Università
Ca'Foscari
Venezia

Master's Degree

in Computer Science

Final Thesis

Mining for vulnerabilities in embedded TCP/IP stacks with a set of static analysis queries

Supervisor
Ch. Prof. Riccardo Focardi

Co-supervisor
Dr. Daniel dos Santos

Graduand
Gabriele Acerbi
877653

Academic Year
2019 / 2020

Table of Contents

A mio nonno Gino,

Abstract

In this thesis, we focus on helping the process of finding vulnerabilities in software. Even though it is a widely addressed topic, insecure code is still one of the main causes of security issues in software because a single bug can potentially mine the security of an entire codebase.

The goal of this thesis is to provide a solution that supports and ease the manual code auditing performed by a researcher. Our implementation will do so by providing a set of codebase-independent static analysis queries that can be quickly run on a target source code to identify code regions, across a whole codebase or across several projects, that may suffer from a particular vulnerability or weakness, therefore allowing to fix them all at once.

We started by going through the available literature in the field as well as the available tools usually employed for this purpose. We then designed and implemented our solution and we finally evaluated it on the source code of seven popular embedded TCP/IP stacks, being able to identify a total of 14 zero-days out of the 46 we found during this research.

Keywords — Vulnerabilities, Static Analysis, Variant Analysis, Joern, CWE, Embedded TCP/IP Stacks

Acknowledgments

This thesis has been fully developed during my internship at Forescout Technologies Inc. [33] in Eindhoven, Netherlands, as part of one of the company's research projects.

For this reason I'd like to mention and thank the whole Forescout research team [34] for this amazing opportunity and for their extremely valuable mentorship and supervision, in particular: Daniel dos Santos, Stanislav Dashevskyi, Amine Amri and Jos Wetzels.

1. Introduction

Insecure coding is one of the main causes of security issues in software because a single bug can potentially mine the whole security of an entire codebase. Even if finding bugs in a program is a classical topic in security research, the available solutions keep falling short because of the unfeasibility of building a “silver bullet” tool able to detect every vulnerability on every codebase as well as because of the significant expertise that this process requires, the complexity of the problem in general, and the huge size of many codebases.

That is why new vulnerabilities are daily discovered by researchers or introduced in a codebase by developers. This is particularly true both for embedded software and for open-source software (OSS). Regarding embedded software the reason is that it is usually designed without security in mind and it’s also harder to test than standard software. For open-source software the reason is that it often includes code and functionalities taken from different projects that may be vulnerable, therefore making the codebase vulnerable even if the originally-developed chunk of it was safe. This means that a single vulnerability can not only mine the whole security of an entire codebase, but it can easily mine the whole security of several different projects because of this silent sharing of potentially vulnerable code between independent codebases.

In this research, we aim at improving this situation by presenting a solution that allows a researcher to:

- Quickly scan an entire codebase and get all the instances of a particular vulnerability or weakness, therefore fixing them all at once.
- Scan any codebase without having to modify the corpus of the analysis, since our solution is completely codebase-independent.
- Quickly scan several codebases to check if a particular vulnerability or weakness has spread through them.
- Speed-up the vulnerability research process by providing hints about regions of the codebase that are more likely to suffer from a particular vulnerability or weakness and that therefore should have a higher priority during the manual analysis.

We conclude this chapter by introducing the definitions of static and variant analysis, by presenting the methodology followed during our research and the target we chose, embedded TCP/IP stacks.

1.1. Introduction to static and variant analysis

Static analysis tools examine the text of a program statically, without attempting to execute it. Theoretically, they can examine either a program's source code or a compiled form of the program to equal benefit, although the problem of decoding the latter can be difficult [27]. Variant analysis is the process of taking a known vulnerability (usually discovered via manual analysis or fuzzing) and finding similar occurrences of that known vulnerability in a codebase. It should be considered that if you write your own software and in particular if it is security- or safety-critical. Since this analysis does not need the code to be executed, it can be considered as a sub-category of static analysis.

Variant analysis is based on the observation that the existence of a given vulnerability in a codebase means that there is a high probability that the same vulnerability may exist in many other places in the codebase or even across multiple projects, since bugs are rarely unique [15]. This can be the consequence of several issues, which include badly tested areas of the codebase, flawed design, confusing APIs, bug duplication due to copy/paste, or the same developer making the same mistakes elsewhere.

The importance of variant analysis relies on the fact that unpatched variants of a known vulnerability can have severe consequences because the code is still vulnerable even if the know vulnerability has been patched and, even worse, while publishing the patch possible attackers can gather hints about where to look to easily exploit the codebase.

Usually variant analysis follows this basic workflow [15]:

1. Fix the known vulnerability.
2. Code review of the function/module where the known vulnerability has been found.
3. Fuzzing, using small variations of the input that triggered the known vulnerability.
4. Look for other code written by the same developer.
5. Search the code for similar patterns.

The techniques used to find similar patterns (that is probably the hardest task of variant analysis so that usually the term variant analysis refers to just this task) have traditionally been manual and highly iterative, making them tedious and time-consuming. However, automated approaches to variant analysis (such as Joern [10] and CodeQL [15]) are now becoming widely available.

These tools look for portions of code that are semantically similar to a given pattern, which gives them a way to automatically find all the variants of a bug in a codebase at once. This is achieved by parsing the source code and

storing some abstraction (implementation-specific) of the codebase and its properties in a database which is then queried for (user-defined or open-source) patterns using implementation-specific languages.

Usually, a developer uses the known vulnerability as a seed to model the pattern that represents all variants of that vulnerability, although starting with a seed is not mandatory if the researcher is able to correctly model the pattern she is looking for. Therefore, variant analysis can be efficiently used to discover zero-day vulnerabilities as well.

A good resource for information about general vulnerable patterns/weaknesses and ideas about how to model a pattern for a specific weakness is the MITRE CWE list [1]. It is a community-developed list of software and hardware weakness types, which aims to serve as a common language, a measuring stick for security tools, and as a baseline for weakness identification, mitigation, and prevention efforts. Even if the CWE list is extremely broad and exhaustive, therefore containing many weaknesses which are not relevant for the scope of our research, we will often reference it during our work to give an authoritative definition of the weaknesses that our solution aims to find.

Some common techniques [25] usually exploited by automated variant analysis tools are:

- Control Flow Analysis, which is useful for finding vulnerable code paths that are only executed under unlikely circumstances.
- Data Flow Analysis, which is useful for finding code paths where potentially unsanitized data is used in a potentially harmful way.
- Taint Analysis, which is useful for tracking attacker-controlled data down the execution paths, taking into account also methods and operations on such data.
- Range Analysis, which is useful for understanding which possible values a variable can hold.
- Semantic Code Search, which is useful for quickly querying a codebase and identifying areas of interest (methods having a particular signature, variables that may contain credentials).

1.2. Research methodology

In this section, we will define the research methodology that will be followed during this project as well as the vulnerability research (VR) infrastructure that we developed for the analysis phase.

Our methodology consists of the following steps (summarized in Figure 2):

1. “Select and prioritize the targets” in which we defined the scope of the project, i.e. selecting which software to analyze and prioritizing them.

2. “Study the literature” in which we tried to gather as much information as possible on the targets like interesting function, entry points and known vulnerabilities as well as gathering information on available candidate tools for the analysis phase.
3. “List and prioritize accessible interfaces” in which we developed a prioritized list of all the interfaces of a particular target. This step was skipped for static analysis since we decided to analyze each target all together.
4. “Analyze each interface” in which we searched for vulnerabilities on the targets. We did that by using our VR infrastructure (see below).
5. “Report the findings” in which we disclose to public our findings following the best practices of coordinated vulnerability disclosure [24].

Figure 2 – Research Methodology Workflow

Regarding our VR infrastructure, our goal was to define an infrastructure combining static analysis through variant analysis, dynamic analysis through fuzzing and manual analysis through manual source code auditing in a way that allowed each analysis technique to both support the other analyses and to validate their findings. The reason we did this was to broaden as much as possible the scope of the bugs that we are able to detect since a specific bug could be only detectable by a specific analysis technique. The tasks performed by each technique, as well as the relationships between both intra-technique tasks and inter-technique tasks are detailed in Figure 3, to ensure a complete understanding of the infrastructure we introduce the following definitions:

- Generic query corpus, set of queries targeting generic weaknesses (CWEs [1]) that are common among almost any software component.
- Precise query corpus, set of queries targeting specific vulnerabilities (CVEs [29]) found in a particular program to check if it’s also present in other codebases.
- Fuzzing [30], a technique to locate implementation flaws by sending malformed or unusual inputs to the target implementation in hopes of producing unexpected behavior.
- Fuzzing success criteria, determines when to stop a fuzzing campaign, often depending on the number of occourred crashes, the time the fuzzer had been running and how much of the codebase had been tested (covered) during the campaign.

- Fuzzing seeds, set of malformed inputs sent to the program under testing and mutated to generate new malformed inputs, usually if a particular input makes the program execute a portion of the codebase that hadn't been previously executed (i.e., improves coverage) then it's added to the set of seeds and used to create new inputs.
- Fuzzing dictionary, set of fuzzers each targeting a different protocol (FTP, DNS, etc.), to be able to exploit protocols specifications a fuzzer must be tailored towords the specific protocol it aims to test.

Furthermore, for the rest of this research we will refer to the zero-days we found as any vulnerability that was found during this project by one of the techniques we included in our infrastructure or by a combination of them.

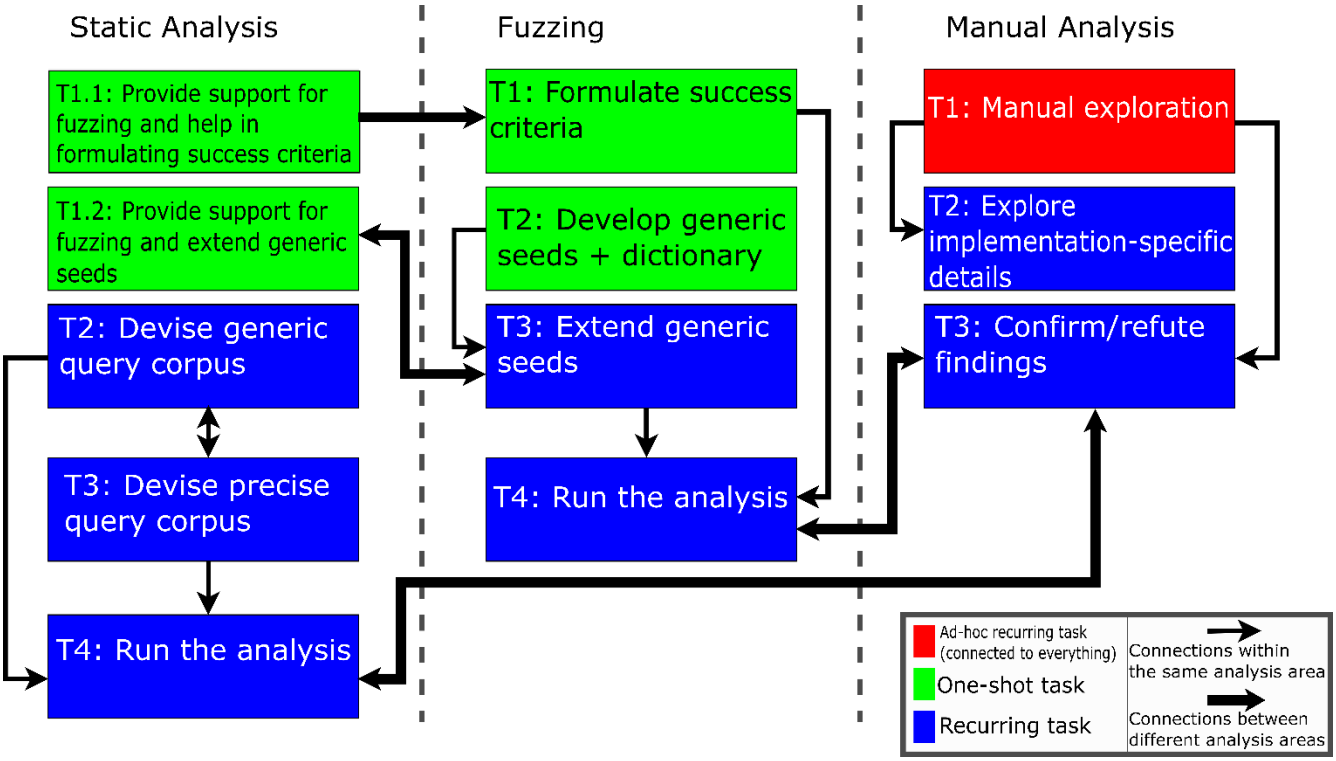

Figure 3 – VR Infrastructure

1.3. Embedded TCP/IP stacks

A TCP/IP stack [31] is designed to build an interconnection of networks that provides universal communication services over heterogeneous physical networks. It's called TCP/IP after two of its most prominent protocols, but there are other protocols as well. The TCP/IP model is based on a five-layer model for networking, these are the physical, data link, network, transport, and application layers. The TCP/IP layers are stacked on top of each other

and contain relatively independent protocols that can be used depending on the needs of the system to provide whatever function is desired. In TCP/IP, each higher layer protocol is supported by lower layer protocols.

An embedded TCP/IP stack [32] is a TCP/IP stack designed (from scratch or by reimplementing an existing "usual" stack) for an embedded system, which is a combination of computer circuitry and software that is built into a product for purposes such as control, monitoring and communication. Whereas embedded systems of the past were usually realized mostly in hardware, nowadays advances in chip technology have made it possible to program complex and pervasive software. From portable wireless devices like sensor nodes to large fixed installations like controlling systems for large chemical plants, embedded software appears everywhere, making it a very interesting research topic. Furthermore, implementing protocol stacks is tedious, error-prone and time-consuming due to the complex and performance-critical nature of network software. The specifications of most modern protocols are quite large. Also, specifications are often not mapped into code in a straightforward manner, which makes it difficult to both achieve and check correctness. In addition, a range of mature optimization techniques designed to make protocol code more efficient tend to make implementations more complicated and are new sources of errors. This is even more so when targeting embedded systems where additional, non-functional constraints come into play: implementations have to minimize code size, energy consumption, memory usage, and other computation resources. In addition, embedded software development often has extended correctness and safety requirements: an embedded system might be controlling a machine that is expected to run continuously without human intervention or to be used in a safety-critical environment. For the interesting factors mentioned above and for the constantly increasing popularity and availability of embedded TCP/IP stacks we selected them as the target of our research.

1.4. Outline of the thesis

The rest of this thesis is organized as follows. Chapter 2 discusses the state of the art and tools selected for development and experimentation. Chapter 3 presents the main contribution of this thesis: a set of queries for the static analysis of embedded TCP/IP stacks. Chapter 4 details the evaluation of these queries using two datasets: one synthetic and one real-world. Finally, Chapter 5 concludes this thesis discussing some limitations and presenting opportunities for future work.

2. Literature review and tool selection

In this chapter, we review the state of the art in static analysis and discuss the tools selected for the development of our queries and their experimentation.

2.1. Classical static analysis tools

We started this research by reviewing the available literature about static analysis to define an initial list of classical static analysis tools that constitute the set of candidate tools for the experiments (presented in chapter 4) that we will run in order to evaluate our solution.

The reason why CodeQL [15] (Section 2.2) and Joern [10] (Section 2.3) are not included in this list but are discussed separately is that they lean towards variant analysis instead of classical static analysis, therefore those are the tools that we compare to select the one for the development of the queries, while the "classical" ones will only be considered to design the experiments for the evaluation.

After an accurate understanding of the available literature we came up with the following list of classical static analysis tools:

- Clang static analyzer [16], which is a source code analysis tool that finds bugs in C, C++, and Objective-C programs. It is intended to be run in tandem with a build of a codebase and it is part of the Clang project.
- Coccinelle [17], which is a program matching and transformation engine that provides the language SmPL for specifying desired matches and transformations in C code. It was initially targeted towards performing collateral evolutions [26] in Linux. Beyond collateral evolutions, Coccinelle is successfully used for finding and fixing bugs in systems code.
- CodeSonar [18], which is used to find and fix bugs and security vulnerabilities in source and binary code, in particular it is often used for embedded software. It uses the concept of abstract interpretation [25] to statically examine all the paths through the application, understand the values of variables and how they impact program state.
- CPAchecker [19], which is a tool for configurable software verification performing a reachability analysis.
- Cppcheck [6], which provides a unique static code analysis on C/C++ projects to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs. Its goal is to have very few false positives. Cppcheck is designed to be able to analyze C/C++ code even if it has non-standard syntax (common in embedded projects).

- Flawfinder [7], which examines C/C++ source code and reports possible security weaknesses sorted by risk level. It works by using a built-in database of C/C++ functions with well-known problems, it then takes the source code text and matches the source code text against those names.
- Fortify [20], which supports more than 30 different programming languages and smoothly integrates in a development environment.
- Frama-C [11], which gathers several static analysis techniques in a single collaborative framework. This approach of Frama-C allows static analyzers to build upon the results already computed by other analyzers in the framework, this makes it also suitable for embedded SW.
- Graudit [8], which is a simple script and signature sets that allows to find potential security flaws in source code using the GNU utility grep. Its goals is being very flexible.
- Klocwork [21], which analyzes code written in C/C++, C#, and Java to identify software security, quality, and reliability issues helping to enforce compliance with standards. It's built for enterprise DevOps and integrates with large complex environments and a wide range of developer tools, it is also suitable for embedded software.
- Polyspace [22], which uses formal methods to prove the absence of critical run-time errors under all possible control flows and data flows. It includes checkers for coding rules, security vulnerabilities, code metrics, and hundreds of additional classes of bugs, it is also suitable for embedded software.
- PVS-studio [12], which is a tool for detecting bugs and security weaknesses in the source code of programs, written in C, C++, C# and Java, It's also suitable for embedded SW. It performs a wide range of code checks, and it is also useful in finding misprints and Copy-Paste errors.
- RATS [9], which scans code written in various languages, including C, C++, Perl, PHP and Python. It is very fast and can easily be integrated into a building process without causing noticeable overhead.
- SonarQube [23], which is a tool meant for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities on 20+ programming languages. It produces reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.
- Splint [13], which is a tool for statically checking C programs for security vulnerabilities and coding mistakes. It supports code instrumentation to improve the quality of the analysis.
- Veracode [14], which provides fast, automated security feedback in the IDE and the pipeline, and conducts a full policy scan before deployment. It then provides clear guidance on what issues to focus on and how to fix them faster.

2.2. CodeQL

CodeQL [15] is a static analysis tool used to automate security checks and perform variant analysis. It was developed by Semmle, which was then acquired by GitHub.

In CodeQL, code is treated like data. Security vulnerabilities, bugs, and other errors are modeled as queries that can be executed against databases extracted from code. You can run the built-in CodeQL queries or write custom ones. CodeQL supports, among others, control and data flow analysis, taint tracking, and threat model exploration.

CodeQL analysis consists of three steps that will detailed later:

- Preparing the code, by creating a CodeQL database.
- Running CodeQL queries against the database
- Interpreting the query results

To create a database, CodeQL first extracts a single relational representation of each source file in the codebase.

For compiled languages, extraction works by monitoring the normal build process. Each time a compiler is invoked to process a source file, a copy of that file is made and all relevant information about the source code is collected. This includes syntactic data about the abstract syntax tree and semantic data about name binding and type information. Hence, for compiled languages, CodeQL needs to invoke the required build system to generate a database, therefore the build method must be available to the CLI.

For interpreted languages, the extractor runs directly on the source code, resolving dependencies to give an accurate representation of the codebase. There is one extractor for each language supported by CodeQL to ensure that the extraction process is as accurate as possible. For multi-language codebases, databases are generated one language at a time.

After it created a CodeQL database, queries are executed against it. CodeQL queries are written in an object-oriented query language called QL that is a combination of both SQL and Java languages. Queries can be run using the CodeQL for VS Code extension or the CodeQL CLI.

The final step converts results produced during query execution into a form that is more meaningful in the context of the source code, how to do this is specified by metadata properties specific to each query.

Following interpretation, results are output for code review and triaging. In CodeQL for Visual Studio Code, interpreted query results are automatically displayed in the source code. Results generated by the CodeQL CLI can be output into a number of different formats for use with different tools.

The tool has some advantages, such as 554 built-in queries provided, running a whole folder (and sub-folders) of queries all at once, and good available documentation. However, it also has some drawbacks, such as: out of the 554 built-in queries only 143 are actually executable, out of the 143 executable ones only 80 are recommended by the developers for actual usage (the filtered 63 are known to yield too many false positives), the execution time seems to be heavily query-dependent (from milliseconds to minutes), the code needs to be in working state (you have to be able to compile it), and the tool is not interactive.

2.3. Joern

Joern [10] is a platform for robust analysis of C/C++ code developed with the goal of providing a useful tool for vulnerability discovery and research in static program analysis. It generates and exploits semantic code property graphs (CPG), a graph representation of code for cross-language code analysis.

Joern allows to work with partial source code and allows to skip the compilation step without major loss of precision of the analysis.

Semantic code property graphs are stored in a custom graph database. This allows code to be mined using search queries formulated in a domain-specific query language based on the graph traversal language Gremlin.

The queries are formulated as steps that Joern takes to traverse the CPG, based on each node's properties, keeping at each step a set of "active" nodes, i.e., the nodes that satisfied all the properties expressed by the query so far. For this reason, the results are presented as the set of nodes in the graph that are still "active" when the query finishes its execution.

Finally, Joern allows to perform code analysis in two ways:

- Interactively, using the interactive shell which serves as an interpreter for the domain-specific code analysis language build on top of Scala and supports flexible manual interaction with the queries allowing to extend and debug them on the fly
- Non-interactively, by coding the queries as Scala scripts and then running them inside the shell.

The tool has some advantages, such as: Fuzzy Parsing of C/C++ since Joern employs a fuzzy parser for C/C++ based on the concept of Island grammars. The parser enables importing arbitrary code even if a working build environment cannot be supplied or parts of the code are missing; Intelligent Search Queries because Joern offers a strongly-typed Scala-based extensible query language for code analysis based on Gremlin-Scala. This

language can be used to manually formulate search queries for vulnerabilities as well as automatically infer them using machine learning techniques; Extendable graphs via CPG passes, Semantic code property graphs are multi-layered, offering information about code on different levels of abstraction. Joern comes with many default passes, but also allows users to add passes to include additional information in the graph and extend the query language accordingly; Fine-grained queries with different trade-offs: Joern allows to compose precise queries for pinpointing "just that one CVE [29]", or less precise but more generic queries that target specific code patterns/bugs (CWEs [1]). Although Joern has some drawbacks as well: Completeness, Joern might not support certain "esoteric" syntax, thus relevant code might be excluded from analysis; Lack of documentation, there is very little documentation, the provided examples are very basic; No built-in queries, Joern provides only the framework, so we had to create our own set of queries from scratch.

2.4. Tool comparison/selection

To compare the tools, we defined the following metrics: free license; support for C/C++; command-line interface (CLI), since our goal is hunting for vulnerabilities, tools that are geared towards a development environment (often coming as simple plugins) are not useful; being ready to use out of the box, since some tools require a user to design and develop the whole analysis on top of the tool it would be unfeasible to do it for each tool because of the time required by this process; requires the code to be built, since, as the targets for our research, we selected embedded TCP/IP stacks, which do not always have a straightforward built process. For example, many of them require both specific hardware and software (configurations, compiler-specific code, ports) in order to be built, a few others can be built only using a particular IDE (usually distributed by the same company that distributes the actual devices).

Below, we reported the KPIs with respect to the classical static analysis tools (Table 16) and with respect to the variant analysis tools (Table 17).

Tool	Free License	Supports C/C++	Provides CLI	Ready out of the box	Requires build
Clang static analyzer [16]	YES	YES	YES	YES	YES
Coccinelle [17]	YES	YES	YES	NO	NO
CodeSonar [18]	NO	YES	NO	YES	YES
Coverity [25]	NO	YES	NO	YES	YES
CPAchecker [19]	YES	YES	YES	NO	NO

Cppcheck [6]	YES	YES	YES	YES	NO
Flawfinder [7]	YES	YES	YES	YES	NO
Fortify [20]	NO	YES	YES	YES	YES
Frama-C [11]	YES	YES	YES	NO	YES
Graudit [8]	YES	YES	YES	YES	NO
Klocwork [21]	NO	YES	YES	YES	YES
Polyspace [22]	NO	YES	NO	YES	YES
PVS-studio [12]	NO	YES	YES	YES	YES
RATS [9]	YES	YES	YES	YES	NO
SonarQube [23]	NO	YES	YES	NO	YES
Splint [13]	YES	YES	YES	YES	YES
Veracode [14]	NO	YES	NO	YES	YES

Table 16 – Metrics on the classical static analysis tools

Tool	Free License	Supports C/C++	Provides CLI	Ready out of the box	Requires build
CodeQL [15]	YES	YES	YES	NO	YES
Joern [10]	YES	YES	YES	NO	NO

Table 17 – Metrics on the variant analysis tools

We conclude this chapter by selecting the tool that will be subsequently used for the development of the queries. Both tools are very similar in terms of performance and both can be effectively used to find new vulnerabilities across codebases (CWE queries) or to find similar vulnerabilities (CVE queries) within the same codebase.

The only difference is the built requirement introduced by CodeQL that, as discussed earlier in this section, can be tricky to achieve on embedded TCP/IP stacks. This is the main reason we decided to pick Joern over CodeQL, another less relevant reason that we considered during Joern selection is the fact that the tool allows you to extend end debug queries very easily and very fast using its interactive mode, this is a very useful feature which isn't present in CodeQL.

3. Vulnerability taxonomy and queries

Since our goal is the discovery of new vulnerabilities in a semi-automated way by accelerating the manual analysis performed by a researcher, it would not have been useful to code queries for specific CVEs or vulnerabilities, so we decided to develop a set of queries looking for more generic patterns and weaknesses that are often a source of vulnerabilities in embedded TCP/IP stacks.

To do so, we started by studying the available literature, in particular previous research on embedded TCP/IP stacks like Ripple20 [2], URGENT11 [3] and Zimperium's vulnerabilities on FreeRTOS+TCP [4]. After integrating it with past research and prior knowledge on the subject within the team, we came up with an original taxonomy of vulnerability classes, specifically tailored for the research field of interest (embedded TCP/IP stacks), the research goal (hunting new vulnerabilities) and the selected tool (Joern).

Our taxonomy, depicted in Figure 1, contains the following main vulnerability classes, each with their own subclasses: Boundary Error, Ill-Defined Expression, Type Confusion, and Uninitialized Use.

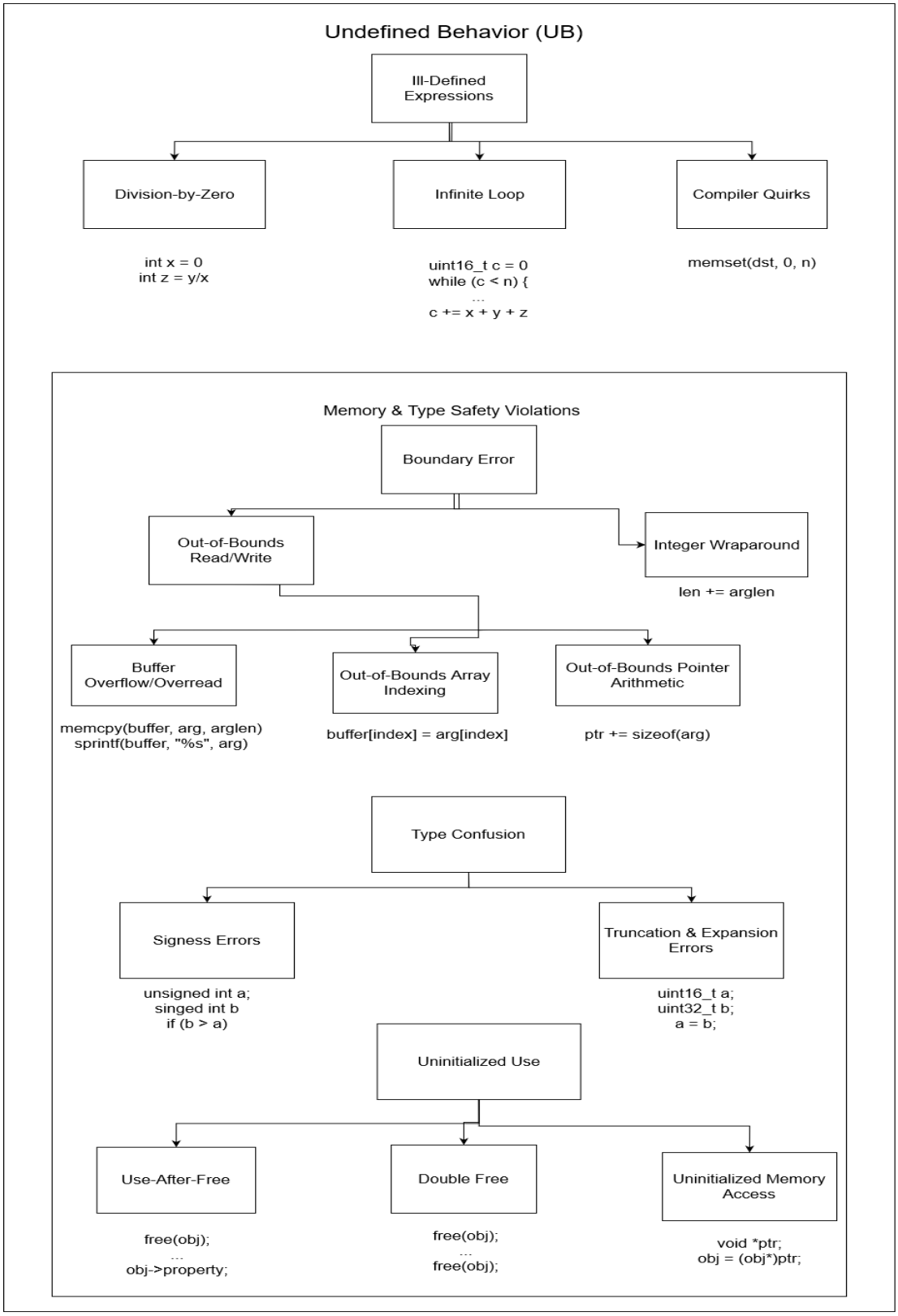

Figure 1 – Taxonomy of vulnerabilities

In the remainder of this Chapter, we detail the classes and subclasses of the taxonomy, as well as the design and development of queries for each class.

Furthermore, to ensure a complete understanding of the queries, we define the following sets that will be often referenced by our queries:

- "dangerous calls" = *scanf, wscanf, sscanf, swscanf, vscanf, vsscanf, strtok, strtok_r, wcstok, strcat, strlcat, strncat, wcscat, wcsncat, strcpy, strlcpy, strncpy, wcscpy, wcsncpy, memcpy, wmemcpy, stpcpy, stpncpy, wcpcpy, wcpncpy, memmove, wmemmove, memcmp, wmemcmp, memset, wmemset, gets, sprintf, slprintf, vsprintf, swprintf, vswprintf, snprintf, vsnprintf, fread, read, malloc.*
- "narrow types" = *bool, char, signed char, unsigned char, u_char, short, short int, signed short, signed short int, unsigned short, unsigned short int, int8_t, uint8_t, s8_t, u8_t, int16_t, uint16_t, s16_t, u16_t.*
- "wide types" = *int, signed int, unsigned int, u_int, wint_t, signed, unsigned, long, long int, signed long, signed long int, unsigned long, unsigned long int, long long, long long int, signed long long, signed long long int, unsigned long long, unsigned long long int, float, double, long double, int32_t, uint32_t, s32_t, u32_t, int64_t, uint64_t, s64_t, u64_t, wchar_t, size_t, ssize_t, ptrdiff_t, intptr_t, uintptr_t.*

3.1. Boundary Errors

This class contains queries looking for weaknesses in which the program is allowed to read and/or write a memory location that isn't within the memory boundaries of the buffer it's operating onto.

3.1.1. Buffer Overflow/Overread

For this subclass, we developed two separate queries: Query_CWE_120_CVE_2018_16601() and Query_CWE_121_type_overrun_mem().

MITRE CWE-120 checks for when the target copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow. The potential impacts of this action are Denial of Service (DoS), memory corruption, and Remote Code Execution (RCE). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. size_t s = readUserInput();
2. memcpy(dest, src, s);
```

The above snippet is vulnerable at line 2 because the program does not validate user input before employing it in a sensitive operation. So the program will copy into dest an arbitrary number of bytes, potentially past the end of dest.

The query to detect such defect in plain English would be: “Give me all the dangerous calls in which the length/size parameter is unchecked user input”. To do so the query simply issues a warning if a variable used as the length/size parameter of a dangerous call is unchecked user input. Below is a pseudocode representation of the query:

```
1. Query_CWE_120_CVE_2018_16601 {
2.     findAllTaintedUserInput()
3.     FOREACH dangerous_call_involving_buffers DO
4.           IF isUserInput(size_parameter) AND NOT
   hasBeenPreviouslyChecked(size_parameter) THEN
5.                 raiseAlert(dangerous_call_involving_buffers)
6.           END
7.     END
8. }
```

MITRE CWE-121 checks for stack-based buffer overflow condition which is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function). The potential impacts of this action are Denial of Service (DoS), memory corruption, Remote Code Execution (RCE), and protection mechanism bypass. [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. typedef struct {
2.     char firstField[16];
3.     int secondField;
4. } myStruct;
5. myStruct mStructure;
6. mStructure.secondField = getData();
7. memcpy(mStructure.firstField, src, sizeof(mStructure));
```

The above snippet is vulnerable at line 7, since the program tries to put data into mStructure.firstField but it uses the size of the whole structure instead of the one of the targeted field so the program will write more data than expected putting them past the end of firstField's boundaries, likely overwriting secondField.

The query to detect such defect in plain English would be: "Give me all the dangerous calls in which the length/size parameter and the destination buffer are of different types". To do so the query takes the following steps:

1. Go over a set of potentially dangerous sinks.
2. Issue a warning if the type of the destination argument (1st, 2nd in some cases) of the sink and the type of the argument passed to a sizeof call in the size/length argument (3rd, 2nd in some cases) of the sink might be different.

Below is a pseudocode representation of the query:

```
1. Query_CWE_121_type_overrun_mem {
2.     FOREACH dangerous_call_involving_buffers DO
3.         IF isCallToSizeof(size_parameter) AND NOT equals(sizeof_argument_type,
   dest_buffer_type) THEN
4.             raiseAlert(dangerous_call_involving_buffers)
5.         END
6.     END
7. }
```

3.1.2. Out-of-Bounds Array Indexing

MITRE CWE-129 checks for when a product uses untrusted input when calculating or using an array index, but the product does not validate or incorrectly validates the index to ensure the index references a valid position within the array. The potential impacts of this action are Denial of Service (DoS), infoleak, memory corruption, and Remote Code Execution (RCE). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. unsigned idx = readUserInput();
2. printf("%s\n", array[idx]);
```

The above snippet is vulnerable at line 2 because the program does not validate user input before employing it as an array index so the program will try to access an arbitrary memory location that could potentially be after the end of the array.

The query to detect such defect in plain English would be: “Give me all the array accesses in which the index parameter is unchecked user input”. To do so the query takes the following steps:

1. Find all variables that are user input and from them integrate all the variables which are subsequently tainted.
2. Find all the array accesses.
3. Check whether an array access contains at least one variable that's unchecked user input inside the expression passed as index.

Below is a pseudocode representation of the query:

```
1. Query_CWE_129 {
2.    findAllTaintedUserInput()
3.    FOREACH array_access DO
4.          IF isUserInput(array_index) AND NOT
   hasBeenPreviouslyChecked(array_index) THEN
5.                raiseAlert(array_access)
6.          END
7.    END
8.  }
```

3.1.3. Out-of-Bounds Pointer Arithmetic

MITRE CWE-823 checks for when a program performs pointer arithmetic on a valid pointer, but it uses an offset that can point outside of the intended range of valid memory locations for the resulting pointer. The potential impacts of this action are Denial of Service (DoS), infoleak, memory corruption, and Remote Code Execution (RCE). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. int packet_length = readUserProvidedLen();
2. printf(“First packet: %s\n”, *packet_buffer_ptr);
3. packet_buffer_ptr = packet_buffer_ptr + packet_length;
```

The above snippet is vulnerable at line 3 because the program does not validate user input before employing it to update a pointer so the program will try to access an arbitrary memory location that could potentially be invalid for that particular pointer.

The query to detect such defect in plain English would be: “Give me all the arithmetic expressions involving both pointers and unchecked user input where the result of the expression is used to update a pointer”. To do so the query takes the following steps:

1. Find all variables that are user input and from them integrate all the variables which are subsequently tainted
2. Find all the possibly dangerous assignments (the ones that have a pointer as left hand and some arithmetic operation(s) right-hand)
3. Check whether a dangerous assignment contains at least one pointer and at least one variable that's unchecked user input, both on the right-hand side

Below is a pseudocode representation of the query:

```
1. Query_CWE_823 {
2.    findAllTaintedUserInput()
3.    targets <- {}
4.    FOREACH assignment DO
5.          IF isPointer(assignment_left_heand) AND
   containsArithmeticOperation(assignment_right_hand) THEN
6.                targets <- targets U {assignment}
7.          END
8.    END
9.    FOREACH target DO
10.               IF containsUserInput(target_right_hand) AND
   containsPointer(target_right_hand) THEN
11.                     raiseAlert(target)
12.               END
13.         END
14.   }
```

3.1.4. Integer Wraparound

MITRE CWE-190 checks for when the software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control. The potential impacts of this action are Denial of Service (DoS), protection mechanism bypass, memory corruption, and Remote Code Execution (RCE). [1]

MITRE CWE-191 checks for when a product subtracts one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result. The potential impacts of this action are Denial of Service (DoS), protection mechanism bypass, memory corruption, and Remote Code Execution (RCE). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. unsigned short s;
2. s = a - b;
3. memcpy(dest, src, s)
```

The above snippet is vulnerable at line 3 because the program performs a sensitive operation based on some data that may suffer from integer wraparound at line 2. If, for example, b > a then the program will try to assign a negative value to s but since it is less than the minimum unsigned short value, it will wraparound and become a big positive number, this will cause the program to possibly write into dest much more bytes then intended.

The query to detect such defect in plain English would be: “Give me all the arithmetic expression where the result isn’t checked and is used to set the value of a narrow type variable that subsequently flows into a dangerous call’s length/size parameter”. To do so the query takes the following steps:

1. Collect all the assignments that have a "narrow" type as left hand some arithmetic operation(s) as right-hand side
2. See if the right-hand side of the assignment passes any checks before the assignment. If not, the target of the assignment might underflow/wrap around
3. Check if the target value that potentially underflows/wraps around flows into a dangerous sink
4. If yes, and the "narrow" type of the target is unsigned, see if the target value is checked just before entering the sink.

5. If the "narrow" type is signed, then even if the check exists, it will not do anything useful because of the underflow/wraparound (e.g., a check "if (x < 0xff)" will be always passed when x is a negative value) therefore, we raise an alert regardless whether the check exists
6. If the "narrow" type is unsigned, the check will likely guard against passing invalid values into the sink. Therefore, we only raise an alert when the check does not exist.

Below is a pseudocode representation of the query:

```
1. Query_CWE_190_191_Integer_Arithmetic {
2.    targets <- {}
3.    FOREACH assignment DO
4.          IF isNarrow(assignment_left_hand) AND
   containsArithmeticOperation(assignment_right_hand) AND NOT
   hasBeenPreviouslyChecked(assignment_right_hand) THEN
5.               targets <- targets U {assignment}
6.          END
7.    END
8.    FOREACH dangerous_call_involving_buffers DO
9.          FOREACH target DO
10.                    IF
   size_parameter.isReachableByDataflowFrom(target_left_hand) AND
   callUsesTargetLeftHand(size_parameter) AND THEN
11.                         IF NOT isUnsigned(target_left_hand) THEN
12.                               raiseAlert(target,
   dangerous_call_involving_buffers)
13.                         ELSE IF isUnsigned(target_left_hand) AND NOT
   hasBeenPreviouslyChecked(target_left_hand) THEN
14.                               raiseAlert(target,
   dangerous_call_involving_buffers)
15.                         END
16.                   END
17.              END
18.         END
19.   }
```

3.2. Ill-Defined Expressions

This class contains queries looking for more general weaknesses which are not strictly related to each other, but they all originate from flawed program logic.

3.2.1. Division-by-Zero

MITRE CWE-369 checks for when a product divides a value by zero. The potential impacts of this action are Denial of Service (DoS). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. int x = readUserInput();
2. int result = y/x;
```

The above snippet is vulnerable at line 2 because the program doesn't validate user input against zero before employing it as a divisor so the program will potentially perform a division by zero.

The query to detect such defect in plain English would be: "Give me all the division operations in which the length/size divisor is unchecked user input". To do so the query takes the following steps:

1. Find all user input divisors in the codebase
2. See if they are checked against zero before their usage

Below is a pseudocode representation of the query:

```
1. Query_CWE_369 {
2.    findAllTaintedUserInput()
3.    FOREACH divisor DO
4.          IF isUserInput(divisor) AND NOT
   hasBeenPreviouslyCheckedAgainstZero(divisor) THEN
5.                raiseAlert(divisor)
6.          END
7.    END
8. }
```

3.2.2. Compiler Quirks

MITRE CWE-14 checks for a specific pattern in which sensitive memory is cleared according to the source code, but compiler optimizations leave the memory untouched when it is not read from again, also known as "dead store removal". The potential impacts of this action are protection mechanism bypass, and infoleak. [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. char passwd[24];
2. passwd = getPassword()
3. memset(passwd, 0, sizeof(passwd))
```

The above snippet is vulnerable at line 3 because the program assumes the password to be erased from memory after its usage but if passwd isn't used anymore after the memset call the compiler will remove the memset call as an optimization but this will leave sensitive information in memory.

The query to detect such defect in plain English would be: "Give me all the memset calls in which the second argument is 0 (i.e., clearing a buffer) and there're no usages of the target buffer (first argument) after the aforementioned call". To do so the query takes the following steps:

1. Find all the calls to '(w)memset()' such that the second argument is zero (e.g., clearing some buffer with zero values).
2. See if there's any usage of the cleared buffer (the first argument of the '(w)memset()' call) after '(w)memset()' was called.

Below is a pseudocode representation of the query:

```
1. Query_CWE_14_memset {
2.     FOREACH memset_call_zero DO
3.           cfg_calls <- getCfgCallchainAfter(memset_call)
4.           IF ∄ usesMemsetClearedBuffer(cfg_calls, memset_call_cleared_buf) THEN
5.                 raiseAlert(memset_call_zero)
6.           END
7.     END
8. }
```

3.2.3. Infinite Loop

MITRE CWE-835 checks if a program contains an iteration or loop with an exit condition that cannot be reached, i.e., an infinite loop. The potential impacts of this action are Denial of Service (DoS). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. for(short i = 0; i >= 0; --i) {
2. i = (i + 1) % 256;
```

The above snippet is vulnerable because i will never be negative so the loop will never reach its exit condition. This is due to the fact that at each iteration line 2 will set i to 1 and subsequently the increment call at line 1 will reset i's value to 0 so i will only be moved back and forth between 0 and 1.

The query to detect such defect in plain English would be: "Give me all the loops in which a variable from the exit condition is of a narrow type and is modified using a complex arithmetic expression inside the loop".

To do so the query takes the following steps:

1. Find all loops' exit conditions.
2. Foreach loop get all the calls inside the loop.
3. If something that is a narrow type and that appears in a loop's exit condition is modified inside the loop in an unusual way ('exit_var = some_call_with_arithmetic' or 'f(exit_var) = some_call_with_arithmetic') then raise an alert.

Below is a pseudocode representation of the query:

```
1. Query_CWE_835 {
2.    FOREACH loop_exit_condition DO
3.          loop_calls <- getCallsInsideLoop(loop_exit_condition)
4.          FOREACH loop_call DO
5.                IF isAssignmentOperation(loop_call) AND
   isLoopConditionVariable(loop_call_left_hand, loop_exit_condition) AND
   isNarrow(loop_call_left_hand) AND
   containsArithmeticOperation(loop_call_right_hand) AND
   containsFunctionCall(loop_call_right_hand) THEN
6.                      raiseAlert(loop_exit_condition, loop_call)
7.                END
8.          END
```

```
9.    END
10.   }
```

3.3. Type Confusion

This class contains queries looking for weaknesses in which the program uses, in a sensitive context, a variable that is the result of a type conversion, since data could be omitted or translated in an unexpected way during the process.

3.3.1. Signedness Errors

MITRE CWE-195 checks for when the software uses a signed primitive and performs a cast to an unsigned primitive, which can produce an unexpected value if the value of the signed primitive cannot be represented using an unsigned primitive. The potential impacts of this action are various, often leading to an Integer/Buffer Overflow. [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. // scenario 1
2. int i = readData();
3. memcpy(dest, src, i);
4. // scenario 2
5. int i = readData();
6. unsigned u = i;
7. memcpy(dst, src, u)
```

According to the first scenario, the above snippet is vulnerable at line 3 because the program uses a signed primitive in a sensitive operation which expects size_t (unsigned int) without previously checking it against zero, so if i's value is negative it will become a huge positive number when converted to size_t, this will cause the program to potentially write much more bytes then expected into dest.

According to the second scenario, the above snippet is vulnerable at line 7 because the program uses a primitive which is unsigned but that derives from a signed one in a sensitive operation without previously checking it against zero, so if i's value is negative and since u is unsigned the assignment at line 6 it set u to a huge positive number, this will cause the program to potentially write much more bytes then expected into dest.

The query to detect such defect in plain English would be: “(First scenario) Give me all the dangerous calls in which the length/size parameter is an unchecked signed primitive. (Second scenario) Give me all the assignments in which a signed variable is assigned to an unsigned one that subsequently flows into a dangerous call’s length/size parameter”. To do so the query takes the following steps:

- First scenario:
  1. Check if the length/size argument of a dangerous call is of a signed type and hasn't been previously checked against zero
  2. Raise an alert

- Second scenario:
  1. Go over all the assignments and keep only the ones where the left-hand side is of an unsigned type while the right-hand side is of a signed type and hasn't been checked against zero before the assignment
  2. Check if at least one variable from the assignment's left-hand side is unchecked and used in the length/size parameter of a dangerous call and if there's dataflow from the assignment's left-hand side to the dangerous call
  3. Raise an alert

Below is a pseudocode representation of the query:

```
1. Query_CWE_195 {
2. // scenario 1
3.    FOREACH dangerous_call_involving_buffers DO
4.          IF NOT isUnsigned(size_parameter) AND NOT
   hasBeenPreviouslyCheckedAgainstZero(size_parameter) THEN
5.                raiseAlert(dangerous_call_involving_buffers)
6.          END
7.    END
8. // scenario 2
9.    targets <- {}
10.         FOREACH assignment DO
```

```
11.               IF isUnsigned(assignment_left_hand) AND NOT
      isUnsigned(assignment_right_hand) AND
   hasBeenPreviouslyCheckedAgainstZero(assignment_right_hand) THEN
12.                     targets <- targets U {assignment}
13.               END
14.         END
15.         FOREACH target DO
16.               FOREACH dangerous_call_involving_buffers DO
17.                     IF
   size_parameter.isReachableByDataflowFrom(target_left_hand) AND
   callUsesTargetLeftHand(size_parameter) AND NOT
   hasBeenPreviouslyCheckedAgainstZero(target_left_hand) THEN
18.                           raiseAlert(target, dangerous_call_involving_buffers)
19.                     END
20.               END
21.         END
22.   }
```

3.3.2. Truncation & Expansion Errors

MITRE CWE-192checks for a set of flaws pertaining to the type casting, extension, or truncation of primitive data types. The potential impacts of this action are Denial of Service (DoS), and Remote Code Execution (RCE). [1]

MITRE CWE-197 checks for when a primitive is cast to a primitive of a smaller size and data is lost in the conversion due to a truncation error. The potential impacts of this action are memory corruption. [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. uint32_t a = readData();
2. uint16_t b = a;
3. memcpy(dest, src, b);
```

The above snippet is vulnerable at line 3 because the program uses a variable derived from a potential truncation (line 2, since a is 32 bits and b is 16 bits then only 16 bits from a will be copied into b, this could result in b ending up with an unexpected value) in a sensitive operation without previously performing any check on the value's validity, this will cause the program to potentially write not the expected amount of bytes into dest.

The query to detect such defect in plain English would be: “Give me all the assignments in which the left and the right hand side are of different width and the left hand side subsequently flows into a dangerous call’s length/size parameter without being previously checked”. To do so the query takes the following steps:

1. From all the assignments keep only the ones like: 'wide_type = narrow_type', 'narrow_type = wide_type', 'var = (narrow_cast)wide_type', 'var = (wide_cast)narrow_type'.
2. Discard the ones in which at least a variable from the assignment's right-hand side has been checked before the assignment
3. Check if at least one variable from the assignment's left-hand side is unchecked and used in the length/size parameter of a dangerous call and if there's dataflow from the assignment's left-hand side to the dangerous call.
4. Raise an alert.

Below is a pseudocode representation of the query:

```
1. Query_CWE_192_197 {
2.    targets <- {}
3.    FOREACH assignment DO
4.          IF (NOT hasBeenPreviouslyChecked(assignment_right_hand)) AND
   (oneTypeIsNarrowOneIsWide(assignment_left_hand, assignment_right_hand) OR
   (isCastCall(assignment_right_hand) AND oneTypeIsNarrowOneIsWide(cast_type,
   casted_variable))) THEN
5.               targets <- targets U {assignment}
6.          END
7.    END
8.    FOREACH target DO
9.          FOREACH dangerous_call_involving_buffers DO
10.                   IF
   size_parameter.isReachableByDataflowFrom(target_left_hand) AND
   callUsesTargetLeftHand(size_parameter) AND NOT
   hasBeenPreviouslyChecked(target_left_hand) THEN
11.                        raiseAlert(target, dangerous_call_involving_buffers)
12.                   END
13.              END
14.         END
```

```
15.    }
```

3.4. Uninitialized Use

This class contains queries looking for weaknesses in which the program accesses or uses either a resource that hasn't been initialized or a memory location that's no longer valid, even if it was valid in the past.

3.4.1. Use-After-Free

MITRE CWE-416 checks for references of a memory location after it has been freed. The potential impacts of this action are Denial of Service (DoS), memory corruption, and Remote Code Execution (RCE). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. free(ptr);
2. printf("%s\n", *ptr);
```

The above snippet is vulnerable at line 2 because the program dereferences an arbitrary memory location since we have no idea what's in memory at address ptr after free(ptr) got executed at line 1.

The query to detect such defect in plain English would be: "Give me all the pairs <previous free call, subsequent call using pointer> which operate on the same pointer and where no reallocation of that exact pointer appears in the Control Flow Graph modeling the program steps to go from the free call to the call using the pointer". To do so the query takes the following steps:

1. Find all the pointers allocated through a malloc-like function.
2. Find all free-like calls which are reachable by dataflow from a pointer found in the previous step
3. Traverse the CFG down starting from each free-like call and store all the reached calls.
4. Check if a freed pointer is passed as an argument to a call (not a free-like call) along the traversal (discard the pointers that have been re-allocated after being freed but before being used).

Below is a pseudocode representation of the query:

```
1. Query_CWE_416 {
2.     targets_free <- {}
3.     FOREACH malloc_allocated_var DO
4.           FOREACH free_call DO
```

```
5.                IF free_call.isReachableByDataflowFrom(malloc_allocated_var)
   THEN
6.                      targets_free <- targets_free U {free_call}
7.                END
8.          END
9.    END
10.         FOREACH target_free DO
11.               cfg_calls <- getCfgCallchainAfter(target)
12.               FOREACH cfg_call DO
13.                     IF callReallocatesFreadTarget(cfg_call) THEN
14.                           goToNextTarget()
15.                     ELSE IF callUsesFreadTarget(cfg_call) AND NOT
   isFreeLikeCall(cfg_call) THEN
16.                           raiseAlert(target_free, cfg_call)
17.                     END
18.               END
19.         END
20.   }
```

3.4.2. Double Free

MITRE CWE-415 checks for when a product calls free() twice on the same memory address, potentially leading to modification of unexpected memory locations. The potential impacts of this action are memory corruption, and Remote Code Execution (RCE). [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. free(ptr);
2. free(ptr);
```

The above snippet is vulnerable at line 2 because the program frees arbitrary memory content since we have no idea what's in memory at address ptr after free(ptr) got executed at line 1.

The query to detect such defect in plain English would be: "Give me all the pairs of free calls which operate on the same pointer and where no reallocation of that exact pointer appears in the Control Flow Graph modeling the program steps to go from the first free call to the second one". To do so the query takes the following steps:

1. Find all the pointers allocated through a malloc-like function.
2. Find all free-like calls which are reachable by dataflow from a pointer found in the previous step.
3. Traverse the CFG down starting from each free-like call and store all the reached calls.
4. Check if a freed pointer is passed as an argument to a free-like call along the traversal (discard the pointers that have been re-allocated after being freed the first time but before being freed again).

Below is a pseudocode representation of the query:

```
1. Query_CWE_415 {
2.    targets_free <- {}
3.    FOREACH malloc_allocated_var DO
4.          FOREACH free_call DO
5.                IF free_call.isReachableByDataflowFrom(malloc_allocated_var)
   THEN
6.                      targets_free <- targets_free U {free_call}
7.                END
8.          END
9.    END
10.         FOREACH target_free DO
11.               cfg_calls <- getCfgCallchainAfter(target)
12.               FOREACH cfg_call DO
13.                     IF callReallocatesFreadTarget(cfg_call) THEN
14.                           goToNextTarget()
15.                     ELSE IF callUsesFreadTarget(cfg_call) AND
   isFreeLikeCall(cfg_call) THEN
16.                           raiseAlert(target_free, cfg_call)
17.                     END
18.               END
19.         END
20.   }
```

3.4.3. Uninitialized Memory Access

MITRE CWE-457 checks for a pattern in which the code uses a variable that has not been initialized, leading to unpredictable or unintended results. The potential impacts of this action are various. [1]

Below is a snippet of vulnerable code that falls in this subclass:

```
1. int ptr = (int*)malloc(sizeof(int));
2. printf("%s\n", *ptr);
```

The above snippet is vulnerable at line 2 because the program dereferences a pointer whose result is completely unpredictable since we have no idea what's in memory at address ptr before it's initialization (probably garbage).

The query to detect such defect in plain English would be: "Give me all the calls using a pointer for which no initialization can be found in the Control Flow Graph modeling the program steps to reach that call from the beginning of the program". To do so the query takes the following steps:

1. Find all pointers and remove the ones coming as function parameters and the ones enclosed in assignments like 'ptr = ...'.
2. Traverse the CFG up starting from each expression enclosing a pointer and store all the reached calls, then reverse the calls queue.
3. Check if the pointer was initialized (approximated as '... ptr = ...').
4. If a pointer is used without being previously initialized, then raise an alert.

Below is a pseudocode representation of the query:

```
1. Query_CWE_457 {
2.     pointers <- {}
3.     FOREACH variable DO
4.           IF isPointer(variable) AND NOT isFunctionParameter(variable) AND NOT
   isInAssignment(variable) THEN
5.                 pointers <- pointers U {variable}
6.           END
7.     END
8.     FOREACH pointer DO
9.           cfg_calls <- getCfgCallchainAfter(pointer) U {pointer}
10.                reverseOrder(cfg_calls)
11.                FOREACH cfg_call DO
12.                      IF isAssignment(cfg_call) AND equals(pointer,
   cfg_call_left_hand) THEN
13.                            goToNextPointer()
14.                      ELSE IF callUsesTargetPointer(cfg_call) THEN
```

```
15.                         raiseAlert(pointer, cfg_call)
16.                     END
17.                 END
18.             END
19.     }
```

4. Evaluation

In this chapter, we proceed to evaluate the performance of the queries developed in chapter 3 by comparing them against the results of different static analysis tools. More in detail, we demonstrate how both our queries and such tools perform when trying to find vulnerabilities in a specific and non-standard kind of target (embedded TCP/IP stacks) as well as how easy and useful they are from the point of view of a security analyst processing the analysis results. We will conclude this chapter discussing some usability issues that are common to several static analysis tools and that influenced our experimental design.

4.1. Experimental design

We started by selecting the target datasets against which our queries and the other tools would be tested to get their performances at finding vulnerabilities. We decided to use a synthetic dataset and a real-world one, both written in the same programming language (which is C in this case).

As a synthetic dataset, we selected the Juliet Test Suite [5] because it is explicitly meant for the evaluation of static analysis tools and because it contains tens of thousands of testcases, each with a vulnerable and a fixed version. From all the available test classes, we selected only the suitable ones for our experiment. The factors we considered while picking the test classes were: the fact that we developed a query for that particular weakness, the fact that both our query and the testcase shared the same logic (in some queries we introduced the constraint of flowing into a dangerous sink to “validate” the weakness), the fact that our query was not based on the notion of user input, since we tailored our way to detect user input towards the way embedded TCP/IP stacks get user input.

As a real-world dataset, we selected the 7 embedded TCP/IP stacks that we analyzed during our research, using the zero-days we found as the set of true positives. Such stacks will be referenced as Stack #[1-7], to avoid disclosing confidential information.

Regarding the tools that will be evaluated during the experiment, we started from the list of tools that we presented in Chapter 2. From that, we crossed out all the unsuitable ones, mainly because of not being ready to use out of the box and because of requiring the code to be built. Unfortunately, we were not able to get a free academic license for any commercial tool, so our experiment will only evaluate free tools, even though being a free tool was not a requirement when we started designing this experiment.

The final tools we selected for the experiment are: Cppcheck, which is particularly interesting since it is designed to analyze C/C++ code even if it has non-standard syntax (common in embedded projects) [6], Flawfinder [7],

Graudit [8] and RATS [9]. Our choice also took into consideration the fact that these tools are not fully automated but require a subsequent manual analysis, as Joern.

Furthermore, since it is a better indication of the overall performance of a semi-automated tool aiming to speed-up a subsequent manual analysis, we decided to track hits that are close to a vulnerability rather than hits flagging the exact vulnerability. Therefore, for the remaining of this experiment we will use the following definitions of TP, FP and FP:

- A hit is a True Positive if it flags as vulnerable a function that is actually vulnerable, regardless of the actual vulnerable statement reported by the tool.
- A hit is a False Positive if it flags as vulnerable a function that is not actually vulnerable.
- A hit is a False Negative when there are no hits regarding a vulnerable function.

We finally defined a score function to be able to rank the tools according to their performances. We begin by introducing the definitions of Precision, which is the fraction of hits which are relevant for a query [28], and Recall, which is the fraction of relevant hits that are returned [28].

$$PRECISION = \left(\frac{TRUE\ POSITIVES}{NUM\ HITS}\right)$$

$$RECALL = \left(\frac{TRUE\ POSITIVES}{NUM\ VULNS}\right)$$

We included the number of hits in the calculation as a standalone parameter since it is a very important factor for this kind of semi-automated tools, because each hit must subsequently be triaged by a researcher. So if, for example, two tools have similar performances (precision and recall) but the difference between the number of hits they produced is huge, then we want to rank higher the tool with fewer hits since it would be a much better tool from the point of view of the researcher doing the subsequent manual analysis. Each score is hence computed as follows:

$$SCORE = \left(\frac{PRECISION + RECALL}{NUM\ HITS}\right) * 10000$$

4.2. Quantitative Analysis

In this section, we will present our results as well as some takeaways from the comparison between Joern and the other static analysis tools we selected.

4.2.1. Synthetic dataset: JULIET dataset

For this experiment, we first selected a subset of all the Juliet test classes, keeping only the suitable ones. We then run each tool on each selected test class and extracted the results, as well as computing both the precision and the recall metrics for each tool. Such results are presented in Tables 1-4. We concluded the experiment by creating an aggregate table (Table 5) that summarizes the overall results on the Juliet Test Suite, from that we ranked the tools (Table 6) according to our score function and we finished by analyzing the results.

CWE-121 (4968 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	76	4896	**4**	72	**0,94**	0,01
Cppcheck	4558	4960	4550	8	0,001	0,001
Flawfinder	32703	**0**	27735	**4968**	0,15	**1**
Graudit	1296	3929	257	1039	0,81	0,2
RATS	11219	2003	8254	2965	0,26	0,59

Table 1 – Results on the testcase CWE121_Stack_Based_Buffer_Overflow

Table 1 presents the results we achieved while looking for one of the vulnerabilities which are most often targeted by static analysis tools: buffer overflows. Looking at the results we can notice that Joern [10] got a very high precision, hence a very small number of FPs, but it also produced a very high number of FNs, hence a low recall, this can be due both to the fact that we could run only one of the two queries we developed for buffer overflows (Query_CWE_120_CVE_2018_16601() is based on the notion of user input) and to the fact that while developing the queries we gave priority to limiting the number of FPs to make sure that we were actually speeding-up the subsequent manual analysis instead of risking of slowing it down by having a huge number of hits to be individually triaged. An example of the opposite design choice is Flawfinder [7] (low precision but maximum recall as we can evince from Table 1) which, as other tools, tends to flag as vulnerable any call which may be vulnerable under some circumstances (any call to a function which may suffer from buffer overflow in this case).

CWE-195 (1152 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	3408	**0**	**1934**	**1152**	**0,33**	**1**

Cppcheck	2393	1152	2393	0	0	0
Flawfinder	8208	1152	82080	0	0	0
Graudit	0	1152	0	0	0	0
RATS	0	1152	0	0	0	0

Table 2 – Results on the testcase CWE195_Signed_to_Unsigned_Conversion_Error

Table 2 presents the results we achieved when looking for dangerous signed to unsigned conversions. To detect this kind of weakness a tool has to be aware of the notion of type, hence tracking types through the codebase, and to be able to determine the actual type of a variable (even if it's different from the one in the initial variable declaration), we implemented both in our queries. As we can see in Table 2 Joern [10] performed very good on this task, being able to identify all the vulnerabilities while keeping precision at a reasonable value. On the other hand, it's clear how all the other tools failed on the task, each producing zero TPs.

CWE-415 (962 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	168	**42**	836	126	**0,75**	0,13
Cppcheck	1156	682	876	**280**	0,24	**0,29**
Flawfinder	1102	962	1102	0	0	0
Graudit	10	962	**10**	0	0	0
RATS	0	962	0	0	0	0

Table 3 – Results on the testcase CWE415_Double_Free

CWE-416 (459 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	114	**357**	12	**102**	**0,89**	**0,22**
Cppcheck	0	459	**0**	0	0	0
Flawfinder	508	459	508	0	0	0
Graudit	6	459	6	0	0	0
RATS	0	459	**0**	0	0	0

Table 4 – Results on the testcase CWE416_Use_After_Free

Table 3 and Table 4 present the results we got when looking for uninitialized use vulnerabilities, this kind of vulnerabilities needs the notion of control flow to be implemented in a tool to be detected, we implemented

that in our queries as well. As we can see Joern [10] performed quite good on both while Flawfinder [7], Graudit [8] and RATS [9] weren't able to find any TP, this could be due to the fact that these tools aren't aware of various control flows but they simply look at a potentially vulnerable statement locally, finally Cppcheck [6] is aware of the notion of control flows but it only employ it when looking for double frees, although it performed worse than Joern [10].

Total (7541 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	3766	5295	2786	1452	0,38	0,19
Cppcheck	8107	7253	7819	288	0,03	0,03
Flawfinder	42521	**2573**	37553	**4968**	0,11	**0,65**
Graudit	1312	6502	**237**	1039	**0,79**	0,17
RATS	11219	4576	8254	2965	0,29	0,39

Table 5 – Final results on the Juliet Test Suite

Final Ranking	**Position**	**Score**
Graudit	1	7,31
Joern	2	1,51
RATS	3	0,60
Flawfinder	4	0,17
Cppcheck	5	0,07

Table 6 – Final ranks on the Juliet Test Suite

Finally, Table 5 and Table 6 present the overall conclusions of the experiment. As we can see in the ranking, Joern [10] ended up second with only Graudit [8] performing better. This is due to the latter performing very well on buffer overflows, while introducing very few FPs on the other testcases (even though the only vulnerabilities that Graudit [8] found are related to the first testcase).

We can then conclude that even though our queries were explicitly developed to be used against embedded TCP/IP stacks, they performed reasonably well also on other kinds of codebases, getting a higher score than three out of the four tools we employed in this experiment.

4.2.2. Real dataset: Embedded TCP/IP stacks

For this experiment, we started by running each tool on each embedded TCP/IP stack that we analyzed during this research and extracting the results (we used the set of found zero-days as the set of our TPs), as well as computing both the precision and the recall metrics for each tool; such results are presented in Tables 7-13. We concluded the experiment by creating an aggregate table (Table 14) that summarizes the overall results on the embedded TCP/IP stacks, from that we ranked the tools (Table 15) according to our score function and we finished by analyzing the results.

Tables 7-13 present the results on each individual stack, as we can see there are big differences in the performances of each tool depending on which stack it is run against. This can be a consequence of several factors that range from particular coding patterns or syntax used by the developers to the robustness of a particular stack, or simply being due to the fact that a notable chunk of the TPs on a particular stack were found by dynamic analysis and that they are undetectable without running the code. Furthermore, we presented the results on each individual stack separately for the sake of completeness, but we won't dive deeper into them.

Stack #1 (10 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	327	10	327	0	0	0
Cppcheck	97	10	**97**	0	0	0
Flawfinder	1769	**5**	1764	**5**	0,002	**0,50**
Graudit	202	7	199	3	**0,01**	0,30
RATS	193	9	192	1	0,005	0,10

Table 7 – Results on Stack #1

Stack #2 (3 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	91	3	91	0	0	0
Cppcheck	21	3	**21**	0	0	0
Flawfinder	846	**1**	844	**2**	0,002	**0,66**
Graudit	108	2	107	1	0,009	0,33
RATS	83	2	82	1	**0,01**	0,33

Table 8 – Results on Stack #2

Stack #3 (5 vulns)	# Hits	# False Negatives	# False Positives	# True Positives	Precision	Recall
Joern	31	**3**	29	**2**	**0,06**	**0,40**
Cppcheck	0	5	**0**	0	0	0
Flawfinder	32	5	32	0	0	0
Graudit	64	4	63	1	0,01	0,20
RATS	7	5	7	0	0	0

Table 9 – Results on Stack #3

Stack #4 (2 vulns)	# Hits	# False Negatives	# False Positives	# True Positives	Precision	Recall
Joern	191	**0**	189	**2**	**0,01**	**1**
Cppcheck	32	2	**32**	0	0	0
Flawfinder	693	**0**	691	**2**	0,002	**1**
Graudit	39	2	39	0	0	0
RATS	200	1	199	1	0,005	0,50

Table 10 – Results on Stack #4

Stack #5 (9 vulns)	# Hits	# False Negatives	# False Positives	# True Positives	Precision	Recall
Joern	53	**5**	49	**4**	**0,07**	**0,44**
Cppcheck	34	9	**34**	0	0	0
Flawfinder	364	6	361	3	0,008	0,33
Graudit	55	9	55	0	0	0
RATS	61	8	60	1	0,01	0,11

Table 11 – Results on Stack #5

Stack #6 (9 vulns)	# Hits	# False Negatives	# False Positives	# True Positives	Precision	Recall
Joern	62	7	60	2	**0,03**	0,22
Cppcheck	43	9	**43**	0	0	0
Flawfinder	386	**6**	383	**3**	0,007	**0,33**

Graudit	59	9	59	0	0	0
RATS	61	8	60	1	0,01	0,11

Table 12 – Results on Stack #6

Stack #7 (8 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	29	**4**	25	**4**	0,15	**0,50**
Cppcheck	28	8	28	0	0	0
Flawfinder	253	5	250	3	0,01	0,37
Graudit	9	6	7	2	**0,22**	0,25
RATS	6	8	**6**	0	0	0

Table 13 – Results on Stack #7

Total (46 vulns)	**# Hits**	**# False Negatives**	**# False Positives**	**# True Positives**	**Precision**	**Recall**
Joern	784	32	770	14	**0,01**	0,30
Cppcheck	255	46	**255**	0	0	0
Flawfinder	4343	**28**	4325	**18**	0,004	**0,39**
Graudit	536	49	529	7	**0,01**	0,15
RATS	611	41	606	5	0,008	0,10

Table 14 – Final results on the embedded TCP/IP stacks

Final Ranking	**Position**	**Score**
Joern	1	3,95
Graudit	2	2,98
RATS	3	1,63
Flawfinder	4	0,80
Cppcheck	5	0

Table 15 – Final ranks on the embedded TCP/IP stacks

Finally, Table 14 and Table 15 present the overall conclusions of the experiment. As we can evince from the results Joern [10] performed very well, being able to spot about 30% of the vulnerabilities, while keeping the total number of hits low and hence manageable; it then ended up first in our ranking.

Two interesting results came from Cppcheck [6] and Flawfinder [7]. The former is explicitly designed for non-standard syntax codebases as embedded TCP/IP stacks although it wasn't able to spot any vulnerability during the experiment, making it the worst performing tool against the embedded TCP/IP stacks dataset.

The interesting thing about the latter is about seeing how its "flag everything" detection logic (as we discussed in Section 4.2.1) performs when applied to huge and non-standard codebases as the ones under examination. As we can see from the results, it does not perform well against this kind of targets because it only increased the number of TPs by 9% with respect to Joern [10] but in the meanwhile it increased the number of hits by about 550%, again with respect to Joern [10], making it much less useful and manageable from the point of view of a researcher subsequently triaging each hit.

We can then conclude that the way we tailored our queries towards vulnerability research on embedded TCP/IP stacks was worth it since Joern [10] performed very well during this experiment with respect to both the number of spotted TPs and the fact of being able to keep the total number of hits quite manageable. This ended up with Joern [10] being the top performer among the tools we tested during this experiment and therefore getting the first place in our final rank (Table 15).

4.3. Usability discussion

For these experiments, we were not able to include all the tools that we listed in Chapter 2, but we had to select a subset of suitable ones. In this section, we detail some usability issues that influenced our tools selection process.

- IDE plugins (e.g., Veracode [14]), since we aren't developing code but we are hunting for vulnerabilities we discarded all the tools coming as plugins for an IDE because they wouldn't have been of any use for our purposes.
- Tools for continuous development and automated code auditing (e.g., Veracode [14]), since the goal of our research is finding vulnerabilities we selected only the tools whose goal matches our, discarding all the ones that are leaning towards a development/production environment rather than a research one.

- Commercial licensed tools (e.g., PVS-studio [12]), even if this wasn't an issue when we first started designing the experiments it became one later on since, unfortunately, we weren't able to get a free academic license for any commercial tool.
- Tools that aren't ready to use out of the box (e.g., Frama-C [11]), we decided to discard all those tools that require a user to design and develop the whole analysis on top of the tool since it'd have been unfeasible to model it for each tool included in the experiment because of the time required by this process. Since it requires a user to design and develop the whole analysis (the queries) on top of it, we can take as an example Joern [10], for which it took about five months to design, develop, tune and assure the quality of our set of queries.
- Tools that require the code to be built (e.g., Clang Static Analyzer [16]), we decided to discard these tools because of the targets we selected for our research, embedded TCP/IP stacks, which don't always have a straightforward built process. For example many of them require both specific HW (an actual device) and SW (configurations, compiler-specific code, ports) in order to be built, a few others can be built only using a particular IDE (usually distributed by the same company that distributes the actual devices). Furthermore, Joern [10] doesn't compile the code but simply parses it so by discarding the tools exploiting information gathered from the compiler we selected a subset of tools which are more closely related to each other, hence more easily comparable.

5. Conclusion

Being able to spot vulnerabilities in a software component is a key process when assuring the robustness of a product. In this thesis, we presented a solution to improve and ease the process of finding bugs in source code. Our approach exploits the concepts of variant analysis, integrates them with our expertise and tailors them towards the target of our research, embedded TCP/IP stacks. Each query went through a rigorous refinement process to assure its quality. We demonstrated the effectiveness of our solution by testing it against a synthetic dataset and then comparing its results with the ones produced by other tools as well as by testing it against seven popular embedded TCP/IP stacks and being able to spot 14 zero-days.

We conclude this paper by pointing out some limitations of our approach as well as some possible future improvements.

5.1. Limitations

Even if we got very nice results on the embedded TCP/IP stacks, our solution is still subject to some limitations that we want to discuss:

- Because of our approach being based on static analysis it will not overcome the limitations of this technique, therefore vulnerabilities that can only be spotted through executing the code won't be found
- Because of the problem of finding every vulnerability in a codebase being unfeasible only a subset of all the ones affecting a codebase will be found
- Because we only implemented the queries that we included in our taxonomy, any vulnerability that doesn't fall within one of those classes won't be detected
- Because while developing the queries we always kept in mind the effort-results ratio for some of them we preferred a limited query over an exhaustive one, therefore some of our queries won't cover any possible corner case

5.2. Future work

Possible further improvements could be:

- Extending our taxonomy of vulnerability classes, and therefore our corpus of queries, to be able to detect a broader range of weaknesses
- Improving the quality of individual queries as well as integrating all the uncovered corner cases

- Developing a front-end for our solution to wrap up and make available all its capabilities through a web interface

References

[1] - MITRE CWEs - https://cwe.mitre.org/

[2] - JSOF Ripple20 - https://www.jsof-tech.com/ripple20/

[3] - Armis URGENT11 - https://www.armis.com/urgent11/

[4] - Zimperium FreeRTOS+TCP - https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/

[5] - Juliet Test Suite - https://samate.nist.gov/SARD/testsuite.php

[6] - Cppcheck - http://cppcheck.sourceforge.net/

[7] - Flawfinder - https://dwheeler.com/flawfinder/

[8] - Graudit - https://github.com/wireghoul/graudit

[9] - RATS - https://security.web.cern.ch/recommendations/en/codetools/rats.shtml

[10] - Joern - https://joern.io/

[11] - Frama-C - https://frama-c.com/

[12] - PVS-studio - https://www.viva64.com/en/pvs-studio/

[13] - Splint - http://splint.org/

[14] - Veracode - https://www.veracode.com/products/binary-static-analysis-sast

[15] - CodeQL - https://securitylab.github.com/tools/codeql

[16] - Clang static analyzer - http://clang-analyzer.llvm.org/

[17] - Coccinelle - http://coccinelle.lip6.fr/

[18] - CodeSonar - https://www.grammatech.com/codesonar-cc

[19] - CPAchecker - https://cpachecker.sosy-lab.org/

[20] - Fortify - https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[21] - Klocwork - https://www.perforce.com/products/klocwork

[22] - Polyspace - https://nl.mathworks.com/products/polyspace.html

[23] - SonarQube - https://www.sonarqube.org/

[24] - Forescout Disclosure Policy - https://www.forescout.com/company/resources/forescout-vulnerability-disclosure-policy/

[25] - Static Program Analysis - https://cs.au.dk/~amoeller/spa/spa.pdf

[26] - Collateral Evolution - https://coccinelle.gitlabpages.inria.fr/website/papers/RR-5769.pdf

[27] - Static Analysis for Security - https://ieeexplore.ieee.org/abstract/document/1366126

[28] - Precision and Recall - https://www.cs.cornell.edu/courses/cs578/2003fa/performance_measures.pdf

[29] - MITRE CVEs - https://cve.mitre.org/

[30] - Fuzzing - https://cs.uwaterloo.ca/~sgorbuno/publications/autofuzz.pdf

[31] - TCP/IP Stacks - http://www.exa.unicen.edu.ar/catedras/comdat1/material/TP1-Ejercicio5-ingles.pdf

[32] - Embedded TCP/IP Stacks - http://www.diva-portal.org/smash/get/diva2:235613/FULLTEXT01.pdf

[33] - Forescout Technologies Inc. - https://www.forescout.com/

[34] - Forescout Research Labs - https://www.forescout.com/forescout-research-labs/