D.1 Infer output example


```
{
    "bug_type": "RESOURCE_LEAK",
    "qualifier": "resource of type `_IO_FILE` acquired by
      ↪ call to `fopen()` at line 29, column 16 is not
      ↪ released after line 53, column 13.",
    "severity": "ERROR",
    "line": 53,
    "column": 13,
    "procedure": "gioco",
    "procedure_start_line": 12,
    "file": "libRisika.c",
    "bug_trace": [
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 12,
            "column_number": 1,
            "description": "start of procedure gioco()"
        },
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 14,
            "column_number": 5,
            "description": ""
        },
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 17,
```

```
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 24,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 25,
        "column_number": 9,
        "description": "Taking false branch"
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 29,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 31,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 35,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 36,
        "column_number": 5,
```

```
            "description": ""
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 165,
            "column_number": 1,
            "description": "start of procedure
              ↪ importaTerritori()"
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 167,
            "column_number": 5,
            "description": ""
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 169,
            "column_number": 5,
            "description": ""
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 170,
            "column_number": 9,
            "description": "Taking false branch"
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 176,
            "column_number": 16,
            "description": "Loop condition is false. Leaving
              ↪ loop"
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 183,
```

```
        "column_number": 5,
        "description": ""
    },
    {
        "level": 1,
        "filename": "libRisika.c",
        "line_number": 185,
        "column_number": 1,
        "description": "return from a call to
          ↪ importaTerritori"
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 37,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 1,
        "filename": "libRisika.c",
        "line_number": 191,
        "column_number": 1,
        "description": "start of procedure importaCarte()"
    },
    {
        "level": 1,
        "filename": "libRisika.c",
        "line_number": 192,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 1,
        "filename": "libRisika.c",
        "line_number": 196,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 1,
        "filename": "libRisika.c",
        "line_number": 197,
```

```
            "column_number": 9,
            "description": "Taking false branch"
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 202,
            "column_number": 16,
            "description": "Loop condition is false. Leaving
              ↪ loop"
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 217,
            "column_number": 9,
            "description": ""
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 197,
            "column_number": 5,
            "description": ""
        },
        {
            "level": 1,
            "filename": "libRisika.c",
            "line_number": 219,
            "column_number": 1,
            "description": "return from a call to
              ↪ importaCarte"
        },
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 41,
            "column_number": 5,
            "description": ""
        },
        {
            "level": 0,
            "filename": "libRisika.c",
```

```
        "line_number": 42,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 43,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 44,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 46,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 47,
        "column_number": 5,
        "description": ""
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 49,
        "column_number": 9,
        "description": "Taking false branch"
    },
    {
        "level": 0,
        "filename": "libRisika.c",
        "line_number": 49,
```

```
            "column_number": 26,
            "description": "Taking true branch"
        },
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 50,
            "column_number": 13,
            "description": "Taking true branch"
        },
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 51,
            "column_number": 13,
            "description": ""
        },
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 52,
            "column_number": 13,
            "description": ""
        },
        {
            "level": 0,
            "filename": "libRisika.c",
            "line_number": 53,
            "column_number": 13,
            "description": "Skipping importaSalvataggio():
              ↪ empty list of specs"
        }
    ],
    "key": "libRisika.c|gioco|RESOURCE_LEAK",
    "node_key": "e3b34024edc7d12570c55c64cf68cabc",
    "hash": "c1efa4810177987049e841226ecd4efc",
    "bug_type_hum": "Resource Leak"
}
```