
Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica
(Computer Engineering)
Academic year: 2021/2022
Graduation session: October 2022

# Evaluation of Static Security Analysis Tools on Open Source Distributed Applications

Supervisor
Prof. Riccardo Sisto

Candidate
Vincenzo Di Stasio

# Contents

# Abstract

The use of static security analysis tools is becoming common practice in distributed application development in terms of discovering as many security vulnerabilities as possible. To compare static analysis tools for web applications, a benchmark adapted to the vulnerability categories included in the known standard Open Web Application Security Project (OWASP) Top Ten project is required. The aim of the thesis is to evaluate some static security analysis tools by applying them to a significant set of distributed open-source applications. However, distinct tools provide different results depending on factors such as the complexity of the code under analysis and the application scenario, thus missing some of the vulnerabilities while reporting false problems. While some benchmarks already exist for evaluating these tools, they are not well aligned with the latest web development techniques. The work consists in identifying some relevant and modern open source projects to use as benchmarks. Then, some of the static security analysis tools were tested on the selected projects and the results on their performance were collected, following the evaluation methodology suggested by OWASP. The results of this work have been obtained using widely acceptable metrics to classify them.

# Chapter 1

# Introduction

Web applications are remarkably capable of being instantaneously accessible to millions of people and deployable in a short amount of time. The software business is paying more attention to the factors about their protection as a result of their ever-increasing popularity and the high-value data they disclose. The need for new web applications with sophisticated features is rapidly rising because they benefit most business sectors in terms of competition. This can result in a lot of flaws and security issues because their development is frequently done on a tight deadline. When utilised, their impact can have serious negative effects on enterprises, such as financial losses, liability issues, brand harm, and market share loss. Every single day, more flaws are found. Software development practices must be altered to improve software security. This is not a simple job. Neither the system administrator nor the end user should be responsible for software security. Although network security, prudent management, and prudent use are all crucial, these efforts will ultimately fail if the software is intrinsically insecure. By incorporating security into the software development process, one can achieve good software security with the correct information and resources. It takes some extra thought, focus, and work to maintain security. Programmers who haven't yet encountered security issues use their good fortune as justification to keep ignoring security, even though this additional labour wasn't nearly as crucial in earlier decades. A hole in the computer system that exposes data to uninvited parties is known as a vulnerability in computer security. Three things can be used to define it: the existence of a weakness in the system, the attacker's capacity to find the flaw, and the attacker's capacity to use the flaw. There could be a defect for several reasons: the absence or improper application of best practices when coding (such as input/output validation); frequently, security testing is overlooked in favour of functional requirements; occasionally, attack detection mechanisms are not included in the environment due to performance overheads and potential false positives that could disrupt normal behaviour.

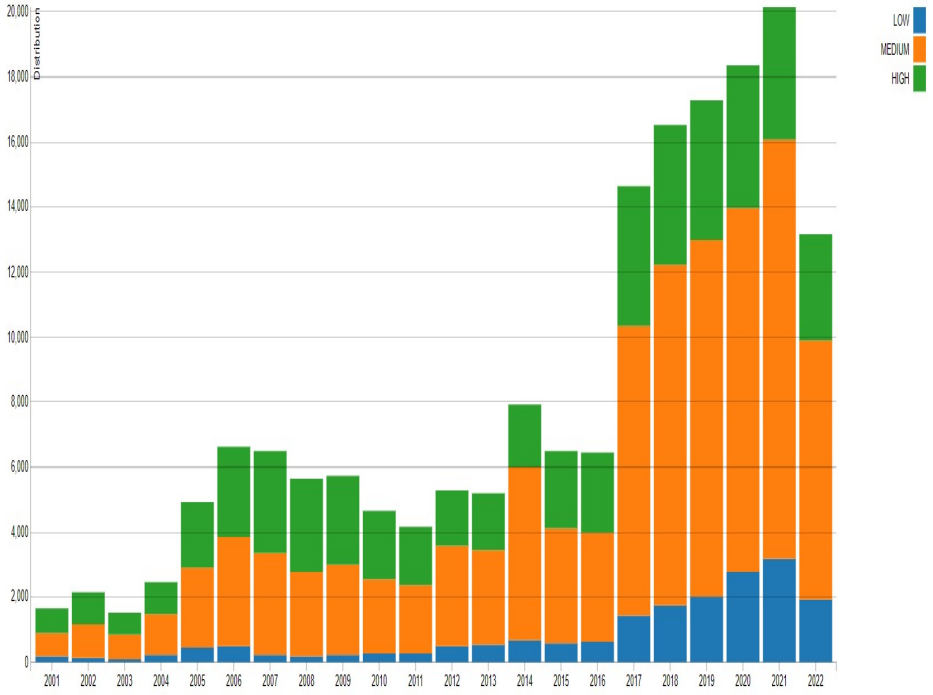


Figure 1.1: distribution of vulnerabilities by severity over time.

Figure 1.1 makes it abundantly evident that the number of vulnerabilities has been rising over time. In fact, since 2016, when there were roughly 6608 vulnerabilities altogether, the number has more than tripled, reaching a total of 20137 vulnerabilities in the year 2021. All the data used for Figure 1.1 have been retrieved from the National Vulnerability Database (NVD)[21]. The National Vulnerability Database (NVD) is a repository for vulnerability management data that follows standards and is represented using the Security Content Automation Protocol (SCAP). Automation of compliance, security measurement, and vulnerability management are made possible by this data. The NVD has databases of product names, impact metrics, security-related software defects, configuration errors, and references to security checklists. The Common Vulnerability Scoring System (CVSS), which is based on a set of equations using metrics such as access complexity and availability of a remedy, is used by the NVD in addition to giving a list of Common Vulnerabilities and Exposures (CVEs). A dictionary or glossary of vulnerabilities that have been found for particular code bases, such as software applications or open libraries, can be found in the Common Vulnerabilities and Exposures (CVE) program. By using a special identification called the CVE ID, this list enables interested parties to

obtain the specifics of vulnerabilities. Since it has been more well-known in recent years, participants and users must comprehend the key components of the program. Because it allows for the speedy assessment of numerous options, static analysis is effective. Without having to perform all the calculations required to execute the code for each situation, a static analysis tool can examine a huge number of "what if" scenarios. Because many security issues arise in hard-to-reach states and corner situations that can be challenging to test by actually running the code, static analysis is especially well suited for security. A quick way to obtain a consistent and thorough evaluation of a body of code is to use good static analysis tools. One of the most crucial tasks in the early phases of the software development lifecycle is static analysis, which helps find flaws. Although static analysis can examine all of the code, it has drawbacks including triggering false alarms and overlooking some application issues since how successful it is depends on things like the complexity of the code, the programming structures used, and the presence of third-party components. Organizations should perform a security analysis utilising the finest SAST techniques as a result of the security flaws that web apps have in their code. Manually checking for security flaws in a web application with a large number of lines of code can be laborious and time-consuming. The most recent automatic techniques, including SAST tools for source and binary code, must be studied. To verify each security vulnerability, a final audit of a SAST tool report is necessary. A false negative is more difficult to locate, though, if the tool hasn't already identified it as posing a genuine threat. False positives can usually be corrected by the security analyst, therefore they present minimal threat. Every security flaw in the source code that has been found must be fixed. In light of this, we believe that users and professionals of web applications should be aware of which commercial and open-source SAST tools are more effective in terms of real detection rates (true positives), unreal detections (false positives), and vulnerabilities that have not yet been discovered (false negatives). As a result, several tools frequently produce wildly disparate findings, making it difficult to choose the SAST that is most appropriate for a given project. By contrasting their behaviour while evaluating pertinent applications, benchmarking could help in the selection of alternative SAST. The most well-known SAST benchmark now in use is the OWASP Benchmark. These benchmarks are unable to be adjusted to a particular context (such as critical or non-critical applications), which may alter the usefulness of the results in addition to not producing results that are true to reality. This thesis suggests using workloads made up of real applications with known vulnerabilities to build benchmarks for the evaluation of SAST that detect vulnerabilities in web applications (used to exercise the SAST, thus supporting their evaluation). In contrast to processing considerably simpler synthetic code samples or test cases, this ensures that SAST is tested taking into account the requirement to address both the complexity

and the method real code is generated (as done by OWASP). SAST perform better with simulated test cases than with actual software.

**Goal**
The objective of this project is to create a benchmark application to evaluate SAST tools' performance. SAST tools have a difficult time exploring some of the code in this benchmark program, which may be hiding some vulnerabilities. We require many criteria to rank the SAST since no single indicator is enough to assess all aspects of SAST performance across all situations. The proposed strategy is based on applying various criteria in that order to rank the tools and, if necessary, resolve ties between two or more products. We take into account a sample selection of exposed applications to create the workload. 16 SAST in total were evaluated using the benchmark. Because Javascript and Python are the two languages most frequently used in the creation of online applications, these technologies are taken into consideration in this analysis. The findings demonstrate that it may be used to rank the SAST and that various tools have diverse vulnerability detection capabilities, with some doing very poorly in particular circumstances. Furthermore, we discovered that no tool is the best option in every situation, which supports the need to modify the workload and ranking metrics in which the tools would be utilised. We demonstrate the benefit of taking into account certain metrics and specialised workloads by comparing our results with the OWASP benchmark.

**Overview**
The contents of the thesis are organized into six chapters, including the current introductory one.

In Chapter 2, we begin with an overview of the basic concepts needed to better understand this thesis. In particular, security in web-based technologies is presented. Subsequently, the benchmark is described in general, highlighting its various main characteristics. We then moved on to the generic description of the SAST tools in which their strengths and weaknesses were identified. Finally, we explain the static analysis and why we chose it.

In Chapter 3, we describe the OWASP assessment methodology as it has been used within this thesis. First, the benchmark used by OWASP is described, highlighting the basic characteristics. Furthermore, the operation of the tools that carry out the static analysis is described in general, highlighting the strengths and weaknesses. Finally, an explanation is given on the use of static analysis and why we chose it.

In Chapter 4, we describe the objectives chosen as samples for the experiment and also the reason behind the selection of these objectives. Then we dealt with the introduction of the tools used for this thesis. There is a list of the tools and hence they are presented one by one in detail.

In chapter 5, we analyzed the results obtained from the experiment. In the beginning, we dealt with describing the methodology used to analyze a web application through a tool. Next, it is explained how the experiment was conducted and under what conditions. Then, we show the results of the experiment in a table which is followed by some considerations and interpretations of the results. In particular, a lot of attention is given to the results when the tools were launched on our benchmark program. Finally, with the analysis of some specific metrics, we have drawn up a ranking to compare the different static analysis tools.

Chapter 6 summarizes the most relevant results achieved in the thesis, highlighting the advantages and weaknesses associated with the results found. We also propose some extensions to this work.

# Chapter 2

# Background

The background on web technology security, background on benchmarking procedures and SAST tools are presented in this chapter.

## 2.1 Web Technologies Security

Web applications are becoming more and more developed, which makes them a valuable target for attack for those looking to profit from exposing potential security flaws in organisations and businesses connected via the Internet. Organizations must be aware that investing in web application security must be planned from the start, usually speaking, starting in the early stages of application development, given this ongoing threat. The most popular languages now are PHP, Javascript, and Python, and Android and iOS are becoming more and more well-known. Even today, Java is the most common language. There is a discussion of web apps' vulnerabilities: Asynchronous Javascript and XML (AJAX), HTML5, and flash apps are the subjects of Web 2.0 attacks. I use the term "secure" to describe programming languages that carry out runtime checks automatically to stop programs from exceeding the allotted memory limits. I must look into which security flaws are more prevalent and harmful in web apps to construct suitable benchmarks to evaluate and choose the best-performance SAST tools. The online applications that were examined failed the OWASP Top Ten project because they contained more frequent and dangerous vulnerabilities. While web applications share traits with traditional software, they also stand out due to the dispersed nature of the Internet, the use and reuse of third-party components created in many languages, web user interfaces, the speed of data access, and transaction security.

## 2.2 Benchmarking

Utilizing a set of sample test cases to run various tools and comparing the results is the most popular way to evaluate and compare their performance. A benchmark is a regular procedure for carrying out this activity and normally has three primary parts:

1. Workload, a group of sample test cases for the tools being benchmarked.
2. Metrics to evaluate how effectively the benchmarking tools perform their intended purpose.
3. Guidelines and procedures for running the benchmark.

The component that is most impacted by the benchmarking domain is the workload, which has a significant impact on the outcomes. Consequently, the workload should guarantee the following attributes:

- **Representativeness**: The workload ought to reflect that of the area where the benchmark will be used. The quantity and variety of the test cases have an impact on this. The consumers should receive pertinent information from the benchmark results in the context of its intended use.
- **Comprehensiveness**: All of the significant features that are often employed in the target domain should be able to be exercised by the workload. Features should be balanced based on how they are used in the real situation.
- **Focus**: The workload should be focused on describing the benchmarking targets. The following three factors should be taken into account: ground truth, relevance, and coverage.
- **Configurability**: Users ought to be able to alter the workload according to their requirements.
- **Scalability**: The workload should change depending on the quantity and level of test cases according to the actual application.

## 2.3 SAST Tools

The best way to prevent flaws in a web application's code is through prevention. To prevent making errors linked to programming vulnerabilities, developers ought to have specialized training in web security programming. There will always be flaws in the code, even if security programmers receive excellent training, and it will be much harder to evaluate the source code

once the first version of the application or certain portions of it have been built. Manual static analysis needs highly skilled personnel and a lot of time. It takes a lot of time and highly specialised employees to manually analyse the behaviour of the built application in an effort to ethically hack it. It is also quite challenging to cover the full application attack surface. Any technique of performing a web application security analysis requires that you cover the complete attack surface, including all components and application layers. You should also use tools to automate security analysis as much as possible, mixing different types of technologies to get the best results. White box analysis, carried out using SAST tools, is one type of security analysis that examines both source code and executables as necessary. The act of determining whether or not a programme reaches its final state is a problem for SAST tools. Without actually running the programme, SAST examine the source code to look for potential issues. Many people believe that using these tools is the most effective way to find software vulnerabilities automatically. But SAST have their limitations and frequently have flaws in them. A few specialised programming constructs, such as dynamic file inclusion, dynamic string evaluation, object-oriented programming (OOP), and automated typecasts, are also difficult to analyse. As a result, SAST developers tend to produce approximations of their solutions, which frequently result in false alarms and hidden flaws. There are many SAST accessible now, and new tools are developing to meet emerging demands. However, depending on the technologies and algorithms utilised in their development, these tools have varying strengths. There is no agreement on which SAST is the best because to the wide range of findings produced by SAST, particularly the trade-off between soundness and completeness, since false alarms take a long time to verify and undetected weaknesses can be exploited. The detection of vulnerabilities can be enhanced by combining different SAST. Combining numerous tools can actually work against you because it will result in more false positives reported rather to more vulnerabilities being found. A final audit of a SAST tool report is required to eliminate the false positives and find the false negatives (much more complicated). Security analysts need to adequate the training to reconnaissance the security vulnerabilities in the code for a particular programming language. SAST tools interfaces can be more or less "friendly" in terms of the error trace facilities to audit a security vulnerability. The fact that SAST tools examine the entire programme taking into account all source inputs is one of their most significant advantages.

## 2.4 Static Analysis

Static analysis tools are discussed in this section along with their benefits, drawbacks, and definitions. Static analysis is the process of examining code

without running it. The static analysis tools that work much like a spell checker to stop well-known types of errors from going undiscovered are the ones that we are most interested in using to identify security issues. Even people who are proficient spellers need a spell checker since spelling errors usually occur. A spell checker won't catch every typo; for example, it won't assist if you put mute when you meant moot. The same applies to tools for static analysis. A clean run doesn't guarantee that your code is perfect; rather, it just shows that it is free of some types of typical issues. A spell checker is a beneficial tool for the majority of experienced and professional writers. A spell checker is useful for both good and bad writers, but it won't make them better writers. The static analysis follows the same rules: Static analysis tools can be effectively used by good programmers, while bad programmers will still write bad code no matter what techniques they employ. Although the focus is on tools for static analysis that can spot security flaws, we start by considering the variety of issues that static analysis can assist in resolving. Later in the chapter, we examine the underlying issues that, from both a theoretical and a practical perspective, make static analysis challenging, and we look at the compromises that tools must make to achieve their goals.

### 2.4.1 Capabilities and limitations of static analysis

The same kind of small errors that cause even a strong speller to occasionally produce a typo can cause security issues: a tiny bit of perplexity, a momentary lapse, or a temporary disconnect between the brain and the keyboard. However, security issues can also arise from a lack of knowledge about what safe programming implies. Programmers frequently have no idea how an attacker would try to exploit a piece of code, which is not unusual. In light of this, there are a variety of reasons why static analysis is effective at locating security issues:

- Without any of the prejudice that a programmer could have regarding which portions of code are "interesting" from a security standpoint or which pieces of code are simple to exercise through dynamic testing, static analysis tools conduct checks completely and consistently.

- Static analysis tools may frequently identify the main source of a security issue, not simply one of its symptoms, by looking at the code itself. This is especially crucial to ensure that vulnerabilities are appropriately repaired.

- Early in the development process, even before the program is run for the first time, the static analysis might discover flaws. Early error detection not only reduces the cost of resolving the issue, but the short feedback loop can also direct a programmer's work: A programmer has

the chance to fix errors they weren't even aware could occur. A static analysis tool's attack scenarios and data on code constructions serve as a means of knowledge transfer.

- Static analysis tools make it simple to examine a large body of code when a security researcher finds a new type of attack to determine where it might succeed. The ability to analyse legacy code for newly identified types of faults is crucial because some security flaws in software can go undetected for years.

Static analysis technologies that focus on security are sometimes criticised for making too much noise. Particularly, they generate an excessive number of false positives, also known as false alarms. A false positive in this situation is when a program reports a problem when none truly exists. A significant number of false positives can lead to significant problems. A programmer who needs to look through a big list of false positives may miss critical results that are hidden in the list. False positives are terrible, but false negatives are even worse in terms of security. A false negative occurs when the tool fails to detect an issue that is present in the program. The amount of time lost examining the result is the fine for a false positive. The cost of a false negative is much higher. In addition to paying the cost of having a vulnerability in your code, you also suffer from a distorted feeling of security because the tool gave the impression that everything was fine. There will always be some false positives or false negatives produced by static analysis tools. The majority of the tools create both. The ratio that a tool achieves between false positives and false negatives is frequently a good indicator of the tool's intended use. Static analysis tools that are designed to find common faults and static analysis techniques that focus on security-relevant flaws have quite different ideal balances. The cost of missing a common bug is quite low because the most significant bugs may be caught using a variety of methods and procedures. Because of this, code quality tools often aim to produce fewer false positives and are more tolerant of false negatives. Security is another matter. Security technologies typically produce more false positives to decrease false negatives since the cost of failing to catch security issues is considerable. A flaw needs to be evident in the code for a static analysis tool to find it. Even though it might seem obvious, it's crucial to realise that architectural risk analysis is a vital addition to static analysis.

### 2.4.2 Static Analysis Internals

What motivates static analysis tools is covered in this section. We examine the inner workings of sophisticated static analysis tools, including data structures, methods of analysis, guidelines, and strategies for reporting outcomes. The goal is to provide enough information about the components of

static analysis tools so that you can get the most out of the ones you use. All static analysis tools that focus on security work in a manner that is largely consistent regardless of the analysis methods employed. They all start with accepting code, creating a model of the program that represents it, evaluating that model using a corpus of security knowledge, and then finishing by showing the user their findings. This section outlines the procedure and examines each step in further detail.

### Building a Model

A static analysis tool must first convert the source code into a program model a collection of data structures to examine it. As might be expected, the type of analysis a tool does is directly related to the model it builds, but generally speaking, compilers are a big influence on static analysis tools. In reality, academics who studied compilers and compiler optimization issues were responsible for the development of numerous static analysis approaches. Now let's take a quick look at the key methods and data structures that compilers and static analysis tools have in common.

#### Lexical Analysis

Tools that work with source code start by breaking the code down into tokens, removing any unnecessary text elements like whitespace and comments along the way. Lexical analysis is the process of generating the token stream. Regular expressions are frequently used in lexical rules to identify tokens. Observe that while the majority of tokens are solely represented by their token type, the ID token also needs the name of the identifier to function. Tokens should include at least one extra type of information to facilitate later relevant error reporting: their position in the source text (usually a line number and a column number). At this stage, the task is almost complete for the most basic static analysis tools. The analyzer can search through the token stream for identifiers, compare them to a list of names of risky functions, and return the results if all the tool is going to do is match the names of dangerous functions.

#### Parsing

To match the token stream, a language parser employs context-free grammar (CFG). A group of productions that describe the symbols (or elements) in the language make up the grammar. The token stream is compared to the production rules by the parser, which then conducts a derivation. A parse tree is created if each symbol is linked to the symbol it was derived from.

#### Abstract Syntax

Because a parse tree is the most accurate representation of the code as written by the programmer, it may be used for considerable analysis and

is the best tool for several stylistic checks. However, there are a lot of reasons why executing extensive analysis on a parse tree can be difficult. It is generally preferable to abstract away both the specifics of the grammar and the syntactic sugar present in the program text. The nodes in the tree are directly derived from the grammar's production rules, and those rules can introduce nonterminal symbols that exist solely to make parsing simple and unambiguous, rather than to produce an easily understood tree. An abstract syntax tree (AST) is a type of data structure that performs these functions. The AST's goal is to deliver a standardised program version appropriate for analysis in the future. Typically, the production rules of the grammar are associated with tree construction code to create the AST. The AST may have fewer constructs than the source language, depending on the requirements of the system. For instance, for and do loops could be changed to while loops and method calls could be changed to function calls. This kind of significant software simplicity is referred to as lowering. Although it runs the danger of misrepresenting the programmer's intent, closely comparable languages can be reduced into the same AST structure. Syntactically related languages may share many of the same AST node types, but they almost certainly will have unique types of nodes for aspects that are unique to that language.

**Semantic Analysis**

The tool simultaneously creates a symbol table and an AST. The symbol table links each identifier in the program to its type and a pointer to its declaration or definition. Now that it has access to the AST and the symbol table, the tool may do type-checking. Although type information is crucial for the analysis of an object-oriented language, static analysis tools are not needed to disclose type-checking problems in the same way that compilers do. This is because the type of an object specifies the set of methods that the object can execute. Additionally, it is typically preferred to at least translate implicit type conversions from the source language into explicit type conversions in the AST. Because of these factors, a sophisticated static analysis tool must perform the same amount of type-checking work as a compiler. Because the compiler assigns meaning to the symbols discovered in the program, *semantic* analysis is the term used to describe symbol resolution and type checking in the context of compilers. These data structures give static analysis tools a clear edge over those that do not. Compilers and more sophisticated static analysis techniques diverge after semantic analysis. A modern compiler creates an *intermediate representation* which is a generic form of machine code that may be optimised and then converted into platform-specific object code using the AST, symbol, and type information. Static analysis tools have a more complicated future. A static analysis tool may perform further modifications on the AST or create its unique varia-

tion of intermediate representation depending on the sort of analysis being performed. It is typical for static analysis tools to at least support assignment, branching, looping, and function calls when using their intermediate representation. A static analysis tool's intermediate representation typically provides a higher-level view of the program than a compiler's intermediate representation.

**Tracking Control Flow**
Numerous static analysis algorithms (and compiler optimization methods) investigate the various possible directions in which a function can be executed. The majority of tools construct a control flow graph on top of the AST or intermediate representation to make these algorithms efficient. Basic building blocks, or nodes, are sequences of instructions that will always be carried out sequentially, beginning with the first instruction and ending with the last instruction, without the possibility of any instructions being skipped. The control flow graph's edges are directed and signify probable control flow routes between fundamental building elements. Potential loops are represented by the back edges in a control flow graph. The sequence of fundamental blocks that a program performs can be used to characterise its control flow when it is in use. A *trace* is a group of fundamental building components that chart a course through the code. Potential control flow between functions or procedures is represented by a *call graph*. Building a call graph without function pointers or virtual methods is as easy as examining the function identifiers referred to in each function. The graph's nodes stand in for functions, and its directed edges reflect the likelihood that one function will call another. When function pointers or virtual methods are called, the tool can limit the list of potential functions that can be called from a call site by combining dataflow analysis (described below) with data type analysis. The control flow graph cannot be guaranteed to be complete if the program loads code modules dynamically during runtime because the program may execute code that is hidden at the time of analysis. A static analysis tool will ideally piece together a control flow graph that depicts the links between the elements for software systems that span several programming languages or are composed of numerous cooperating processes. The information required to bridge the call graphs in various contexts is stored in the configuration files for particular systems.

**Tracking Dataflow**
Algorithms for dataflow analysis look at how data flow within a program. Dataflow analysis is a process used by compilers to allocate registers, eliminate dead code, and carry out a variety of additional optimizations. Examining the control flow graph of a function and identifying where data values are created and consumed are typical steps in dataflow analysis. For many

dataflow issues, converting a function to *Static Single Assignment (SSA)* form is helpful. Only one value can be given to a variable by a function in SSA form. The software needs new variables added to comply with this constraint. Given any program variable, it is simple to determine the source of the variable's value using the SSA form. There are numerous uses for this quality. For instance, if a constant value is ever set to an SSA variable, all instances of the SSA variable may be replaced by the constant. The term *constant propagation* refers to this method. Finding security issues like hard-coded passwords or encryption keys can be done with constant propagation alone. In SSA form, a variable must be reconciled at the point where the control flow paths converge if it is given distinct values along multiple control flow paths. By establishing a new version of the variable and giving the new version the value from one of the two control flow channels, SSA achieves this merge. A "-function" is the notational abbreviation for this merge point. Depending on the control flow path that is followed, the -function acts as a stand-in for the selection of the suitable value.

**Taint Propagation**
Which program values could be under the possible control of an attacker must be known by security tools. Taint propagation refers to the use of dataflow to discover what an attacker can control. Knowing the program's entry points and how data flows through it is necessary. Many input validation and representation flaws can be found through taint propagation. For instance, a dataflow channel from an input function to a vulnerable action is nearly always present in software that has an exploitable buffer overflow vulnerability. When we look at analysis algorithms and then again when I look at rules, I go into more detail on taint propagation. Static analysis methods are not the only ones that may track contaminated data. The Perl taint mode, which uses a runtime method to ensure that user-supplied data are checked against a regular expression before they are utilised as a part of a sensitive operation, is probably the most well-known implementation of dynamic taint propagation.

**Pointer Aliasing**
Another dataflow issue is pointer alias analysis. Understanding which pointers can potentially refer to the same memory location is the goal of alias analysis. Alias analysis techniques use phrases like "must alias," "may alias," and "cannot alias" to characterise pointer relationships. The correctness of alias analysis is a requirement for many compiler optimizations. Static analysis tools frequently assume that pointers, or at least pointers supplied as function arguments, do not alias. This premise appears to be true frequently enough for many tools to generate valuable findings, but it could lead a tool to miss out on significant outcomes.

### Analysis Algorithms

The goal of utilising sophisticated static analysis methods is to increase context sensitivity to identify the contexts and circumstances in which a specific piece of code executes. A more accurate evaluation of the hazard the code poses is made possible by improved context sensitivity. An *intraprocedural analysis* component for examining a single function and an *interprocedural analysis* component for examining interactions between functions make up the minimum of two major components of any advanced analysis technique. We use the phrases local analysis to refer to intraprocedural analysis and global analysis to refer to interprocedural analysis because the names intraprocedural and interprocedural are so close.

### Rules

The rules that specify what a security tool should report are equally, if not even more, crucial than the analysis methods and heuristics that the tool employs. Although the analysis algorithms do the majority of the work, the rules are ultimately responsible. Analysis algorithms occasionally have a lucky break and draw incorrect conclusions for the right reasons, but a tool can never alert you to a problem outside the bounds of its rule set. To find a problem, several rules may be used, and each rule may make reference to an abstract interface or check method names against a regular expression. More rules do not always result in a better static analysis tool, just as more code does not always result in a better program. Sometimes the rules that code quality tools are analyzing are derived from the code itself. For detecting security issues, this statistical method of inferring rules does not perform very well. The code may consistently apply the construct wrongly throughout the program if a programmer does not realize that a particular construct poses a security risk, which would provide a 100 % false negative rate if merely a statistical technique were used. Rules are not just used to specify security attributes. Additionally, they are employed to define any programmed behaviour that is implicit in the programming language, such as that of the system or external libraries that the program may use. A good set of modelling rules for system libraries and well-known third-party libraries requires a lot of work to develop and maintain. Good static analysis tools try to express the rules they examine, allowing for the addition, removal, or modification of rules without altering the tool itself. All of the rules that the top static analysis tools examine are transmitted. An external rules interface enables the end user to add checks for new types of errors or to extend current checks in ways that are particular to the semantics of the programme being analysed, in addition to modifying the out-of-the-box behaviour of a tool. The rule format can be adjusted to the capabilities of the analysis engine by maintaining external files that use a certain format

for specifying rules. In some cases, it is advantageous for rules to appear as annotations right within the program's text. If a certain module is subject to special rules, including those rules directly in the module (or the module's header file) is an excellent approach to ensure that the rules are followed each time the module is used. Because an annotation's context is already provided by the code surrounding it, it can be written more concisely than rules that are found in other files. For instance, an annotation can just appear right before the function declaration in place of needing to give the function's name. There are drawbacks to these close links to the source code. For instance, they might not be permitted to add permanent annotations if the persons conducting the analysis are not the owners or maintainers of the code. By making special source files that almost entirely include annotations and using those source files purely for analysis, it could be possible to get around this kind of restriction. Some annotations typically take the form of comments that have been carefully prepared. More applications than just static analysis benefit from annotations. Static analysis needs to use a variety of different rule types to solve taint spreading problems. I list the different taint propagation rule types here because so many security issues can be described as taint propagation issues:

- Program areas where contaminated data enters the system are defined by **source** rules.
- Program areas that shouldn't get contaminated data are specified by **sink** rules.
- **Pass-through** rules specify how a function deals with tainted data.
- A **cleanse** rule is a type of pass-through rule that cleans up a variable of taint. Input validation functions are represented by clean rules.
- **Entry-point** rules are similar to source rules in that they introduce taint into the program, but entry-point functions are invoked by an attacker rather than at locations in the programme when the function is called.

### Reporting Results

The value that the tool offers is significantly impacted by how it presents the results. The results must be provided so that the user may assess the accuracy and significance of the finding and decide whether to take the necessary corrective action. This procedure might include changing the code, but it might also involve modifying the tool. The phrase *false positive* is frequently used by tool users to describe anything that could be considered an unwanted result. It doesn't matter how sophisticated the underlying analytical algorithms are from the user's perspective. The outcome is meaningless if you

can't understand what the tool is trying to tell you. In that regard, poor outcomes can result from poor presentation just as readily as they might from the poor analysis. The tool's responsibility includes presenting results in a way that allows users to determine their potential significance. It's crucial to include basic code navigation tools like jump to definition. Everyone benefits if a static analysis tool is available as a plug-in for an integrated development environment (IDE) for programmers. The static analysis tool developers don't have to invent code browsing, and the programmer benefits from a comfortable arrangement for code navigation. Auditors need at least three features for managing tool output:

- **Grouping and sorting results**: Users can frequently eliminate a significant number of undesirable results without having to review each issue individually if they can organise and classify difficulties flexibly. Users like having results presented in a ranked order since static analysis techniques might provide a lot of data, making it harder to find the most meaningful results. Tools for static analysis can rank data along two dimensions. If the tool is a mistake, severity indicates how serious the discovery is. In general, a tool's confidence in a result decreases as more assumptions are required to reach that conclusion. A tool must add the severity and confidence scores for each outcome to provide a ranking. Usually, confidence and severity are combined into a straightforward discrete scale of importance, such as Critical (C), High (H), Medium (M), and Low (L). This makes it simple for auditors to set priorities for their job.

- **Eliminating unwanted results**: The ability to suppress results so that they are not reported in later analysis runs is a feature that all sophisticated static analysis systems offer. In a functioning system, suppression data will be carried over to upcoming releases of the same codebase. Users must have the option to disable entire categories of warnings in addition to being able to fix specific faults. There should be a means to save suppression information outside the code if the person conducting the code review does not have authorization to edit the code, as is the case with many tools that allow results to be suppressed using pragmas or code annotations. Simply storing the filename, line number, and issue kind is one option. The issue is that any change to the file, no matter how slight, can cause all the line numbers to change, invalidating the suppression data. By storing a line number as an offset from the function's beginning or the closest named statement, this issue can be mitigated. Another strategy is to create an identifier for the result based on the programme components that make up the trace. This strategy is particularly helpful if a result consists of a trace through the programme rather than simply a single line. The names of functions and variables, pertinent segments of the

control flow graph, and IDs for any rules involved in determining the outcome are all useful inputs for creating the identifier.

- **Explaining the significance of results**: A description of the issue, an explanation of whom it affects or why it is significant, and the procedures required to duplicate the issue are all included in static analysis tools. The tool must describe the danger it has found as well as any potential effects of an exploit.

# Chapter 3

# OWASP Evaluation Method

We'll concentrate on the evaluation methodology used in this thesis to analyze web applications using SAST tools. A nonprofit organisation called the Open Web Application Security Project(OWASP) aims to increase the security of software. The OWASP Foundation serves as a resource for developers and technologists to secure the web through community-led open-source software projects, hundreds of local chapters globally, tens of thousands of members, and premier educational and training conferences. They provide forums, tools, videos, and documentation among other things. [25] The OWASP Top 10 is a project for which they are best recognized and which lists the 10 most important security issues for web application security. Data from various organisations are gathered for the report by a group of international security specialists, who subsequently analyse it. This also indicates that OWASP does not define the Top10 alone; rather, it gathers information from a wide range of sources and organisations before making it available to us for feedback. The analysis is highly interesting and resulted in forty-three CWE for the Top 10. The OWASP Benchmark is another well-known project that OWASP has created. [26] A Java test suite called the OWASP Benchmark Project was created to assess the precision, depth, and speed of automated software vulnerability detection technologies. It is challenging to comprehend these instruments' strengths and drawbacks and evaluate them against one another without the ability to measure them. To better comprehend the many metrics employed for the study of web applications by static analysis tools, I will now show this project.

## 3.1 OWASP Benchmark

A free and open test suite called the OWASP Benchmark for Security Automation (OWASP Benchmark) was created to assess the efficiency, scope, and precision of automated software vulnerability detection tools and services. Each iteration of the OWASP Benchmark includes thousands of fully

executable and exploitable test cases, each of which corresponds to the relevant CWE number for that vulnerability. Java is used to implement Benchmark's current version. Other languages could be included in later versions. OWASP Benchmark can be used to analyze any type of Application Security Testing (AST) tool, including SAST, DAST and IAST tools.

**Score**
When security tools (SAST, DAST, and IAST) identify a sophisticated vulnerability in your code, they are truly fantastic. However, because the precise vulnerabilities that automated tools cover are so widely misunderstood, end users frequently feel insecure. You want to evaluate how effective these technologies are at identifying and correctly diagnosing application security issues. The test suite examines genuine and fictitious vulnerabilities. There are four possible results in the Benchmark:

- **True Positive (TP)**: Tool identifies a real vulnerability in a correct way.
- **False Positive (FP)**: Tool does not ignore a false alarm.
- **True Negative (TN)**: Tool ignores a false alarm in a correct way.
- **False Negative (FN)**: Tool does not identify a real vulnerability.

You can learn a lot about a tool from these four metrics. The evaluation of security tools in comparison to the Benchmark is shown in the graphic below. In the OWASP graph, the dashed red line establishes a line that closely corresponds to random guessing. A point plotted on this graph offers a visual representation of how well a tool performed while taking into account both the True Positives and False Positives it reported. Additionally, we want to determine a unique value for that point between -100 and 100, which we refer to as the Benchmark Accuracy Score. The Benchmark Accuracy Score is essentially a Youden Index, a common method for summing up the accuracy of a set of tests. It is also a general indicator of a test's performance that is used to assess a diagnostic procedure's overall discriminative capacity and to compare this test to others. Youden's index is determined by subtracting 1 from the total of a test's sensitivity and specificity, which is presented as a fraction rather than a percentage: (sensitivity + specificity) - 1. Youden's index is equivalent to zero for a test with poor diagnostic accuracy and one for a perfect test. The distance from the spot on the graph to the diagonal guessing line represents the Benchmark Score. It should be noted that if a point is below the line, the Benchmark score may be negative. This occurs when the True Positive Rate is lower than the False Positive Rate.

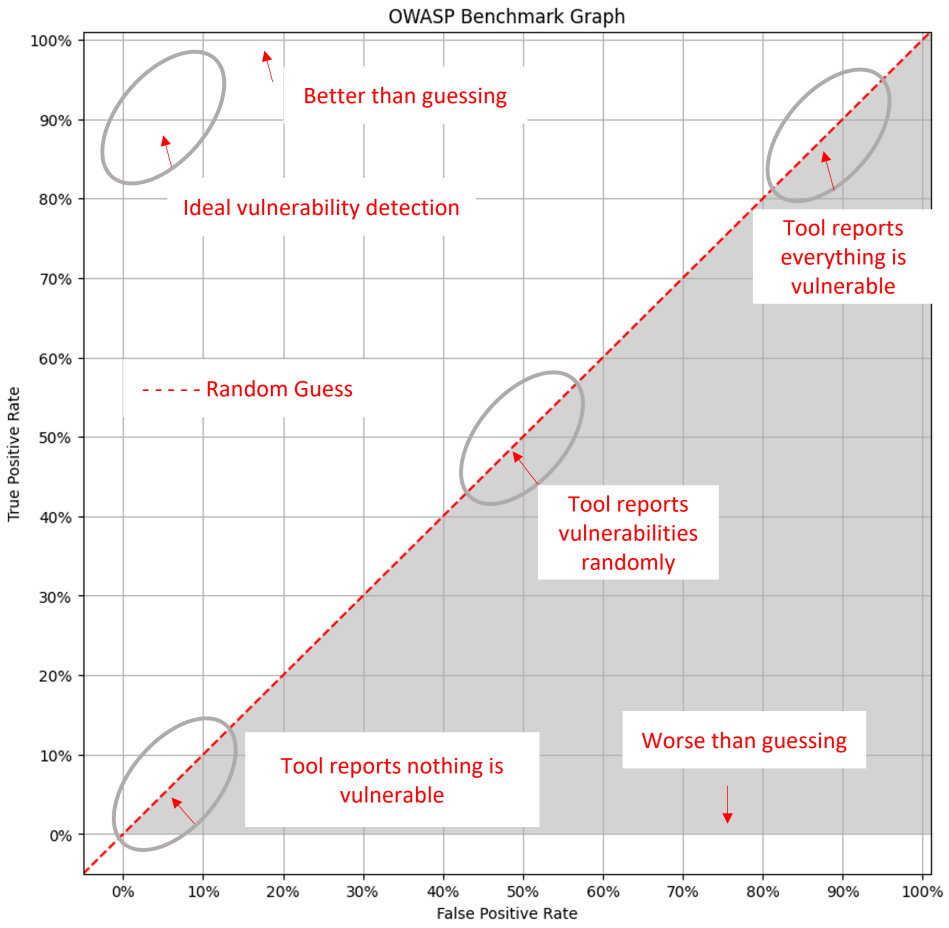


Figure 3.1: Image that represents general OWASP Benchmark Results

## 3.2 Evaluation Method Used

Throughout the entire project, the same methodology used within OWASP's benchmark is applied. The main components of the benchmark—the score and report generation—are used throughout the entire project. When calculating the score, the same benchmark outcomes are always used: TP, FP, TN, and FN. The project uses Benchmark Accuracy Score (BAS) to plot the various tools for each application under analysis on the graph.

**Reporting Format**
We use the following tabular format to record all the data obtained throughout the evaluation because the tools produce the raw output.

Table 3.1: Representation of the report format used to gather the analysis results from the various tools' applications.

| Security Category | TP | FP | TN | FN | Total | TPR | FPR | Score |
|---|---|---|---|---|---|---|---|---|
| **Command Injection** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Weak Cryptography** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Weak Hashing** | ... | ... | ... | ... | ... | ... | ... | ... |
| **LDAP Injection** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Path Traversal** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Secure Cookie Flag** | ... | ... | ... | ... | ... | ... | ... | ... |
| **SQL Injection** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Trust Boundary Violation** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Weak Randomness** | ... | ... | ... | ... | ... | ... | ... | ... |
| **XPATH Injection** | ... | ... | ... | ... | ... | ... | ... | ... |
| **XSS (Cross-Site Scripting)** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Other categories** | ... | ... | ... | ... | ... | ... | ... | ... |
| **Total Test Cases** | ... | ... | ... | ... | ... | ... | ... | ... |

The OWASP benchmark uses the same security categories as those listed in the report. The categories refer to the most prevalent issues that are present in the different web apps. The OWASP Top Ten project, which identifies the 10 most important security concerns for web applications, is also connected to these categories. The following security categories were used in the report:

- **Command Injection**: An attack known as command injection aims to use a weak application to execute arbitrary commands on the host operating system. When an application sends unsecured user-supplied data (forms, cookies, HTTP headers, etc.) to a system shell, command injection attacks are conceivable. In this attack, the vulnerable application's privileges are typically used to execute the operating system commands supplied by the attacker. Attacks using command injection are largely made possible by inadequate input validation. In contrast to this attack, code injection enables the attacker to insert custom code that the programme will then run. Without having to inject code, the attacker can enhance the application's default capability, which allows it to run system commands.

- **Weak Cryptography**: Sensitive data exposure, key leakage, broken authentication, insecure sessions, and spoofing attacks can all be caused by improper application of encryption methods. Some hashing or encryption techniques, such as MD5 and RC4, are known to be insecure and are not advised for use. The proper application of parameters is just as important for the security level as making the right

decisions on safe encryption or hash techniques. For instance, it is not advised to utilise ECB (Electronic Code Book) mode for asymmetric encryption.

- **Weak Hashing**: Some hashing or encryption techniques, such as MD5 and RC4, are known to be insecure and are not advised for use. The proper application of parameters is just as important for the security level as making the right decisions on safe encryption or hash techniques.

- **LDAP Injection**: An attack known as LDAP Injection targets web-based applications that build LDAP statements based on user input. It is possible to alter LDAP statements via a local proxy when an application doesn't properly sanitise user input. As a result, arbitrary commands could be executed, providing access to unauthorised queries and changing the content of the LDAP tree. LDAP Injection can be exploited using the same sophisticated exploitation methods as SQL Injection.

- **Path Traversal**: An attempt to access files or directories kept outside the web root folder is known as a path traversal attack (also known as a directory traversal attack). It may be feasible to access any files and directories stored on the file system, including vital system files and application source code, by manipulating variables that reference files with "dot-dot-slash (../)" sequences and their variations. It should be remembered that system operational access control restricts access to files (such as in the case of locked or in-use files on the Microsoft Windows operating system). Dot-dot-slash, directory traversal, directory climbing, and backtracking are other names for this attack.

- **Secure Cookie Flag**: When providing a new cookie to the user within an HTTP Response, the application server has the option of setting the secure attribute. Since the cookie is transmitted in clear text, the objective of the secure attribute is to prevent the display of cookies by unauthorized parties. Browsers that offer the security feature will only send cookies with the secure attribute when the request is directed to an HTTPS page. If an HTTP request is not encrypted, the browser will not transmit a cookie with the secure attribute set. The secure property, when set, will stop a cookie over an unencrypted channel by the browser.

- **SQL Injection**: A SQL injection attack involves inserting, or "injecting," a SQL query through the client's input data into the program. A successful SQL injection exploit can read sensitive database data, alter database data (Insert/Update/Delete), carry out database administration tasks, recover the contents of a specific file that is present

on the DBMS file system, and in some situations, send commands to the operating system. An example of an injection attack is a SQL injection attack, in which predefined SQL commands are affected by the injection of SQL commands into data-plane input.

- **Trust Boundary Violation**: Web applications frequently combine trusted and untrusted data in the same data structures by accident, which can result in situations where unvalidated/unfiltered data is trusted or used. Before transporting data into trusted bounds, proper input validation and output encoding should be used on it.

- **Weak Randomness**: Cryptographic attacks cannot be resisted by common pseudo-random number generators. When a function that can provide predictable values is employed as a source of randomness in a situation where security is a concern, insecure randomness can result. Computers are deterministic machines and can't provide true randomness because of this. Pseudo-Random Number Generators (PRNGs) start with a seed and derive subsequent numbers from it to approach randomness algorithmically. PRNGs come in two varieties: statistical and cryptography. Although statistical PRNGs offer useful statistical qualities, they are not appropriate for use in situations where security depends on generated values being surprising because of their relatively predictable output and simplicity in replication. To solve this issue, cryptographic PRNGs provide more unpredictable output. A value must be impossible or extremely unlikely for an attacker to distinguish it from a truly random value to be cryptographically secure. In general, statistical PRNGs should not be utilised in security-sensitive scenarios if a PRNG method is not advertised as being cryptographically secure.

- **XPATH Injection**: Similar to SQL Injection, XPath Injection attacks happen when a website builds an XPath query for XML data using data supplied by the user. If the XML data is used for authentication, they might even be able to increase their rights on the website. XPath, a type of straightforward descriptive statement that enables an XML query to locate a piece of information, is used for XML querying. Similar to SQL, you can define specific attributes to look for and match patterns. When a website uses XML, it's typical to accept user input in the form of a query string to specify the content to look for and display on the page. To ensure that this input doesn't muck up the XPath query and return the incorrect data, it must be cleaned up. Since XPath is a standard language and its syntax and notation are always implementation independent, an automated attack is possible. As requests are made to the SQL databases, there are no distinct dialects. Access to the entire document is possible because there is no

level of access control. As we may have learned from SQL injection assaults, we won't experience any restrictions.

- **XSS (Cross-Site Scripting)**: In these situations, the application processes erroneous user-controlled data, which prompts the execution of malicious scripts. XSS flaws can provide attackers access to user data and/or let them insert HTML code into an online application.
- **Other Categories**: All the flaws and issues that weren't covered by the earlier security categories are included in this category.

**Evaluation of TNs and FNs**

When analysing apps for which not all real vulnerabilities are known, it is crucial to highlight how TNs and FNs are discovered. In the first phase, vulnerabilities of various applications are collected and categorised as TP or FP. Vulnerabilities that do not belong to the various tools are classified as FN or TN using the following process: if the vulnerability is a TP, it is classified as FN, and if it is an FP, it is classified as TN.

# Chapter 4

# Targets and Tools Selection

To evaluate the behaviour of the tools, in terms of vulnerability found, as a first phase we selected a sample of vulnerable targets. In particular, a target is an open-source web application. Some aspects have been taken into consideration when choosing the various applications that have been used. These criteria have been set up so that a search of the applications will be conducted to find a more limited set of acceptable and relevant results. The selection has three mandatory requirements: first, that the program is a web application; second that the web application has a client part and a server part; finally that the client part's target is written in Javascript and the server part's target is written in Javascript or Python. Thus, every target that was selected complies with these three criteria. While, for the second phase, we chose a sample of tools that have been selected to represent the most common products used in the market including both open-source and commercial. Then, for each couple of targets and tools, we analyzed the behaviour to evaluate the performance of the tool. To create a program that serves as a benchmark for the tools, we finally collected the findings in tables and extracted the most challenging pattern to identify from the results. The benchmark created will be used to compare and rank sixteen SAST tools with minimum changes in their configurations, seven SAST commercial tools and nine open-source tools. In the first phase true positive and false positive metrics will be obtained and in the second phase, other metrics will be calculated with the objective of ranking the tools having into account the different levels of criticality that the web applications can have. The basic concept is to execute the targets with a set of real-world vulnerable software as input, collect the vulnerabilities discovered by the SAST, and confirm their accuracy before using a limited set of metrics that condense the detection abilities of the tools to derive a ranking for each application scenario. It is impossible to create a benchmark for all SAST in all circumstances due to the wide range of applications built with diverse components and the diversity of vulnerability classes. As a result, a benchmark should be specifically

created or set up for a certain domain to enable making informed decisions while defining the components. In this work, choosing the classes of vulnerabilities that the target SAST will be able to identify directly impacts the workload. A representative sample of real software code that contains vulnerabilities should be used to build the workload. The targets and the tools that have been chosen are presented in this chapter.

## 4.1 Targets

The vulnerable targets will be presented in this part as a dataset on which to record the tool's performance. Applications written in javascript for both the server and the client, as well as applications written in python for the server and javascript for the client, are the sixteen targets that will be used to test the scanners. Since most web applications are created using these two languages, we have decided to adopt them as our reference programming languages. We suggest a procedure for identifying vulnerabilities and non-vulnerabilities in real software that combines manual evaluation with the findings of several SAST tools. The proposed method for developing the workload is shown in Figure 4.1, and it consists of two steps (illustrated in the image by the grey boxes). As shown in Figure 4.2, the technique for

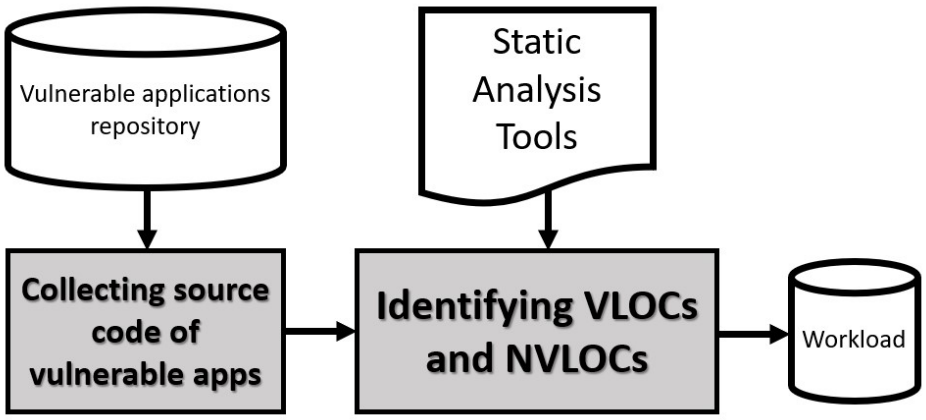


Figure 4.1: Process to compose the workload

choosing a representative group of vulnerable applications to establish the benchmark includes the following steps:

1. Selecting applications that have open-source code (SAST require the source of the application to detect vulnerabilities).

2. Selecting the vulnerability classes that apply to the benchmark domain.

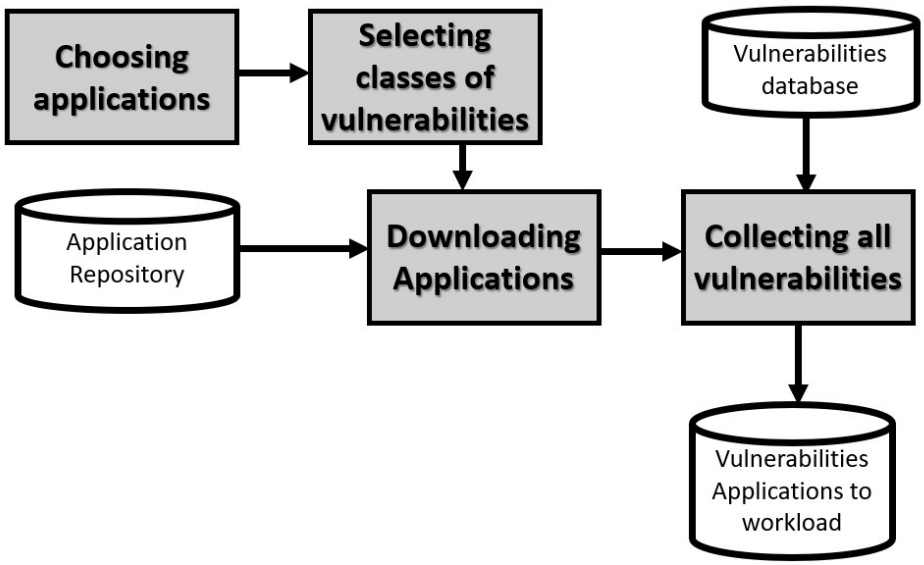


Figure 4.2: Process for collecting vulnerable applications.

3. Downloading the applications from source code repositories.

4. Collecting all vulnerabilities of the chosen applications registered in their development repository or from vulnerability databases.

Since the vulnerabilities in this methodology are representative of real applications and have been demonstrated to be exploitable, they have a significant advantage over existing benchmarks. The vulnerable targets' code is not published here to make the document concise and, therefore, not too large; instead, it can be obtained on their GitHub pages, which will be referenced. These are the targets used:

1. *Home-Cloud* [4]: Simple Web Application written in Javascript for client and server part in which the user can host his cloud at home. In total, there are 18564 lines of code.

2. *Websocket-chat* [38]: Simple Web Application written in Javascript for client-side and in Python for server-side. The application is a Websockets IRC-style chat server and client. In total, there are 157 lines of code.

3. *Hack-chat* [12]: Simple Web Application written in Javascript for client and server part in which is a minimal, distraction-free, accountless, disappearing chat service which is easily deployable as your service. In total, there are 7226 lines of code.

4. *Video-labelling-tool* [37]: Simple Web Application written in Javascript for client-side and in Python for server-side. The application is a tool

for labelling video clips (both front-end and back-end). In total, there are 9648 lines of code.

5. *Cinema-plus* [3]: Simple Web Application written in Javascript for client and server part in which Cinema + is an online Movie Ticket Booking web app with MERN Stack. The functionalities are the Online Booking System, Admin Dashboard and Dark Theme UI. In total, there are 11425 lines of code.

6. *StoreKeeper* [36]: Simple Web Application written in Javascript for client-side and in Python for server-side. StoreKeeper is an open-source, multilingual warehouse/store management software. In total, there are 13124 lines of code.

7. *StackOverflow-Clone* [34] [35]: Simple Web Application written in Javascript for client and server part in which is a clone of a famous Q/A website for professional and enthusiast programmers built using a completely different stack. In total, there are 9062 lines of code.

8. *Excel-to-json* [8]: Simple Web Application written in Javascript for client-side and in Python for server-side. The application converts files from excel to JSON, storing the information in a non-relational database (MongoDB), allowing the user to search data by a common identifier in the database. In total, there are 13260 lines of code.

9. *Slack-Clone* [31]: Simple Web Application written in Javascript for client and server part in which Full-Stack live chat application that recreates Slack's main features. In total, there are 40813 lines of code.

10. *Nano-SpeedTest* [22]: Simple Web Application written in Javascript for client-side and in Python for server-side. The application is used to test the speed of Nano Transactions. In total, there are 22607 lines of code.

11. *React-Social* [28]: Simple Web Application written in Javascript for client and server part in which is a social network. In total, there are 15082 lines of code.

12. *On my way* [23]: Simple Web Application written in Javascript for client-side and in Python for the server side. On My Way is a web application that algorithmically generates a tourism route for a given city and interests. In total, there are 1715 lines of code.

13. *Hello-books* [13]: Simple Web Application written in Javascript for client and server part in which is an application that provides users with access to books from wherever they are. Being a virtual library, users can borrow and read their favourite books using any device. In total, there are 33697 lines of code.

14. *Kiptab* [16]: Simple Web Application written in Javascript for client-side and in Python for server-side. Kiptab helps you and your friends keep track of expenses during vacations and other social settings by balancing debts automatically. In total, there are 9829 lines of code.

15. *Events-manager-io* [7]: Simple Web Application written in Javascript for client and server part which is a basic site for managing event centres and scheduling events. In total, there are 39235 lines of code.

16. *$\mu$uCrypt* [20]: Simple Web Application written in Javascript for client-side and in Python for server-side. $\mu$uCrypt Messenger is a secure and simple end-to-end encrypted chat. In total, there are 273 lines of code.

## 4.2 Tools Used

This section describes the resources utilised to examine the target web apps' source code for this thesis. especially the explanation of how the tools function and how they identify code vulnerabilities. The tools are chosen to reflect the most popular products on the market, as was described in the previous section. These tools come in both commercial and free versions. Additionally, another factor that we considered when choosing the tools is that they are typically found in the OWASP Benchmark. The choice of the tools constitutes the initial step. The Static Application Security Testing (SAST) tools were chosen because they are used to protect software by evaluating the source code of the software to discover sources of vulnerabilities and because the goal of this thesis is the static analysis of the code. SAST tools concentrate on the application's code content, as opposed to dynamic application security testing (DAST) tools, which assess application functionality in a black-box fashion. Because some vulnerabilities are simpler to identify through code analysis, it is better to analyse the source code. Some aspects have been taken into consideration when choosing the various tools that have been used. These criteria have been set up to conduct a selection of tools to find the ones that perform best. The selection includes three mandatory requirements: first, that the tool is easily configurable; second that the tool supports at least Javascript and Python language; finally, the tool must be able to work on public repositories without requiring a trial period. Table 4.1 lists all the various tools that have been selected. Only the languages taken into consideration throughout the entire process have been entered in the supported languages column. Each tool's licence type is displayed in the licence column. The majority of commercially licenced tools allow for free use on public repositories. Commercially licenced tools have also been chosen as a consequence. The tools highlighted are those that were used within the thesis because they were the best ones that satisfied the requirements.

Table 4.1: A collection of the several tools that have been considered.

| Tool | Supported Languages | Licence |
|---|---|---|
| Bandit | Python | Free |
| Codacy | Javascript - Python | Commercial |
| Codiga | Javascript - Python | Commercial |
| Coverity | Javascript - Python | Commercial |
| Deepsource | Javascript - Python | Commercial |
| Fluid Attack's Scanner | Javascript - Python | Commercial |
| Graudit | Javascript - Python | Free |
| Horusec | Javascript - Python | Free |
| HCL AppScan CodeSweep | Javascript - Python | Commercial |
| Insider CLI | Javascript | Free |
| LGTM | Javascript - Python | Free |
| Pysa | Python | Free |
| Semgrep | Javascript - Python | Free |
| Shiftleft Scan | Javascript - Python | Commercial |
| Sl Scan | Python | Free |
| Sonarcloud | Javascript - Python | Free |
| SNYK Code | Javascript - Python | Commercial |
| Debricked | Javascript - Python | Commercial |
| WhiteSource Bolt | Javascript - Python | Commercial |
| RATS | Python | Free |
| Reshift | Javascript | Commercial |
| Veracode | Javascript - Python | Commercial |

Then, all of the analysis instruments will be introduced. The following is the list:

1. **Bandit** [2]: is a comprehensive source vulnerability scanner for Python. It is an Open-Source tool. Bandit uses the ast module from Python's standard library to analyze Python code. The ast module is only able to parse Python code that is valid in the version of the interpreter from which it is imported.

2. **Coverity** [5]: is a tool to find and fix defects in open-source project for free. The tool is a commercial but open-source project for free. Coverity Scan is a service that provides analysis results on open-source projects. Defects are identified by the engine for quick and easy repair. It offers the results of the completed analysis of open-source projects.

3. **Deepsource** [6]: is a tool that helps ship clean and secure code with powerful static analysis, OWASP Top 10 compliance, and Autofix. Supports all major programming languages. It's a commercial tool

but an open-source project for free. The main features are advanced static analysis in which there is the largest collection of static analysis rules. Centrally track key metrics of code with documentation coverage, number of dependencies, and more. Code coverage in which seamless coverage insights and reporting are in a single dashboard. Continuous secrets scanning detects hardcoded security credentials and sensitive data in source code. Minimal configuration with no need to install or add anything in your build process because works out of the box. Highly accurate static analyzers, optimized for minimal noise in results. It protects code from critical security risks recommended by OWASP.

4. **Fluid Attack's Scanner** [15]: is a commercial tool that performs SAST, DAST and SCA vulnerability detection tool with perfect OWASP Benchmark score. It's a commercial tool but an open-source project for free. When running as a free and Open Source CLI tool, you are in charge of configuring the tool. It will scan vulnerabilities in the target of your choice and report results back to you in pretty-printed or CSV format.

5. **Horusec** [14]: is an open-source tool that analyses static code to find vulnerabilities in projects and stores all results in a database for analysis and generation of metrics. When Horusec starts an analysis, it follows the actions listed below: it will identify what are the current languages in your project; now, the tool will start the analysis searching for vulnerabilities; when the analysis finishes, it will show the analysis' output in the interface or the file.

6. **Insider CLI** [10]: is focused to make source code analysis to find vulnerabilities. It is focused on agile and easy-to-implement software.

7. **LGTM** [18]: is free for open-source static analysis service that automatically monitors commits to publicly accessible code in Bitbucket Cloud, GitHub, or GitLab. LGTM is a variant analysis platform that automatically checks your code for real CVEs and vulnerabilities. LGTM ranks the most relevant results to show only the alerts that matter. The concept of LGTM comes from the observation that the same bugs often reappear over and over again throughout a project's lifetime, and in multiple places in a codebase. They may also be present under different forms, called "variants". When such bugs lead to security vulnerabilities, the consequences can be pretty severe. The technology behind LGTM is CodeQL. Using CodeQL, you can write a query to find a bug in your projects. Once you've found the original issue and fixed it, you can extend the query to find code patterns that

are semantically similar to the original bug. The standard or built-in queries used on LGTM are all open-source and are constantly updated.

8. **Pysa** [24]: is a free open-source tool that can perform Taint Analysis to identify potential security problems using Python libraries. Its work is tracking flows of data from where they originate (sources) to where they terminate in a dangerous location (sinks). This analysis uses models that provide annotations about the source code and also rules that define which sources are dangerous for which sinks. The tool prevents false negatives, but it can also lead to false positives.

9. **Semgrep** [30]: is a free open-source tool that uses a static analysis engine for finding bugs, detecting dependency vulnerabilities and enforcing code standards. No compilation is needed to scan the source code and analyzes the code locally on your computer or in your build environment.

10. **Sonarcloud** [1]: is an open-source cloud-based code analysis service designed to detect code quality issues performing Static Analysis. The main features are the protection of the software assets - embedded, web, mobile apps, and cloud-native apps. Automatic analysis with no extra configuration is required for most languages to receive the results of the first analysis. The super-fast analysis gets super-fast feedback to help you quickly assess where the code stands in pull requests and branches. Actionable, highly precise results receiving clear reports at the right place and time. Maximize your impact with high-precision analysis that helps you focus on real issues, and less on false positives.

11. **SNYK Code** [32]: is a commercial tool that finds, learns and fixes vulnerabilities in open-source dependencies, in application code. It is free for open-source projects. Snyk Code utilizes a semantic analysis AI engine that learns from millions of open-source commits and is paired with Snyk's Security Intelligence database: this creates a continually growing code security knowledge base, which reduces false positives to near-zero and provides actionable findings with that matter. Snyk Code leverages its security knowledge base to provide fix examples from real-world projects, which offer insights on how to fix the discovered issues.

12. **Scan** [43]: is a free open-source security tool with integrated multi-scanner based design. It detects various kinds of security flaws in your application, and infrastructure code in a single fast scan without the need for any remote server.

13. **Debricked** [39]: is a commercial tool that identifies, fixes and prevents known vulnerabilities through automation without the need to give ac-

cess to your source code. It is free for open-source projects. Integrate, scan and receive the first results within minutes. It solves vulnerabilities more efficiently and generates automated pull requests for fixing them. Automatically keep new vulnerabilities from entering in codebase by customizing your very own rules and policies. Debricked only warns you when you're using the vulnerable function.

14. **Whitesource Bolt** [19]: is a commercial tool that finds and fixes Open Source Vulnerabilities getting detailed information on security vulnerabilities and suggesting fixes for quick remediation. It's free for open-source projects. The tool allows you to work on both public and private repositories.

15. **RATS** [11]: is a free open-source tool that performs only a rough analysis of source code flagging common security-related programming errors. It will not find every error and will also find things that are not errors. Manual inspection of your code is still necessary but greatly aided with this tool.

16. **Reshift** [29]: is a commercial tool that identifies and fixes vulnerabilities. It's free for open-source projects. The tool improves code security and provides rich content and best practices to learn about security while writing code.

The selected tools will be referred to below as Tool 1, Tool 2, Tool 3, Tool 4, Tool 5, Tool 6, Tool 7, Tool 8, Tool 9, Tool 10, Tool 11, Tool 12, Tool 13, Tool 14, Tool 15 and Tool 16.

**Identifying vulnerabilities and non-vulnerabilities**
As shown in Figure 4.2, the second grey box is related to identifying vulnerabilities and non-vulnerabilities. To evaluate a SAST, we must know which Lines Of Codes (LOCs) are vulnerable (i.e., positive examples or Vulnerable LOCs) and which LOCs are not (i.e., negative instances or Non-Vulnerable LOCs). This is a difficult task that requires a detailed inspection by security specialists for huge code bases, and the outcome may not be entirely correct because experts sometimes fail. As a result, a LOC with one or more vulnerabilities counts as one positive instance in this study since we count the vulnerabilities at the level of the LOC. Then, we go over how to define vulnerable LOCs and non-vulnerable LOCs and show how to obtain them.

a) *Characterizing Vulnerable LOCs and Non-Vulnerable LOCs*: The proposed strategy for discovering more Vulnerable LOCs in the workload is based on running multiple SAST tools and using a manual review to validate the results. To find the Vulnerable LOCs, we run the SAST to check the chosen apps for vulnerabilities. After combining the outputs, each potential vulnerability is manually examined

to determine whether it is a TP (i.e., vulnerability) or an FP (i.e., non-vulnerability). As a result, the list of positive instances (P) is created by merging the original Vulnerable LOCs with all TPs, and the list of negative instances (N), or non-vulnerable LOCs, is created by including all FPs (Non-Vulnerable LOCs).

b) *Obtaining Vulnerable LOCs and Non-Vulnerable LOCs*: The method for obtaining Vulnerable LOCs and Non-Vulnerable LOCs includes five steps:

   1) Determine the group of SAST that will be utilised to create the list of Vulnerable LOCs and Non-Vulnerable LOCs. This involves providing the configuration options for the chosen tools.
   2) Utilize the workload applications to run the SAST to find vulnerabilities. This stage produces a list of potential Vulnerable LOCs.
   3) Classify the vulnerabilities indicated by the tools as Vulnerable LOCs or Non-Vulnerable LOCs after manually verifying them.
   4) List the susceptible file, the vulnerable LOC, the vulnerable variable, the vulnerability class, and a description to describe the set of vulnerable LOCs.
   5) Describe the collection of non-vulnerable LOCs, giving details about the file, the LOC, the variable, the potential vulnerability class, and the description.

## 4.3 Procedure to run Benchmark

To implement and run the benchmark, a specific set of procedures and guidelines must be followed (as shown in Fig. 4.3):

1. *Preparation*: Select the SAST that will be compared. Tools must, whenever possible, be configured following the characteristics of applications in the benchmark domain because they are performed differently depending on their features, configurations, and user interfaces.

2. *Execution*: running the SAST under benchmarking to detect vulnerabilities in the workload.

3. *Normalization of reports*: Each tool produces results in a distinct format, so these results must be normalised and combined into a single report with a standard format that includes the following data for every vulnerability: the LOCs reported as vulnerable, the files where it was found, a description of the vulnerability, the reported CVSS Score, and the application where it was found.

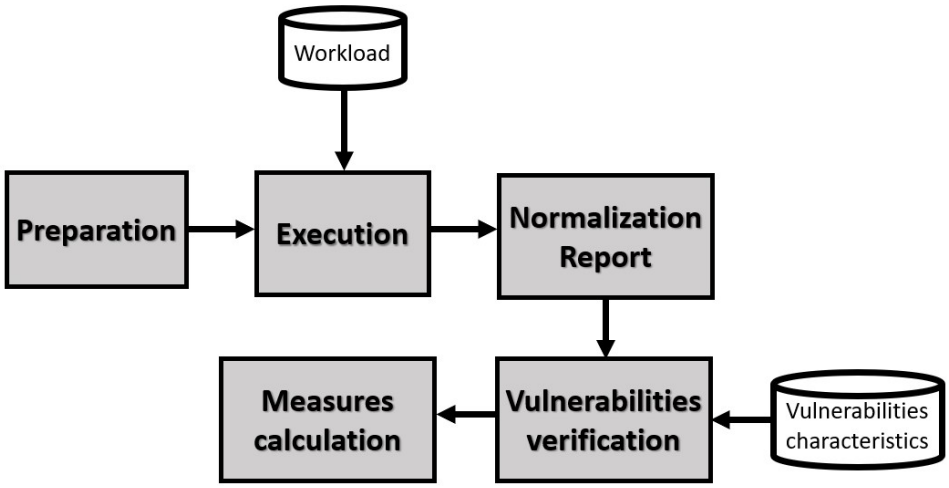


Figure 4.3: Process for collecting vulnerable applications.

4. *Vulnerability verification*: Analysis of the SAST tools' results by applying the vulnerabilities reported by the SAST performing manual verification and inserting to the list of VLOCs or NVLOCs.

5. *Metrics calculation*: based on the SAST outputs the benchmark metrics are calculated automatically.

# Chapter 5

# Experimental Results

In this chapter, we describe the experiment, its comparative outcomes, and how the data were interpreted. Multiple tables will be used to summarise all the results, and they will be followed by a more detailed explanation of the results. The methodology for comparing SAST tools used to analyse various web applications is then presented.

## 5.1 Experiment

The experiment includes the use of sixteen tools (Section 4.2) that are launched at sixteen various vulnerable targets (Section 4.1). Figuring out how the tools behave and perform is the aim of this analysis. First, it is important to understand what the four values mean:

- **True Positive (TP)**: A test result that indicates the presence of a vulnerability in a correct way.
- **True Negative (TN)**: A test result that indicates the absence of a vulnerability in a correct way.
- **False Positive (FP)**: A test result that indicates a particular vulnerability is present in a wrong way.
- **False Negative (FN)**: A test result that indicates a particular vulnerability is absent in a wrong way.

We suggest using metrics to compare the outcomes and rank the SAST. In practice, the metrics depend on the vulnerability detection goals, which are related with the amount of available resources to fix the vulnerabilities. Given that there are more Negative (N, non-vulnerabilities) instances than Positive (P, vulnerabilities) instances in the workload, the evaluation metrics are described in the following paragraphs. In a classification task, the *precision* for a class is the number of true positives divided by the number

of true positives plus the number of false positives. *Recall* in this context is defined as the number of true positives divided by the number of true positives plus the number of false negatives. The recall is also known as *True Positive Rate (TPR). False positive rate (FPR)* is another crucial measure which shows the likelihood of incorrectly rejecting the null hypothesis for a specific test. FPR is determined as the ratio of negative events that were mistakenly classified as positive (false positives) to all of the real negative occurrences (regardless of classification). Another important metric used is the *Benchmark Accuracy Score (BAS)* which is a standard way of summarizing the accuracy of a set of tests. This metric is equivalent to the Informedness metric normalized to the range -100 to 100. Other metric considered is the *F-measure* [27]. The accuracy of a test is measured by the F-measure. It is derived from the test's precision and recall. To summarize in a formula, precision Equation (5.1), recall Equation (5.2), FPR Equation (5.3), BAS Equation (5.4) and the F-measure (5.5) are defined as:

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

$$Recall(TPR) = \frac{TP}{TP + FN} \tag{5.2}$$

$$FPR = \frac{FP}{FP + TN} \tag{5.3}$$

$$BenchmarkAccuracyScore(BAS) = TPR - FPR \tag{5.4}$$

$$F - Measure = \frac{2 * TP}{2 * TP + FP + FN} \tag{5.5}$$

From this, we could extract some significant information regarding the tools. All runs were performed using a VM machine based on Kali-Linux at version 2021.3 running on 64-bit architecture. All tools were run on a VM with a two-core, Intel Core i7-7700HQ machine with 8GB of RAM. All the tools were run on it.

## 5.2 Results

This section reproduces several graphs and tables related to the tool's outcomes across a range of applications. In particular, for every application, screenshots of the tool's graphic on the OWASP graphic are displayed first. The tables related to each tool with the results are displayed subsequently.

### 5.2.1 Home-Cloud

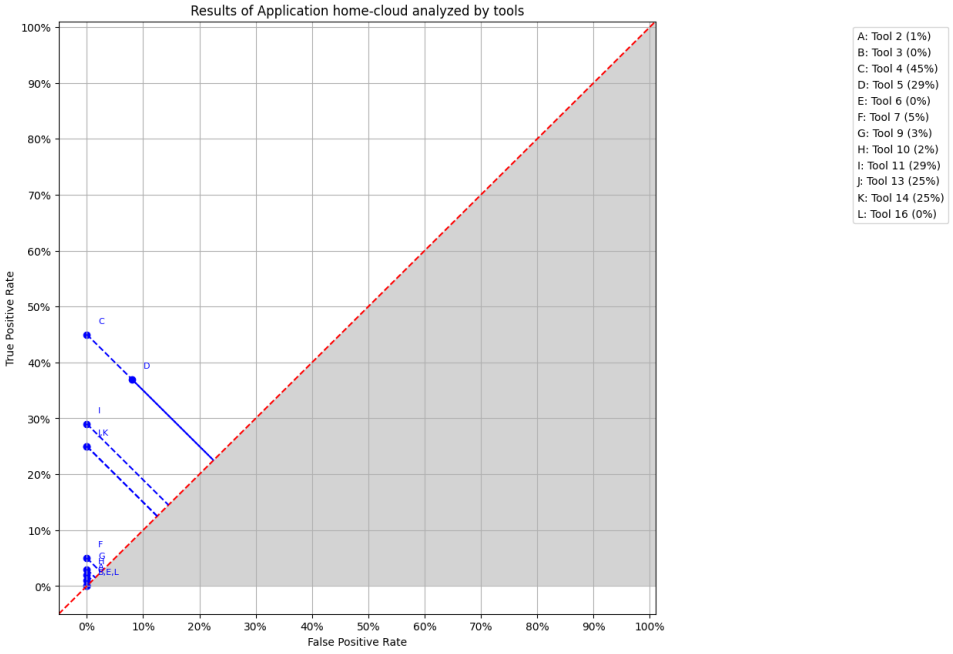


Figure 5.1: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question.

Table 5.1: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.011 | 0.0 | 0.011 | 1.1% |
| **Tool 3** | 0.0017 | 0.0 | 0.0017 | 0.17% |
| **Tool 4** | 0.45 | 0.0 | 0.45 | 45.0% |
| **Tool 5** | 0.3743 | 0.033 | 0.291 | 29.1% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.04507 | 0.0 | 0.04507 | 4.51% |
| **Tool 9** | 0.0343 | 0.0 | 0.0343 | 3.43% |
| **Tool 10** | 0.0242 | 0.0 | 0.0242 | 2.42% |
| **Tool 11** | 0.2856 | 0.0 | 0.2856 | 28.56% |
| **Tool 13** | 0.2530 | 0.0 | 0.2530 | 25.30% |
| **Tool 14** | 0.2465 | 0.0 | 0.2465 | 24.65% |
| **Tool 16** | 0.0017 | 0.0 | 0.0017 | 0.17% |

### 5.2.2 Websocket-chat

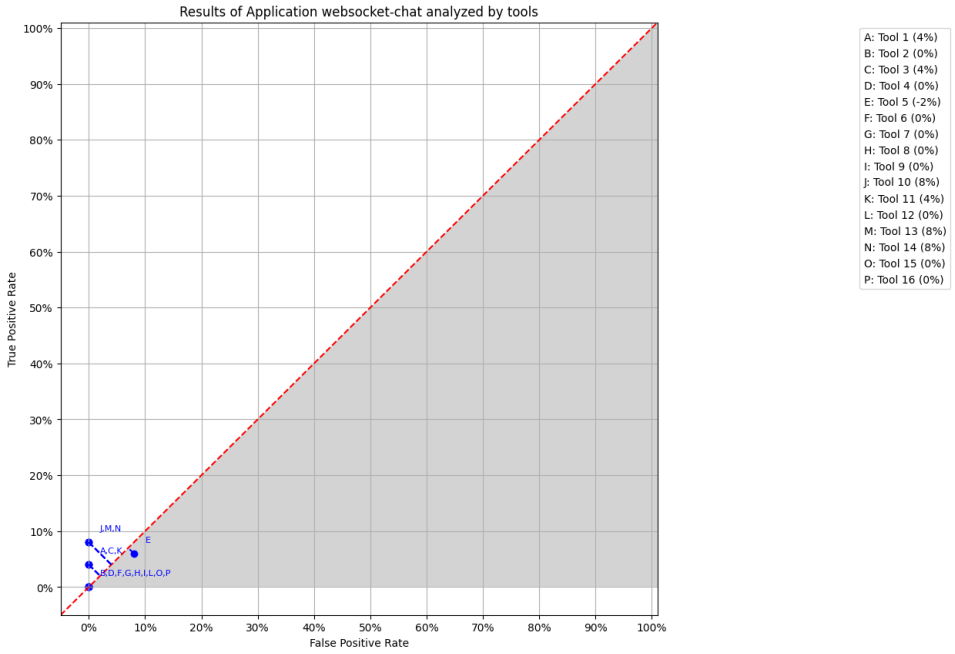


Figure 5.2: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.2: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.0417 | 0.0 | 0.0417 | 4.17% |
| **Tool 2** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 3** | 0.0417 | 0.0 | 0.0417 | 4.17% |
| **Tool 4** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 5** | 0.0625 | 0.083 | -0.02083 | -2.08% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 10** | 0.083 | 0.0 | 0.083 | 8.33% |
| **Tool 11** | 0.0417 | 0.0 | 0.0417 | 4.17% |
| **Tool 12** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 13** | 0.0833 | 0.0 | 0.0833 | 8.33% |
| **Tool 14** | 0.0833 | 0.0 | 0.0833 | 8.33% |
| **Tool 15** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 16** | 0.0 | 0.0 | 0.0 | 0.0% |

### 5.2.3 Hack-chat

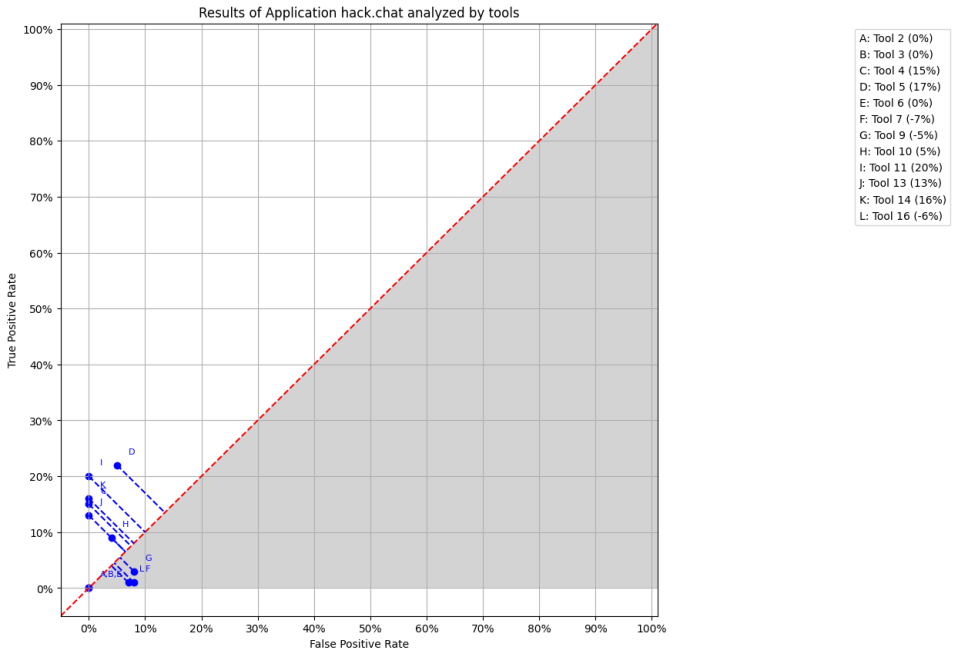


Figure 5.3: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question. The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.3: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.1474 | 0.0 | 0.1474 | 14.74% |
| **Tool 5** | 0.2173 | 0.0492 | 0.1681 | 16.81% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.01190 | 0.0833 | -0.0714 | 7.14% |
| **Tool 9** | 0.02690 | 0.07576 | -0.04886 | -4.89% |
| **Tool 10** | 0.0926 | 0.04167 | 0.0509 | 5.09% |
| **Tool 11** | 0.2034 | 0.0 | 0.2034 | 20.34% |
| **Tool 13** | 0.1307 | 0.0 | 0.1307 | 13.07% |
| **Tool 14** | 0.1618 | 0.0 | 0.1618 | 16.18% |
| **Tool 16** | 0.0119 | 0.072 | -0.0601 | -6.01% |

### 5.2.4 Video-labelling-tool

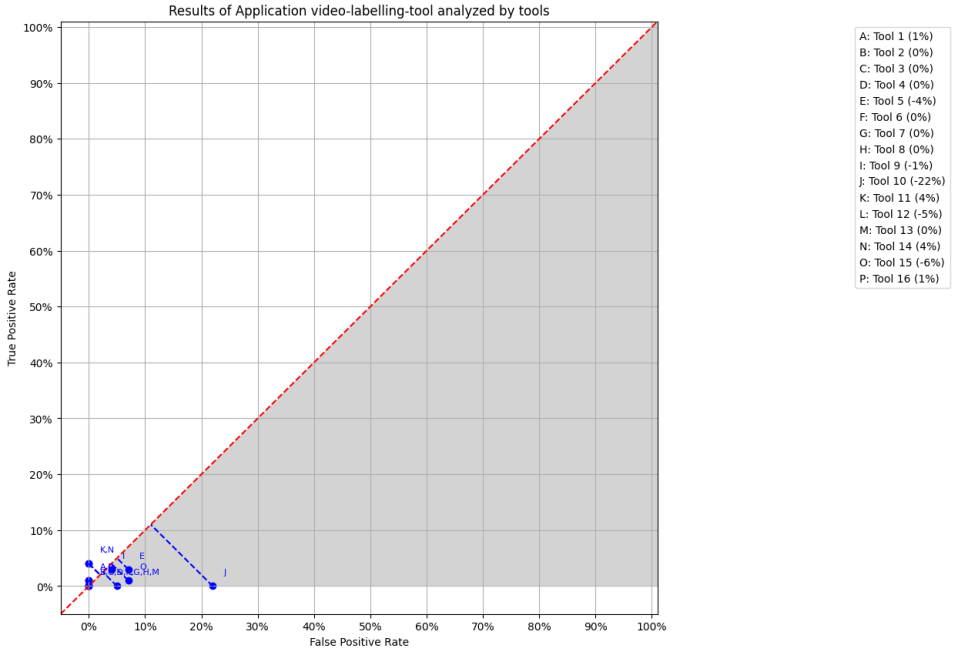


Figure 5.4: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.4: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.0208 | 0.0 | 0.0208 | 2.08% |
| **Tool 2** | 0.0 | 0.00149 | -0.00149 | -0.15% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 5** | 0.028 | 0.0684 | -0.0406 | -4.06% |
| **Tool 6** | 0.0 | 0.003 | -0.003 | -0.3% |
| **Tool 7** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.0278 | 0.0432 | -0.0154 | -1.54% |
| **Tool 10** | 0.0 | 0.2198 | -0.2198 | -21.98% |
| **Tool 11** | 0.04167 | 0.0 | 0.04167 | 4.17% |
| **Tool 12** | 0.0 | 0.1111 | -0.1111 | -11.11% |
| **Tool 13** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 14** | 0.04167 | 0.0 | 0.04167 | 4.17% |
| **Tool 15** | 0.0208 | 0.0803 | -0.0625 | -6.25% |
| **Tool 16** | 0.01389 | 0.0 | 0.01389 | 1.39% |

### 5.2.5 Cinema-plus

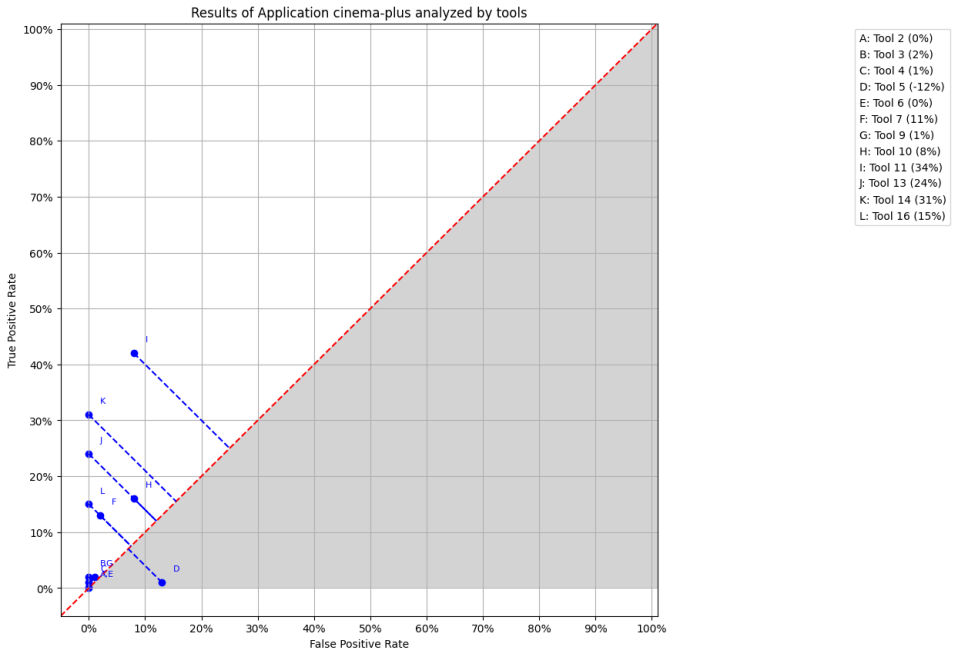


Figure 5.5: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question. The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.5: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.0 | 0.000706 | -0.000706 | -0.07% |
| **Tool 3** | 0.01667 | 0.0 | 0.01667 | 1.67% |
| **Tool 4** | 0.01164 | 0.0 | 0.01164 | 1.16% |
| **Tool 5** | 0.00694 | 0.1321 | -0.12516 | -12.52% |
| **Tool 6** | 0.002347 | 0.00071 | 0.00164 | 0.16% |
| **Tool 7** | 0.13088 | 0.01994 | 0.11094 | 11.1% |
| **Tool 9** | 0.01846 | 0.01271 | 0.00575 | 0.58% |
| **Tool 10** | 0.1616 | 0.0833 | 0.0783 | 7.83% |
| **Tool 11** | 0.41851 | 0.08404 | 0.33447 | 33.45% |
| **Tool 13** | 0.23988 | 0.0 | 0.23988 | 23.99% |
| **Tool 14** | 0.3056 | 0.0 | 0.3056 | 30.56% |
| **Tool 16** | 0.1523 | 0.0 | 0.1523 | 15.23% |

### 5.2.6 StoreKeeper

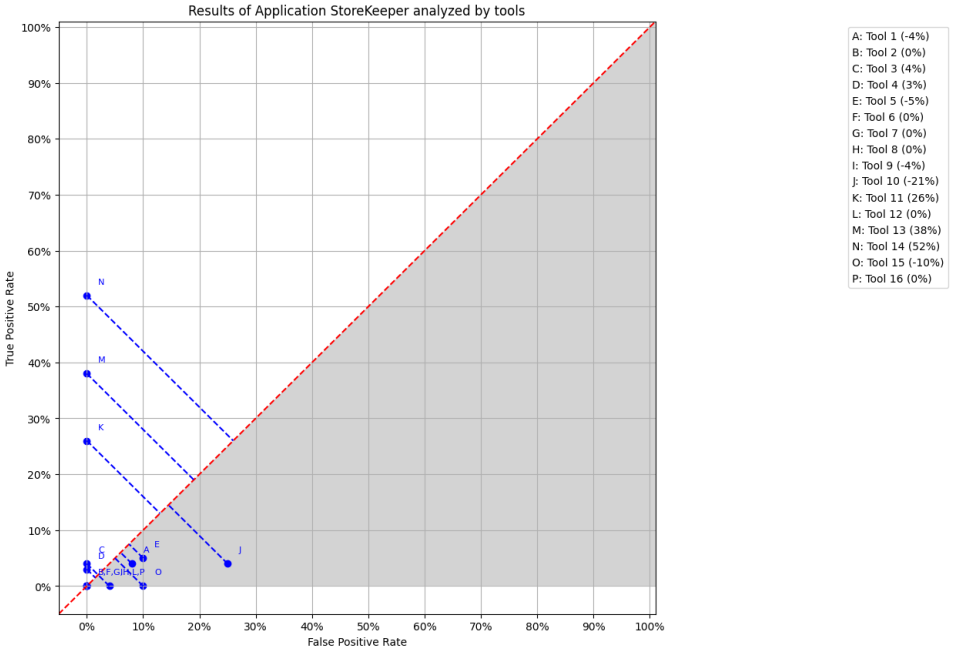


Figure 5.6: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.6: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.0875 | 0.0833 | -0.004167 | -0.42% |
| **Tool 2** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 3** | 0.04392 | 0.0 | 0.04392 | 4.39% |
| **Tool 4** | 0.02724 | 0.0 | 0.02724 | 2.72% |
| **Tool 5** | 0.0522 | 0.1 | -0.0478 | -4.78% |
| **Tool 6** | 0.00150 | 0.00278 | -0.00128 | -0.13% |
| **Tool 7** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.00150 | 0.04446 | -0.04296 | -4.3% |
| **Tool 10** | 0.0424 | 0.2505 | -0.2081 | -20.81% |
| **Tool 11** | 0.26148 | 0.0 | 0.26148 | 26.15% |
| **Tool 12** | 0.00273 | 0.00694 | -0.00421 | -0.42% |
| **Tool 13** | 0.3785 | 0.0 | 0.3785 | 37.85% |
| **Tool 14** | 0.51527 | 0.0 | 0.51527 | 51.53% |
| **Tool 15** | 0.0 | 0.125 | -0.125 | -12.5% |
| **Tool 16** | 0.0 | 0.0 | 0.0 | 0.0% |

### 5.2.7 StackOverflow-Clone

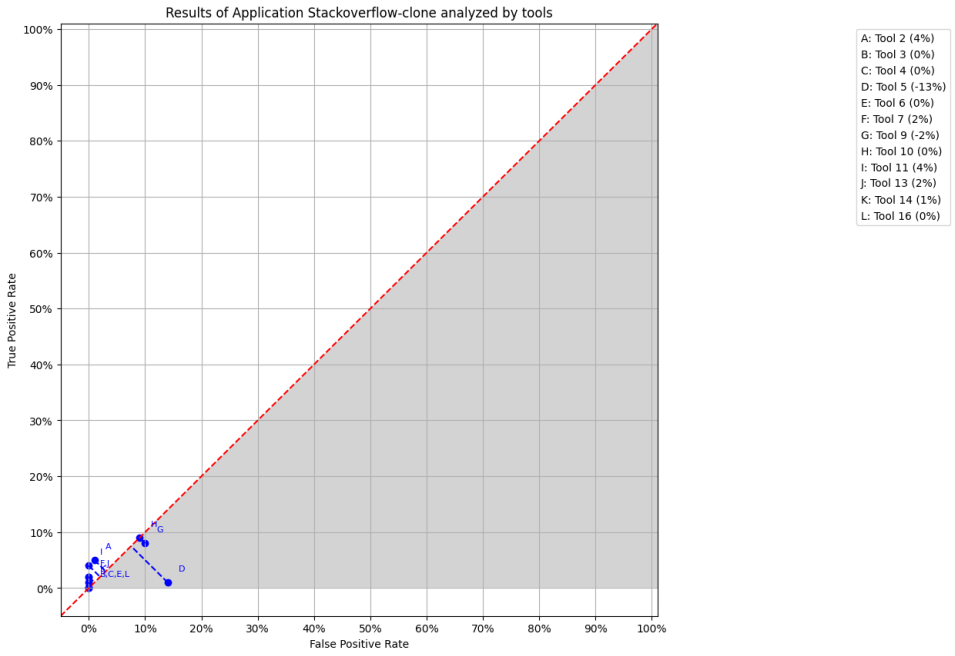


Figure 5.7: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question. The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.7: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.05159 | 0.01458 | 0.03701 | 3.7% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.003788 | 0.0 | 0.003788 | 0.38% |
| **Tool 5** | 0.01326 | 0.13675 | -0.12349 | -12.35% |
| **Tool 6** | 0.0 | 0.00214 | -0.00214 | -0.21% |
| **Tool 7** | 0.0171 | 0.0 | 0.01701 | 1.7% |
| **Tool 9** | 0.0833 | 0.1004 | -0.0171 | -1.71% |
| **Tool 10** | 0.08523 | 0.08547 | -0.00024 | -0.02% |
| **Tool 11** | 0.0352 | 0.0 | 0.0352 | 3.52% |
| **Tool 13** | 0.01667 | 0.0 | 0.01667 | 1.67% |
| **Tool 14** | 0.00945 | 0.0 | 0.00945 | 0.95% |
| **Tool 16** | 0.00379 | 0.0 | 0.00379 | 0.38% |

### 5.2.8 Excel-to-json

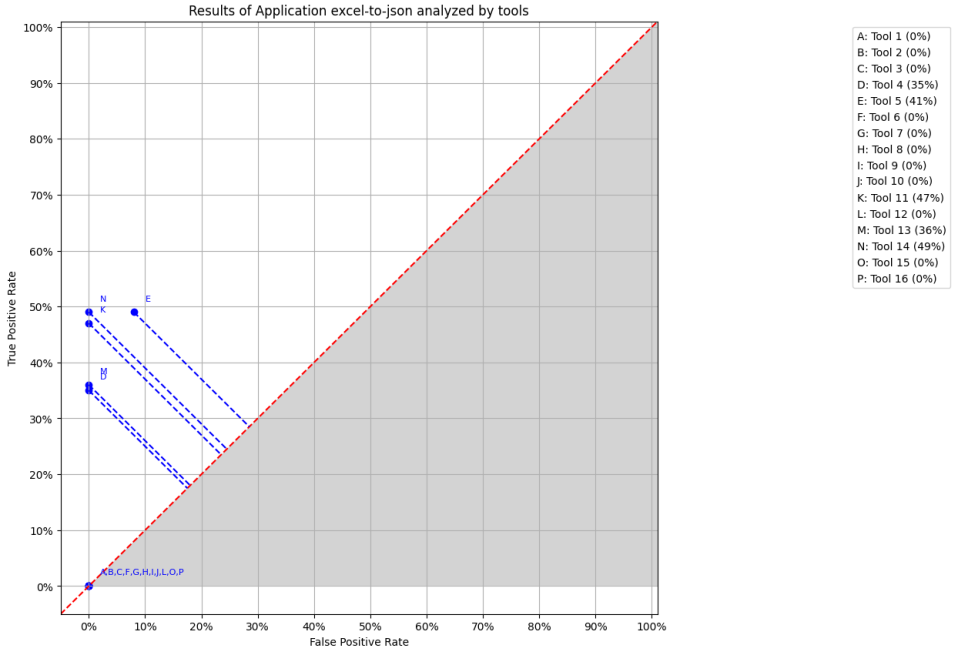


Figure 5.8: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS).

Table 5.8: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.0083 | 0.0 | 0.0083 | 0.83% |
| **Tool 2** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 3** | 0.000786 | 0.0 | 0.000786 | 0.08% |
| **Tool 4** | 0.3548 | 0.0 | 0.3548 | 35.48% |
| **Tool 5** | 0.490400 | 0.08333 | 0.407067 | 40.71% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 10** | 0.000786 | 0.0 | 0.000786 | 0.08% |
| **Tool 11** | 0.46955 | 0.0 | 0.46955 | 46.96% |
| **Tool 12** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 13** | 0.364278 | 0.0 | 0.364278 | 36.43% |
| **Tool 14** | 0.48791 | 0.0 | 0.48791 | 48.79% |
| **Tool 15** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 16** | 0.0 | 0.0 | 0.0 | 0.0% |

### 5.2.9 Slack-Clone

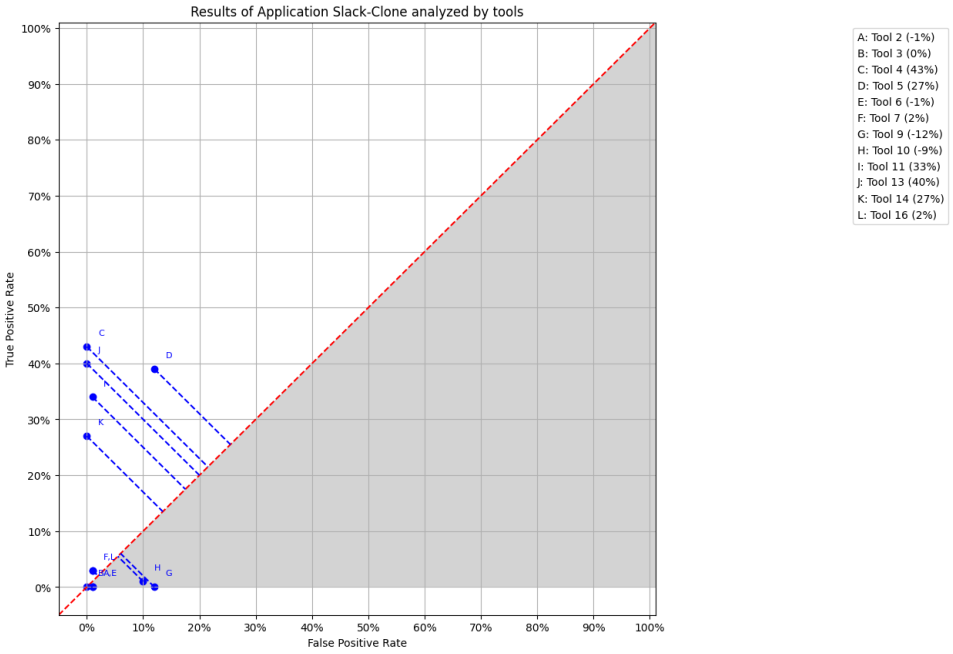


Figure 5.9: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question. The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.9: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.0033 | 0.0139 | -0.0106 | -1.06% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.4296 | 0.0 | 0.4296 | 42.96% |
| **Tool 5** | 0.3916 | 0.1174 | 0.2741 | 27.41% |
| **Tool 6** | 0.0 | 0.0145 | -0.0145 | -1.45% |
| **Tool 7** | 0.0348 | 0.0109 | 0.024 | 2.4% |
| **Tool 9** | 0.004 | 0.1196 | -0.1155 | -11.55% |
| **Tool 10** | 0.0145 | 0.1014 | -0.0869 | -8.69% |
| **Tool 11** | 0.3427 | 0.0072 | 0.3355 | 33.55% |
| **Tool 13** | 0.4038 | 0.0 | 0.4038 | 40.38% |
| **Tool 14** | 0.268 | 0.0 | 0.268 | 26.8% |
| **Tool 16** | 0.0271 | 0.0072 | 0.0198 | 1.98% |

### 5.2.10 Nano-SpeedTest

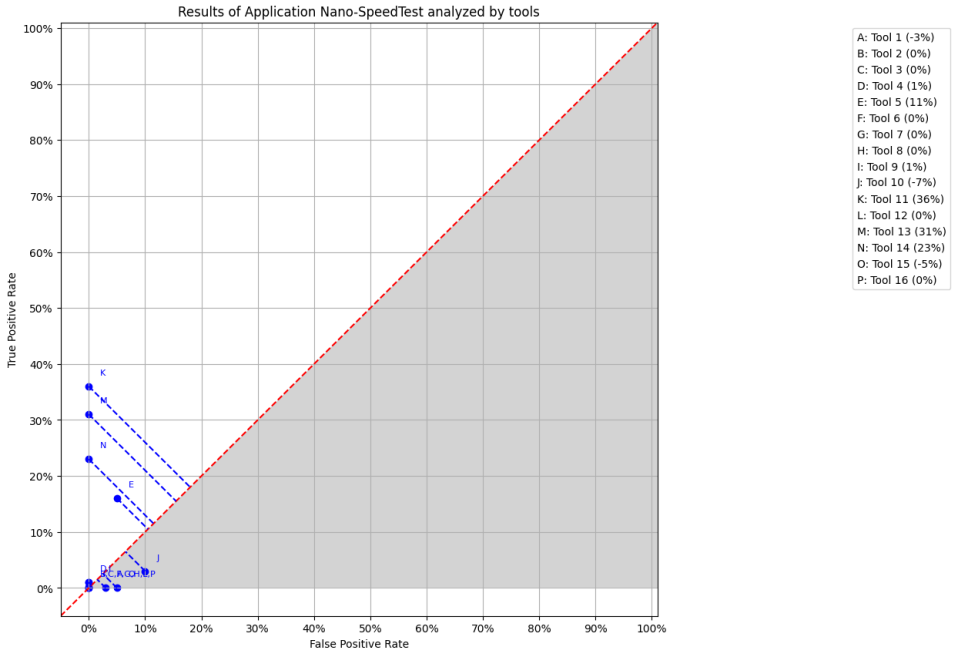


Figure 5.10: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.10: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.0021 | 0.0333 | -0.0312 | -3.12% |
| **Tool 2** | 0.0013 | 0.0 | 0.0013 | 0.13% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.0065 | 0.0 | 0.0065 | 0.65% |
| **Tool 5** | 0.1633 | 0.0452 | 0.1181 | 11.81% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.0126 | 0.0 | 0.0126 | 1.26% |
| **Tool 10** | 0.0342 | 0.1048 | -0.0706 | -7.06% |
| **Tool 11** | 0.3643 | 0.0 | 0.3643 | 36.43% |
| **Tool 12** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 13** | 0.3116 | 0.0 | 0.3116 | 31.16% |
| **Tool 14** | 0.225 | 0.0 | 0.225 | 22.5% |
| **Tool 15** | 0.0021 | 0.05 | -0.0479 | -4.79% |
| **Tool 16** | 0.0 | 0.0 | 0.0 | 0.0% |

### 5.2.11 React-Social

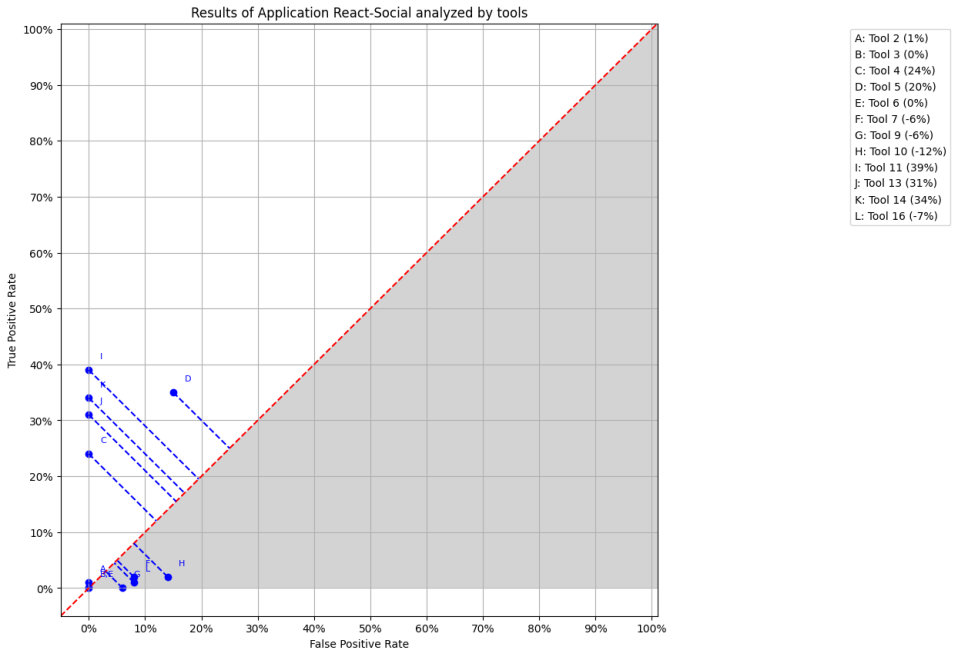


Figure 5.11: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question. The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.11: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.0074 | 0.0 | 0.0074 | 0.74% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.2403 | 0.0 | 0.2403 | 24.03% |
| **Tool 5** | 0.3454 | 0.1458 | 0.1996 | 19.96% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.0181 | 0.0833 | -0.0653 | -6.53% |
| **Tool 9** | 0.0008 | 0.0556 | -0.0547 | -5.47% |
| **Tool 10** | 0.02 | 0.1389 | -0.1189 | -11.89% |
| **Tool 11** | 0.3856 | 0.0 | 0.3856 | 38.56% |
| **Tool 13** | 0.3068 | 0.0 | 0.3068 | 30.68% |
| **Tool 14** | 0.3426 | 0.0 | 0.3426 | 34.26% |
| **Tool 16** | 0.0142 | 0.0833 | -0.0692 | -6.92% |

### 5.2.12 On my way

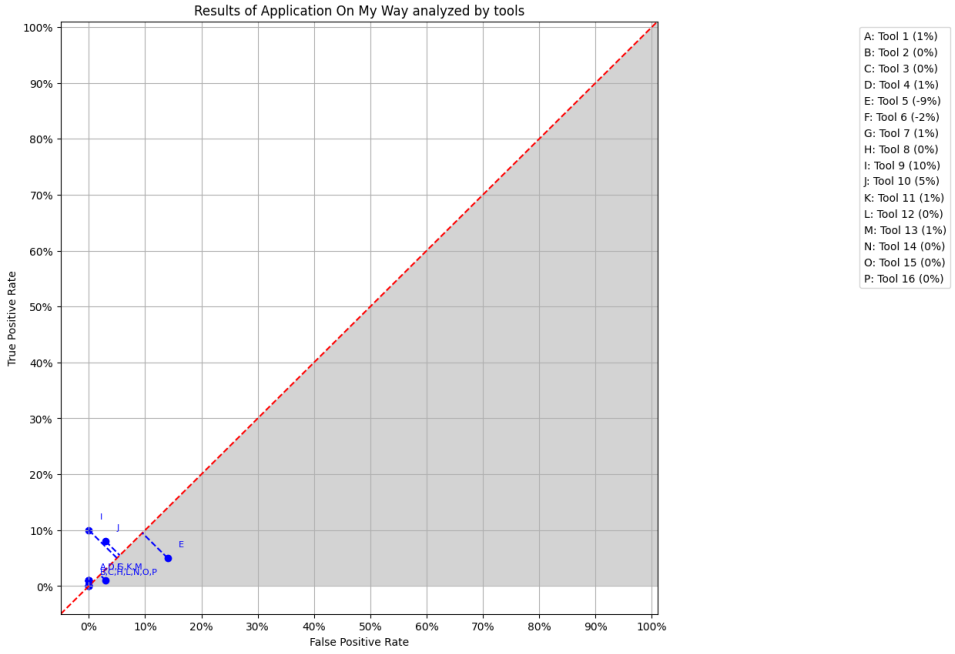


Figure 5.12: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.12: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.0278 | 0.0 | 0.0278 | 2.78% |
| **Tool 2** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.0111 | 0.0 | 0.0111 | 1.11% |
| **Tool 5** | 0.05 | 0.1389 | -0.0889 | -8.89% |
| **Tool 6** | 0.0056 | 0.0278 | -0.0222 | -2.22% |
| **Tool 7** | 0.0056 | 0.0 | 0.0056 | 0.56% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.1 | 0.0 | 0.1 | 10.0% |
| **Tool 10** | 0.0833 | 0.0278 | 0.0556 | 5.56% |
| **Tool 11** | 0.0056 | 0.0 | 0.0056 | 0.56% |
| **Tool 12** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 13** | 0.0056 | 0.0 | 0.0056 | 0.56% |
| **Tool 14** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 15** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 16** | 0.0 | 0.0 | 0.0 | 0.0% |

### 5.2.13 Hello-books

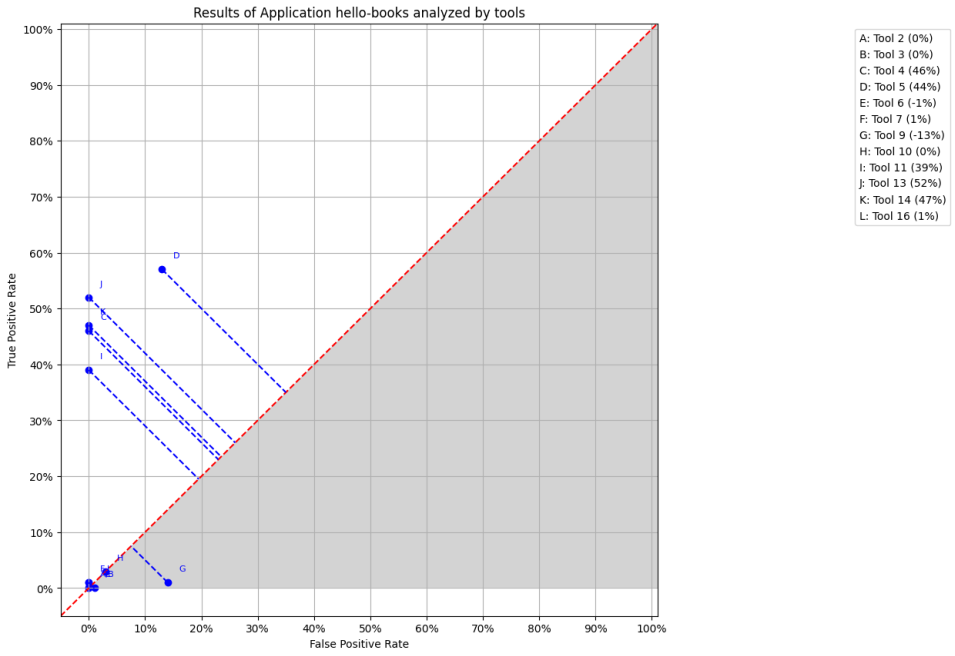


Figure 5.13: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question. The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.13: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.0014 | 0.0 | 0.0014 | 0.14% |
| **Tool 3** | 0.0009 | 0.0 | 0.0009 | 0.09% |
| **Tool 4** | 0.462 | 0.0 | 0.462 | 46.2% |
| **Tool 5** | 0.5712 | 0.13 | 0.4412 | 44.12% |
| **Tool 6** | 0.0 | 0.0088 | -0.0088 | -0.88% |
| **Tool 7** | 0.012 | 0.0 | 0.012 | 1.2% |
| **Tool 9** | 0.0129 | 0.1447 | -0.1318 | -13.18% |
| **Tool 10** | 0.0292 | 0.0317 | -0.0025 | -0.25% |
| **Tool 11** | 0.3854 | 0.0 | 0.3854 | 38.54% |
| **Tool 13** | 0.5187 | 0.0 | 0.5187 | 51.87% |
| **Tool 14** | 0.4715 | 0.0 | 0.4715 | 47.15% |
| **Tool 16** | 0.0055 | 0.0044 | 0.0011 | 0.11% |

### 5.2.14 Kiptab

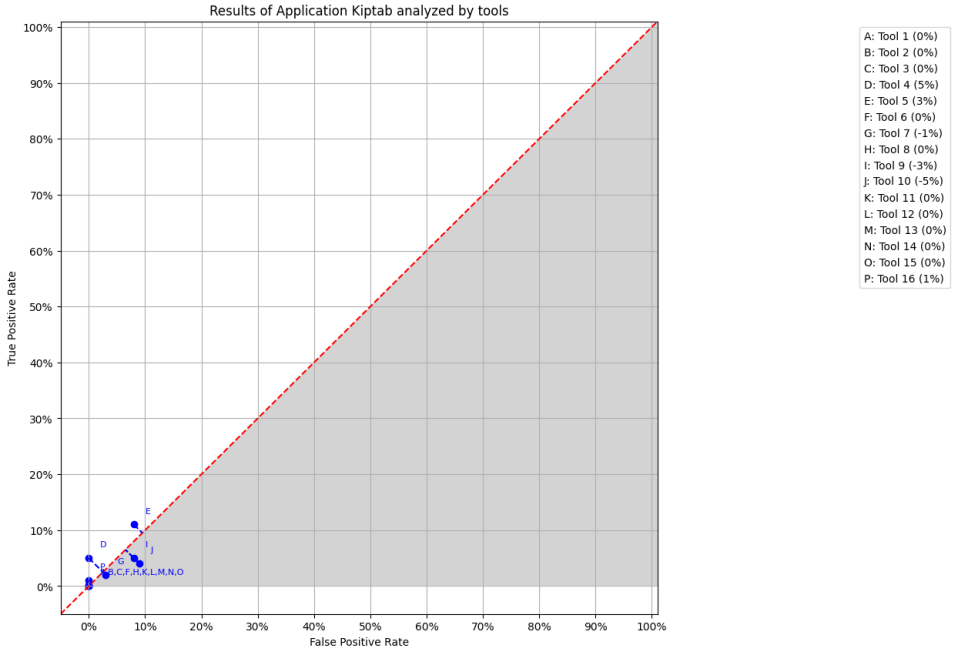


Figure 5.14: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.14: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.0167 | 0.0 | 0.0167 | 1.67% |
| **Tool 2** | 0.003 | 0.0 | 0.003 | 0.3% |
| **Tool 3** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 4** | 0.0457 | 0.0 | 0.0457 | 4.57% |
| **Tool 5** | 0.1053 | 0.0833 | 0.0219 | 2.19% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.0169 | 0.0333 | -0.0165 | -1.65% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.0508 | 0.0833 | -0.0326 | -3.26% |
| **Tool 10** | 0.0384 | 0.0917 | -0.0533 | -5.33% |
| **Tool 11** | 0.0015 | 0.0 | 0.0015 | 0.15% |
| **Tool 12** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 13** | 0.0015 | 0.0 | 0.0015 | 0.15% |
| **Tool 14** | 0.0045 | 0.0 | 0.0045 | 0.45% |
| **Tool 15** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 16** | 0.0139 | 0.0 | 0.0139 | 1.39% |

### 5.2.15 Events-manager-io

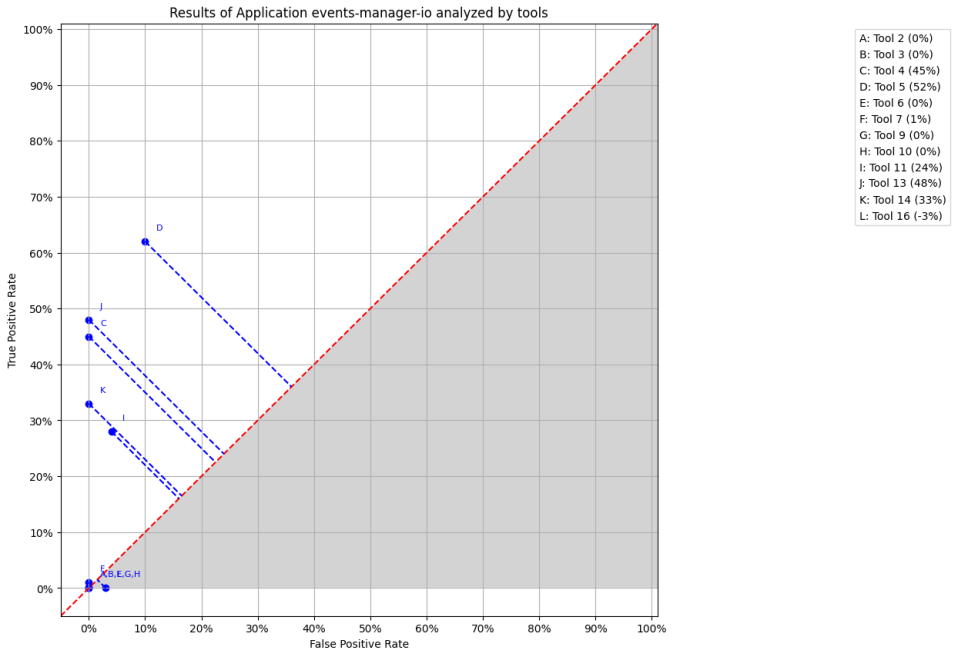


Figure 5.15: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS). Four tools are not taken into consideration because they do not support the Javascript language, which is used exclusively in the application in question. The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR.

Table 5.15: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 2** | 0.0015 | 0.0 | 0.0015 | 0.15% |
| **Tool 3** | 0.001 | 0.0 | 0.001 | 0.1% |
| **Tool 4** | 0.4495 | 0.0 | 0.4495 | 44.95% |
| **Tool 5** | 0.6162 | 0.0972 | 0.519 | 51.9% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.0094 | 0.0 | 0.0094 | 0.94% |
| **Tool 9** | 0.001 | 0.0 | 0.001 | 0.1% |
| **Tool 10** | 0.0005 | 0.0 | 0.0005 | 0.05% |
| **Tool 11** | 0.2841 | 0.0417 | 0.2425 | 24.25% |
| **Tool 13** | 0.4845 | 0.0 | 0.4845 | 48.45% |
| **Tool 14** | 0.3333 | 0.0 | 0.3333 | 33.33% |
| **Tool 16** | 0.003 | 0.0278 | -0.0248 | -2.48% |

### 5.2.16 $\mu$uCrypt

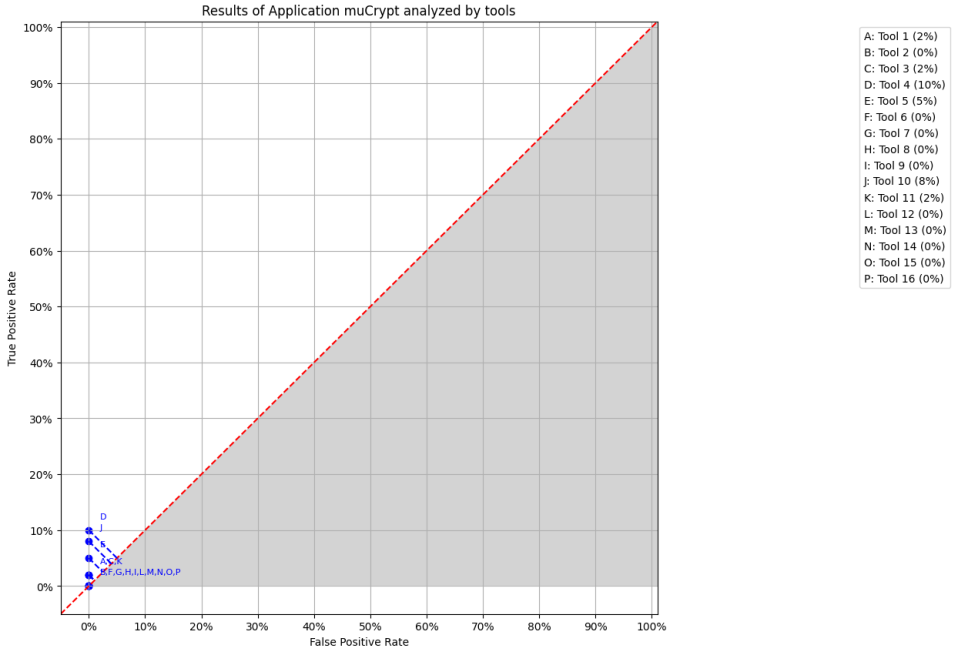


Figure 5.16: Results of application analyzed by the tools

In the following table, there is the description for every tool the TPR, the FPR and the Benchmark Accuracy Score (BAS).

Table 5.16: Results of tools on the application

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 1** | 0.04167 | 0.0 | 0.04167 | 4.17% |
| **Tool 2** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 3** | 0.0167 | 0.0 | 0.0167 | 1.67% |
| **Tool 4** | 0.1 | 0.0 | 0.1 | 10.0% |
| **Tool 5** | 0.05 | 0.0 | 0.05 | 5.0% |
| **Tool 6** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 7** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 8** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 9** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 10** | 0.0833 | 0.0 | 0.0833 | 8.33% |
| **Tool 11** | 0.0167 | 0.0 | 0.0167 | 1.67% |
| **Tool 12** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 13** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 14** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 15** | 0.0 | 0.0 | 0.0 | 0.0% |
| **Tool 16** | 0.0 | 0.0 | 0.0 | 0.0% |

## 5.3 Interpretation of Results

Looking at the results it is possible to observe which tool behaves best and which is worse. Table 5.17 collects the results for all target applications and calculates a cumulative score for each tool. This was obtained by calculating the TPR, the FPR and the score on the set of all target applications. This table also shows a ranking of the various tools considered.

Table 5.17: Results of the tools on all Applications

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 11** | 0.555 | 0.013 | 0.542 | **54.2** |
| **Tool 4** | 0.527 | 0 | 0.527 | **52.7** |
| **Tool 14** | 0.467 | 0 | 0.467 | **46.7** |
| **Tool 13** | 0.443 | 0 | 0.443 | **44.3** |
| **Tool 5** | 0.602 | 0.445 | 0.157 | **15.7** |
| **Tool 16** | 0.054 | 0.024 | 0.03 | **3** |
| **Tool 3** | 0.007 | 0 | 0.007 | **0.7** |
| **Tool 8** | 0 | 0 | 0 | **0** |
| **Tool 7** | 0.075 | 0.076 | -0.001 | **-0.1** |
| **Tool 2** | 0.02 | 0.024 | -0.004 | **-0.4** |
| **Tool 6** | 0.003 | 0.022 | -0.019 | **-1.9** |
| **Tool 12** | 0.012 | 0.047 | -0.035 | **-3.5** |
| **Tool 10** | 0.037 | 0.29 | -0.253 | **-25.3** |
| **Tool 9** | 0.06 | 0.359 | -0.299 | **-29.9** |
| **Tool 1** | 0.067 | 0.468 | -0.401 | **-40.1** |
| **Tool 15** | 0.012 | 0.637 | -0.625 | **-62.5** |

The score column contains several negative values. Negative values indicate that the tool is reporting many FPs and has an FPR that is higher than the TPR. The category "Alternate categories" in our results, as described in the evaluation methodology, is where vulnerabilities that don't belong to the previously chosen categories are entered. Numerous categories of vulnerabilities are included in this category, which greatly influences the outcomes of the various tools. It is crucial to have this category in place to research and analyse all of the vulnerabilities discovered by the various tools. The selection of the applications that make up the benchmark has a significant impact on the values in this table. From the point of view of the targets, it is interesting to note that none of the vulnerabilities reported is recognized by all tools. Still, from the standpoint of the web applications, we recall that the vulnerable targets were split into two groups: those that were entirely built in Javascript and those that had a Python server with a Javascript client. It's important to keep in mind from the perspective of tools that some tools perform better in Javascript and worse in Python. There are

alternative tools that operate more effectively in Python than in Javascript. Some tools can only be used with Python. Tool 1, Tool 8, Tool 12 and Tool 15 are the tools that work only in Python and only Python vulnerabilities are reported by these tools. Tool 6 and Tool 16 are the tools that work only in Javascript and only Javascript vulnerabilities are reported by these tools. It is important to analyze the performance of the tools on python and javascript separately. For this purpose, the following tables are used which show the rankings of the various tools based on the language used.

Table 5.18: Results of the tools considering only Python language

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 11** | 0.622 | 0 | 0.622 | **62.2** |
| **Tool 13** | 0.491 | 0 | 0.491 | **49.1** |
| **Tool 14** | 0.141 | 0 | 0.141 | **14.1** |
| **Tool 9** | 0.085 | 0 | 0.085 | **8.5** |
| **Tool 3** | 0.043 | 0 | 0.043 | **4.3** |
| **Tool 2** | 0.006 | 0 | 0.006 | **0.6** |
| **Tool 4** | 0 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 0 | **0** |
| **Tool 12** | 0.012 | 0.047 | -0.035 | **-3.5** |
| **Tool 5** | 0.313 | 0.548 | -0.235 | **-23.5** |
| **Tool 1** | 0.067 | 0.468 | -0.401 | **-40.1** |
| **Tool 10** | 0.067 | 0.605 | -0.538 | **-53.8** |
| **Tool 15** | 0.012 | 0.637 | -0.625 | **-62.5** |

Table 5.19: Results of the tools considering only Javascript language

| Tool | TPR | FPR | Score | Score (%) |
|---|---|---|---|---|
| **Tool 4** | 0.558 | 0 | 0.558 | **55.8** |
| **Tool 11** | 0.541 | 0.016 | 0.525 | **52.5** |
| **Tool 14** | 0.491 | 0 | 0.491 | **49.1** |
| **Tool 13** | 0.431 | 0 | 0.431 | **43.1** |
| **Tool 5** | 0.622 | 0.416 | 0.206 | **20.6** |
| **Tool 16** | 0.059 | 0.03 | 0.029 | **2.9** |
| **Tool 3** | 0.004 | 0 | 0.004 | **0.4** |
| **Tool 2** | 0.022 | 0.03 | -0.008 | **-0.8** |
| **Tool 7** | 0.081 | 0.096 | -0.015 | **-1.5** |
| **Tool 6** | 0.002 | 0.028 | -0.026 | **-2.6** |
| **Tool 10** | 0.034 | 0.191 | -0.157 | **-15.7** |
| **Tool 9** | 0.056 | 0.44 | -0.384 | **-38.4** |

Taking into account this categorization of applications into two groups,

it is simple to see that the vulnerabilities identified by the Python tools only exclusively refer to the server side. While the Javascript vulnerabilities listed refer to both sides (client e server). As a result, the number of vulnerabilities identified in Python is lower than the number of vulnerabilities identified in Javascript.

## 5.4 Method to compare SAST Tools

This section aims to classify the tools according to the metrics used to analyse the outcomes of the tool execution against the benchmark. The chosen metrics of precision, recall, false positives, and F-measure were then calculated. Recall metrics work well for highly critical applications where finding the most vulnerabilities is the main goal. When taking into account the false positive score, precision metric penalises the true positive ratio. Typically, a tool's true positive and false positive results are directly proportionate. This direct proportionality should be broken by an effective method. The ideal metric for choosing a tool that finds a lot of vulnerabilities while reporting a few false positives is called the F-measure. This enables very precise tools to produce far superior results. The measures listed in Section 5.1 are false positive rate (percentage), recall, precision, and F-measure are used to analyse the performance of tools against the benchmark. The following tables display the classification order based on F-measure scores (one per application).

Table 5.20: Metrics reported by the tools on Application 1

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 4** | 225 | 0 | 6 | 43 | 0.45 | 0 | 1 | 0.45 | **0.913** |
| **Tool 5** | 166 | 6 | 0 | 37 | 0.374 | 0.083 | 0.965 | 0.374 | **0.885** |
| **Tool 11** | 60 | 0 | 6 | 37 | 0.286 | 0 | 1 | 0.286 | **0.764** |
| **Tool 14** | 66 | 0 | 6 | 42 | 0.246 | 0 | 1 | 0.246 | **0.759** |
| **Tool 13** | 50 | 0 | 6 | 43 | 0.253 | 0 | 1 | 0.253 | **0.699** |
| **Tool 9** | 14 | 0 | 6 | 79 | 0.034 | 0 | 1 | 0.034 | **0.262** |
| **Tool 2** | 7 | 0 | 6 | 90 | 0.011 | 0 | 1 | 0.011 | **0.135** |
| **Tool 7** | 4 | 0 | 6 | 89 | 0.045 | 0 | 1 | 0.045 | **0.082** |
| **Tool 10** | 3 | 0 | 6 | 90 | 0.024 | 0 | 1 | 0.024 | **0.062** |
| **Tool 3** | 1 | 0 | 6 | 92 | 0.002 | 0 | 1 | 0.002 | **0.021** |
| **Tool 16** | 1 | 0 | 6 | 92 | 0.002 | 0 | 1 | 0.002 | **0.021** |
| **Tool 6** | 0 | 0 | 6 | 93 | 0 | 0 | 0 | 0 | **0** |

Table 5.21: Metrics reported by the tools on Application 2

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 14** | 7 | 0 | 1 | 8 | 0.083 | 0 | 1 | 0.083 | **0.636** |
| **Tool 13** | 3 | 0 | 1 | 11 | 0.083 | 0 | 1 | 0.083 | **0.353** |
| **Tool 1** | 1 | 0 | 0 | 5 | 0.042 | 0 | 1 | 0.042 | **0.286** |
| **Tool 10** | 2 | 0 | 1 | 12 | 0.083 | 0 | 1 | 0.083 | **0.25** |
| **Tool 11** | 2 | 0 | 1 | 12 | 0.042 | 0 | 1 | 0.042 | **0.25** |
| **Tool 5** | 2 | 1 | 0 | 12 | 0.062 | 0.083 | 0.667 | 0.062 | **0.235** |
| **Tool 3** | 1 | 0 | 1 | 13 | 0.042 | 0 | 1 | 0.042 | **0.133** |
| **Tool 2** | 0 | 0 | 1 | 14 | 0 | 0 | 0 | 0 | **0** |
| **Tool 4** | 0 | 0 | 1 | 14 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 0 | 1 | 14 | 0 | 0 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 1 | 14 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | **0** |
| **Tool 9** | 0 | 0 | 1 | 14 | 0 | 0 | 0 | 0 | **0** |
| **Tool 12** | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | **0** |
| **Tool 15** | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | **0** |
| **Tool 16** | 0 | 0 | 1 | 14 | 0 | 0 | 0 | 0 | **0** |

Table 5.22: Metrics reported by the tools on Application 3

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 39 | 0 | 21 | 17 | 0.203 | 0 | 1 | 0.203 | **0.821** |
| **Tool 14** | 35 | 0 | 21 | 22 | 0.162 | 0 | 1 | 0.162 | **0.761** |
| **Tool 5** | 35 | 5 | 16 | 19 | 0.217 | 0.049 | 0.875 | 0.217 | **0.745** |
| **Tool 13** | 25 | 0 | 21 | 23 | 0.131 | 0 | 1 | 0.131 | **0.685** |
| **Tool 4** | 28 | 0 | 21 | 28 | 0.147 | 0 | 1 | 0.147 | **0.667** |
| **Tool 10** | 9 | 4 | 17 | 42 | 0.093 | 0.042 | 0.692 | 0.093 | **0.281** |
| **Tool 9** | 3 | 10 | 11 | 45 | 0.027 | 0.076 | 0.231 | 0.027 | **0.098** |
| **Tool 7** | 1 | 5 | 16 | 47 | 0.012 | 0.083 | 0.167 | 0.012 | **0.037** |
| **Tool 16** | 1 | 5 | 19 | 47 | 0.012 | 0.072 | 0.167 | 0.012 | **0.037** |
| **Tool 2** | 0 | 0 | 21 | 48 | 0 | 0 | 0 | 0 | **0** |
| **Tool 3** | 0 | 0 | 21 | 48 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 0 | 21 | 48 | 0 | 0 | 0 | 0 | **0** |

Table 5.23: Metrics reported by the tools on Application 4

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 3 | 0 | 93 | 9 | 0.042 | 0 | 1 | 0.042 | **0.4** |
| **Tool 14** | 3 | 0 | 93 | 9 | 0.042 | 0 | 1 | 0.042 | **0.4** |
| **Tool 1** | 1 | 0 | 20 | 3 | 0.021 | 0 | 1 | 0.021 | **0.4** |
| **Tool 5** | 2 | 4 | 89 | 10 | 0.028 | 0.068 | 0.333 | 0.028 | **0.222** |
| **Tool 16** | 1 | 0 | 93 | 11 | 0.014 | 0 | 1 | 0.014 | **0.154** |
| **Tool 15** | 1 | 11 | 9 | 3 | 0.021 | 0.083 | 0.083 | 0.021 | **0.125** |
| **Tool 9** | 2 | 29 | 64 | 10 | 0.028 | 0.043 | 0.065 | 0.028 | **0.093** |
| **Tool 2** | 0 | 1 | 92 | 12 | 0 | 0.001 | 0 | 0 | **0** |
| **Tool 3** | 0 | 0 | 93 | 12 | 0 | 0 | 0 | 0 | **0** |
| **Tool 4** | 0 | 0 | 93 | 12 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 2 | 91 | 12 | 0 | 0.003 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 93 | 12 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 20 | 4 | 0 | 0 | 0 | 0 | **0** |
| **Tool 10** | 0 | 52 | 45 | 12 | 0 | 0.22 | 0 | 0 | **0** |
| **Tool 12** | 0 | 5 | 18 | 4 | 0 | 0.111 | 0 | 0 | **0** |
| **Tool 13** | 0 | 0 | 93 | 12 | 0 | 0 | 0 | 0 | **0** |

Table 5.24: Metrics reported by the tools on Application 5

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 146 | 2 | 118 | 57 | 0.419 | 0.084 | 0.986 | 0.419 | **0.832** |
| **Tool 16** | 54 | 0 | 115 | 123 | 0.152 | 0 | 1 | 0.152 | **0.468** |
| **Tool 14** | 45 | 0 | 120 | 121 | 0.306 | 0 | 1 | 0.306 | **0.427** |
| **Tool 7** | 50 | 28 | 91 | 125 | 0.131 | 0.02 | 0.641 | 0.131 | **0.395** |
| **Tool 13** | 30 | 0 | 120 | 135 | 0.24 | 0 | 1 | 0.24 | **0.308** |
| **Tool 10** | 16 | 1 | 119 | 149 | 0.162 | 0.083 | 0.941 | 0.162 | **0.176** |
| **Tool 9** | 16 | 18 | 102 | 150 | 0.018 | 0.013 | 0.471 | 0.018 | **0.16** |
| **Tool 4** | 10 | 0 | 120 | 155 | 0.012 | 0 | 1 | 0.012 | **0.114** |
| **Tool 5** | 6 | 70 | 50 | 159 | 0.007 | 0.132 | 0.079 | 0.007 | **0.05** |
| **Tool 6** | 2 | 1 | 119 | 163 | 0.002 | 0.001 | 0.667 | 0.002 | **0.024** |
| **Tool 3** | 1 | 0 | 120 | 164 | 0.017 | 0 | 1 | 0.017 | **0.012** |
| **Tool 2** | 0 | 1 | 119 | 165 | 0 | 0.001 | 0 | 0 | **0** |

Table 5.25: Metrics reported by the tools on Application 6

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 13** | 83 | 0 | 84 | 55 | 0.378 | 0 | 1 | 0.378 | **0.751** |
| **Tool 11** | 75 | 0 | 84 | 63 | 0.261 | 0 | 1 | 0.261 | **0.704** |
| **Tool 14** | 72 | 0 | 84 | 89 | 0.515 | 0 | 1 | 0.515 | **0.618** |
| **Tool 5** | 15 | 56 | 28 | 123 | 0.052 | 0.1 | 0.211 | 0.052 | **0.144** |
| **Tool 1** | 4 | 50 | 16 | 63 | 0.088 | 0.083 | 0.074 | 0.088 | **0.066** |
| **Tool 3** | 4 | 0 | 84 | 134 | 0.044 | 0 | 1 | 0.044 | **0.056** |
| **Tool 4** | 4 | 0 | 84 | 134 | 0.027 | 0 | 1 | 0.027 | **0.056** |
| **Tool 12** | 2 | 1 | 65 | 66 | 0.003 | 0.007 | 0.667 | 0.003 | **0.056** |
| **Tool 6** | 2 | 1 | 83 | 136 | 0.002 | 0.003 | 0.667 | 0.002 | **0.028** |
| **Tool 9** | 2 | 16 | 68 | 136 | 0.002 | 0.044 | 0.111 | 0.002 | **0.026** |
| **Tool 10** | 2 | 50 | 35 | 136 | 0.042 | 0.25 | 0.038 | 0.042 | **0.021** |
| **Tool 2** | 0 | 0 | 84 | 138 | 0 | 0 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 84 | 138 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 66 | 67 | 0 | 0 | 0 | 0 | **0** |
| **Tool 15** | 0 | 56 | 10 | 67 | 0 | 0.125 | 0 | 0 | **0** |
| **Tool 16** | 0 | 0 | 84 | 138 | 0 | 0 | 0 | 0 | **0** |

Table 5.26: Metrics reported by the tools on Application 7

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 19 | 0 | 46 | 34 | 0.035 | 0 | 1 | 0.035 | **0.528** |
| **Tool 7** | 9 | 0 | 47 | 43 | 0.017 | 0 | 1 | 0.017 | **0.295** |
| **Tool 13** | 9 | 0 | 47 | 44 | 0.017 | 0 | 1 | 0.017 | **0.29** |
| **Tool 2** | 6 | 7 | 41 | 44 | 0.052 | 0.015 | 0.462 | 0.052 | **0.19** |
| **Tool 9** | 7 | 15 | 32 | 45 | 0.083 | 0.1 | 0.318 | 0.083 | **0.189** |
| **Tool 14** | 5 | 0 | 47 | 47 | 0.009 | 0 | 1 | 0.009 | **0.175** |
| **Tool 5** | 7 | 26 | 21 | 45 | 0.013 | 0.137 | 0.212 | 0.013 | **0.165** |
| **Tool 4** | 2 | 0 | 47 | 50 | 0.004 | 0 | 1 | 0.004 | **0.074** |
| **Tool 16** | 2 | 0 | 46 | 50 | 0.004 | 0 | 1 | 0.004 | **0.074** |
| **Tool 10** | 2 | 2 | 45 | 50 | 0.085 | 0.085 | 0.5 | 0.085 | **0.071** |
| **Tool 3** | 0 | 0 | 47 | 52 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 1 | 46 | 52 | 0 | 0.002 | 0 | 0 | **0** |

Table 5.27: Metrics reported by the tools on Application 8

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 5** | 231 | 6 | 0 | 52 | 0.49 | 0.083 | 0.975 | 0.49 | **0.888** |
| **Tool 11** | 135 | 0 | 4 | 35 | 0.47 | 0 | 1 | 0.47 | **0.885** |
| **Tool 14** | 144 | 0 | 4 | 42 | 0.488 | 0 | 1 | 0.488 | **0.873** |
| **Tool 4** | 158 | 0 | 4 | 86 | 0.355 | 0 | 1 | 0.355 | **0.786** |
| **Tool 13** | 101 | 0 | 4 | 63 | 0.364 | 0 | 1 | 0.364 | **0.762** |
| **Tool 1** | 1 | 0 | 2 | 9 | 0.008 | 0 | 1 | 0.008 | **0.182** |
| **Tool 3** | 1 | 0 | 4 | 163 | 0.001 | 0 | 1 | 0.001 | **0.012** |
| **Tool 10** | 1 | 0 | 3 | 163 | 0.001 | 0 | 1 | 0.001 | **0.012** |
| **Tool 2** | 0 | 0 | 4 | 164 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 0 | 4 | 164 | 0 | 0 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 4 | 164 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 2 | 10 | 0 | 0 | 0 | 0 | **0** |
| **Tool 9** | 0 | 0 | 4 | 164 | 0 | 0 | 0 | 0 | **0** |
| **Tool 12** | 0 | 0 | 2 | 10 | 0 | 0 | 0 | 0 | **0** |
| **Tool 15** | 0 | 0 | 2 | 10 | 0 | 0 | 0 | 0 | **0** |
| **Tool 16** | 0 | 0 | 4 | 164 | 0 | 0 | 0 | 0 | **0** |

Table 5.28: Metrics reported by the tools on Application 9

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 5** | 230 | 10 | 14 | 72 | 0.392 | 0.117 | 0.958 | 0.392 | **0.849** |
| **Tool 4** | 215 | 0 | 25 | 108 | 0.43 | 0 | 1 | 0.43 | **0.799** |
| **Tool 11** | 153 | 2 | 23 | 104 | 0.343 | 0.007 | 0.987 | 0.343 | **0.743** |
| **Tool 13** | 111 | 0 | 25 | 90 | 0.404 | 0 | 1 | 0.404 | **0.712** |
| **Tool 14** | 99 | 0 | 25 | 116 | 0.268 | 0 | 1 | 0.268 | **0.631** |
| **Tool 16** | 7 | 2 | 23 | 193 | 0.027 | 0.007 | 0.778 | 0.027 | **0.067** |
| **Tool 2** | 5 | 4 | 22 | 196 | 0.003 | 0.014 | 0.556 | 0.003 | **0.048** |
| **Tool 9** | 4 | 11 | 14 | 197 | 0.004 | 0.12 | 0.267 | 0.004 | **0.037** |
| **Tool 7** | 3 | 3 | 22 | 197 | 0.035 | 0.011 | 0.5 | 0.035 | **0.029** |
| **Tool 10** | 2 | 6 | 19 | 198 | 0.015 | 0.101 | 0.25 | 0.015 | **0.019** |
| **Tool 3** | 0 | 0 | 25 | 201 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 4 | 21 | 201 | 0 | 0.014 | 0 | 0 | **0** |

Table 5.29: Metrics reported by the tools on Application 10

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 68 | 0 | 41 | 41 | 0.364 | 0 | 1 | 0.364 | **0.768** |
| **Tool 14** | 57 | 0 | 41 | 58 | 0.225 | 0 | 1 | 0.225 | **0.663** |
| **Tool 13** | 52 | 0 | 41 | 57 | 0.312 | 0 | 1 | 0.312 | **0.646** |
| **Tool 5** | 31 | 11 | 30 | 78 | 0.163 | 0.045 | 0.738 | 0.163 | **0.411** |
| **Tool 9** | 10 | 0 | 41 | 100 | 0.013 | 0 | 1 | 0.013 | **0.167** |
| **Tool 10** | 7 | 26 | 15 | 102 | 0.034 | 0.105 | 0.212 | 0.034 | **0.099** |
| **Tool 4** | 3 | 0 | 41 | 106 | 0.007 | 0 | 1 | 0.007 | **0.054** |
| **Tool 1** | 1 | 8 | 26 | 60 | 0.002 | 0.033 | 0.111 | 0.002 | **0.029** |
| **Tool 15** | 1 | 12 | 22 | 60 | 0.002 | 0.05 | 0.077 | 0.002 | **0.027** |
| **Tool 2** | 1 | 0 | 41 | 108 | 0.001 | 0 | 1 | 0.001 | **0.018** |
| **Tool 3** | 0 | 0 | 41 | 109 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 0 | 41 | 109 | 0 | 0 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 41 | 109 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 34 | 61 | 0 | 0 | 0 | 0 | **0** |
| **Tool 12** | 0 | 0 | 34 | 61 | 0 | 0 | 0 | 0 | **0** |
| **Tool 16** | 0 | 0 | 41 | 109 | 0 | 0 | 0 | 0 | **0** |

Table 5.30: Metrics reported by the tools on Application 11

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 105 | 0 | 11 | 49 | 0.386 | 0 | 1 | 0.386 | **0.811** |
| **Tool 14** | 108 | 0 | 11 | 55 | 0.343 | 0 | 1 | 0.343 | **0.797** |
| **Tool 5** | 141 | 11 | 5 | 73 | 0.345 | 0.146 | 0.928 | 0.345 | **0.77** |
| **Tool 4** | 89 | 0 | 11 | 94 | 0.24 | 0 | 1 | 0.24 | **0.654** |
| **Tool 13** | 65 | 0 | 11 | 80 | 0.307 | 0 | 1 | 0.307 | **0.619** |
| **Tool 7** | 23 | 3 | 8 | 128 | 0.018 | 0.083 | 0.885 | 0.018 | **0.26** |
| **Tool 16** | 17 | 3 | 8 | 128 | 0.014 | 0.083 | 0.85 | 0.014 | **0.206** |
| **Tool 2** | 9 | 0 | 11 | 138 | 0.007 | 0 | 1 | 0.007 | **0.115** |
| **Tool 10** | 5 | 7 | 4 | 140 | 0.02 | 0.139 | 0.417 | 0.02 | **0.064** |
| **Tool 9** | 1 | 2 | 9 | 144 | 0.001 | 0.056 | 0.333 | 0.001 | **0.014** |
| **Tool 3** | 0 | 0 | 11 | 145 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 0 | 11 | 145 | 0 | 0 | 0 | 0 | **0** |

Table 5.31: Metrics reported by the tools on Application 12

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 9** | 15 | 0 | 4 | 13 | 0.1 | 0 | 1 | 0.1 | **0.698** |
| **Tool 5** | 9 | 4 | 1 | 8 | 0.05 | 0.139 | 0.692 | 0.05 | **0.6** |
| **Tool 1** | 1 | 0 | 1 | 2 | 0.028 | 0 | 1 | 0.028 | **0.5** |
| **Tool 4** | 2 | 0 | 4 | 15 | 0.011 | 0 | 1 | 0.011 | **0.211** |
| **Tool 7** | 1 | 0 | 4 | 16 | 0.006 | 0 | 1 | 0.006 | **0.111** |
| **Tool 11** | 1 | 0 | 4 | 16 | 0.006 | 0 | 1 | 0.006 | **0.111** |
| **Tool 13** | 1 | 0 | 4 | 16 | 0.006 | 0 | 1 | 0.006 | **0.111** |
| **Tool 6** | 1 | 1 | 3 | 16 | 0.006 | 0.028 | 0.5 | 0.006 | **0.105** |
| **Tool 10** | 1 | 1 | 3 | 16 | 0.083 | 0.028 | 0.5 | 0.083 | **0.105** |
| **Tool 2** | 0 | 0 | 4 | 17 | 0 | 0 | 0 | 0 | **0** |
| **Tool 3** | 0 | 0 | 4 | 17 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | **0** |
| **Tool 12** | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | **0** |
| **Tool 14** | 0 | 0 | 4 | 17 | 0 | 0 | 0 | 0 | **0** |
| **Tool 15** | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | **0** |
| **Tool 16** | 0 | 0 | 4 | 17 | 0 | 0 | 0 | 0 | **0** |

Table 5.32: Metrics reported by the tools on Application 13

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 5** | 305 | 24 | 28 | 107 | 0.571 | 0.13 | 0.927 | 0.571 | **0.823** |
| **Tool 4** | 220 | 0 | 45 | 162 | 0.462 | 0 | 1 | 0.462 | **0.731** |
| **Tool 11** | 161 | 0 | 45 | 128 | 0.385 | 0 | 1 | 0.385 | **0.716** |
| **Tool 14** | 163 | 0 | 45 | 141 | 0.472 | 0 | 1 | 0.472 | **0.698** |
| **Tool 13** | 134 | 0 | 45 | 140 | 0.519 | 0 | 1 | 0.519 | **0.657** |
| **Tool 7** | 20 | 0 | 45 | 254 | 0.012 | 0 | 1 | 0.012 | **0.136** |
| **Tool 9** | 21 | 97 | 14 | 263 | 0.013 | 0.145 | 0.178 | 0.013 | **0.104** |
| **Tool 16** | 6 | 1 | 44 | 268 | 0.005 | 0.004 | 0.857 | 0.005 | **0.043** |
| **Tool 10** | 5 | 5 | 40 | 270 | 0.029 | 0.032 | 0.5 | 0.029 | **0.035** |
| **Tool 2** | 3 | 0 | 45 | 271 | 0.001 | 0 | 1 | 0.001 | **0.022** |
| **Tool 3** | 2 | 0 | 45 | 272 | 0.001 | 0 | 1 | 0.001 | **0.014** |
| **Tool 6** | 0 | 2 | 43 | 274 | 0 | 0.009 | 0 | 0 | **0** |

Table 5.33: Metrics reported by the tools on Application 14

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 4** | 22 | 0 | 27 | 44 | 0.046 | 0 | 1 | 0.046 | **0.5** |
| **Tool 5** | 16 | 3 | 25 | 52 | 0.105 | 0.083 | 0.842 | 0.105 | **0.368** |
| **Tool 10** | 9 | 4 | 23 | 57 | 0.038 | 0.092 | 0.692 | 0.038 | **0.228** |
| **Tool 1** | 1 | 0 | 1 | 8 | 0.017 | 0 | 1 | 0.017 | **0.2** |
| **Tool 9** | 9 | 20 | 7 | 57 | 0.051 | 0.083 | 0.31 | 0.051 | **0.189** |
| **Tool 14** | 3 | 0 | 27 | 63 | 0.005 | 0 | 1 | 0.005 | **0.087** |
| **Tool 7** | 3 | 2 | 25 | 64 | 0.017 | 0.033 | 0.6 | 0.017 | **0.083** |
| **Tool 2** | 2 | 0 | 27 | 64 | 0.003 | 0 | 1 | 0.003 | **0.059** |
| **Tool 16** | 1 | 0 | 27 | 65 | 0.014 | 0 | 1 | 0.014 | **0.03** |
| **Tool 11** | 1 | 0 | 27 | 65 | 0.002 | 0 | 1 | 0.002 | **0.03** |
| **Tool 13** | 1 | 0 | 27 | 65 | 0.002 | 0 | 1 | 0.002 | **0.03** |
| **Tool 3** | 0 | 0 | 27 | 66 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 0 | 27 | 66 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 1 | 9 | 0 | 0 | 0 | 0 | **0** |
| **Tool 12** | 0 | 0 | 1 | 9 | 0 | 0 | 0 | 0 | **0** |
| **Tool 15** | 0 | 0 | 1 | 9 | 0 | 0 | 0 | 0 | **0** |

Table 5.34: Metrics reported by the tools on Application 15

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 4** | 320 | 0 | 12 | 113 | 0.45 | 0 | 1 | 0.45 | **0.85** |
| **Tool 5** | 249 | 13 | 5 | 107 | 0.616 | 0.097 | 0.95 | 0.616 | **0.806** |
| **Tool 13** | 114 | 0 | 12 | 139 | 0.484 | 0 | 1 | 0.484 | **0.621** |
| **Tool 11** | 87 | 3 | 9 | 175 | 0.284 | 0.042 | 0.967 | 0.284 | **0.494** |
| **Tool 14** | 90 | 0 | 12 | 185 | 0.333 | 0 | 1 | 0.333 | **0.493** |
| **Tool 7** | 19 | 0 | 12 | 234 | 0.009 | 0 | 1 | 0.009 | **0.14** |
| **Tool 16** | 6 | 2 | 10 | 247 | 0.003 | 0.028 | 0.75 | 0.003 | **0.046** |
| **Tool 2** | 3 | 0 | 12 | 251 | 0.001 | 0 | 1 | 0.001 | **0.023** |
| **Tool 3** | 2 | 0 | 12 | 251 | 0.001 | 0 | 1 | 0.001 | **0.016** |
| **Tool 9** | 2 | 0 | 12 | 251 | 0.001 | 0 | 1 | 0.001 | **0.016** |
| **Tool 10** | 1 | 0 | 12 | 252 | 0 | 0 | 1 | 0 | **0.008** |
| **Tool 6** | 0 | 0 | 12 | 253 | 0 | 0 | 0 | 0 | **0** |

Table 5.35: Metrics reported by the tools on Application 16

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 5** | 3 | 0 | 0 | 4 | 0.05 | 0 | 1 | 0.05 | **0.6** |
| **Tool 1** | 1 | 0 | 0 | 2 | 0.042 | 0 | 1 | 0.042 | **0.5** |
| **Tool 4** | 2 | 0 | 0 | 5 | 0.1 | 0 | 1 | 0.1 | **0.444** |
| **Tool 10** | 1 | 0 | 0 | 6 | 0.083 | 0 | 1 | 0.083 | **0.25** |
| **Tool 3** | 1 | 0 | 0 | 6 | 0.017 | 0 | 1 | 0.017 | **0.25** |
| **Tool 11** | 1 | 0 | 0 | 6 | 0.017 | 0 | 1 | 0.017 | **0.25** |
| **Tool 2** | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | **0** |
| **Tool 6** | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | **0** |
| **Tool 9** | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | **0** |
| **Tool 12** | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | **0** |
| **Tool 13** | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | **0** |
| **Tool 14** | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | **0** |
| **Tool 15** | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | **0** |
| **Tool 16** | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | **0** |

After analyzing the various metrics for each application, the following table collects the metrics for all target applications by calculating a cumulative score for each tool. This was achieved by calculating the various parameters on all the various applications. This table also shows a ranking of the various tools considered.

Table 5.36: Metrics reported by the tools on all applications

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 1056 | 7 | 533 | 848 | 0.555 | 0.013 | 1 | 0.555 | **0.712** |
| **Tool 5** | 1448 | 250 | 312 | 958 | 0.602 | 0.445 | 1 | 0.602 | **0.706** |
| **Tool 4** | 1300 | 0 | 541 | 1169 | 0.527 | 0 | 1 | 0.527 | **0.69** |
| **Tool 14** | 897 | 0 | 541 | 1022 | 0.467 | 0 | 1 | 0.467 | **0.637** |
| **Tool 13** | 779 | 0 | 541 | 980 | 0.443 | 0 | 1 | 0.443 | **0.614** |
| **Tool 7** | 133 | 41 | 499 | 1641 | 0.075 | 0.076 | 1 | 0.075 | **0.137** |
| **Tool 16** | 96 | 13 | 525 | 1673 | 0.054 | 0.024 | 1 | 0.054 | **0.102** |
| **Tool 9** | 106 | 218 | 389 | 1675 | 0.06 | 0.359 | 0 | 0.06 | **0.101** |
| **Tool 10** | 66 | 158 | 387 | 1695 | 0.037 | 0.29 | 0 | 0.037 | **0.066** |
| **Tool 2** | 36 | 13 | 530 | 1727 | 0.02 | 0.024 | 1 | 0.02 | **0.04** |
| **Tool 1** | 11 | 58 | 196 | 516 | 0.021 | 0.228 | 0 | 0.021 | **0.037** |
| **Tool 3** | 13 | 0 | 541 | 1745 | 0.007 | 0 | 1 | 0.007 | **0.015** |
| **Tool 12** | 2 | 6 | 251 | 526 | 0.004 | 0.023 | 0 | 0.004 | **0.007** |
| **Tool 15** | 2 | 79 | 175 | 525 | 0.004 | 0.311 | 0 | 0.004 | **0.007** |
| **Tool 6** | 5 | 12 | 529 | 1753 | 0.003 | 0.022 | 0 | 0.003 | **0.006** |
| **Tool 8** | 0 | 0 | 254 | 527 | 0 | 0 | 0 | 0 | **0** |

It is important to analyze the performance of the tools on python and javascript separately. For this purpose, the following tables are used which show the rankings of the various tools based on the language used.

Table 5.37: Metrics reported by the tools on all applications considering only Python Language

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 11** | 102 | 0 | 124 | 62 | 0.622 | 0 | 1 | 0.622 | **0.767** |
| **Tool 13** | 80 | 0 | 124 | 83 | 0.491 | 0 | 1 | 0.491 | **0.658** |
| **Tool 5** | 51 | 68 | 56 | 112 | 0.313 | 0.548 | 0 | 0.313 | **0.362** |
| **Tool 14** | 23 | 0 | 124 | 140 | 0.141 | 0 | 1 | 0.141 | **0.247** |
| **Tool 9** | 14 | 0 | 124 | 150 | 0.085 | 0 | 1 | 0.085 | **0.157** |
| **Tool 1** | 11 | 58 | 66 | 152 | 0.067 | 0.468 | 0 | 0.067 | **0.095** |
| **Tool 10** | 11 | 75 | 49 | 152 | 0.067 | 0.605 | 0 | 0.067 | **0.088** |
| **Tool 3** | 7 | 0 | 124 | 156 | 0.043 | 0 | 1 | 0.043 | **0.082** |
| **Tool 12** | 2 | 6 | 121 | 162 | 0.012 | 0.047 | 0 | 0.012 | **0.023** |
| **Tool 15** | 2 | 79 | 45 | 161 | 0.012 | 0.637 | 0 | 0.012 | **0.016** |
| **Tool 2** | 1 | 0 | 124 | 162 | 0.006 | 0 | 1 | 0.006 | **0.012** |
| **Tool 4** | 0 | 0 | 124 | 163 | 0 | 0 | 0 | 0 | **0** |
| **Tool 7** | 0 | 0 | 124 | 163 | 0 | 0 | 0 | 0 | **0** |
| **Tool 8** | 0 | 0 | 124 | 163 | 0 | 0 | 0 | 0 | **0** |

Table 5.38: Metrics reported by the tools on all applications considering only Javascript language

| Tool | TP | FP | TN | FN | TPR | FPR | Precision | Recall | F-Measure |
|---|---|---|---|---|---|---|---|---|---|
| **Tool 5** | 1397 | 182 | 256 | 849 | 0.622 | 0.416 | 1 | 0.622 | **0.73** |
| **Tool 4** | 1300 | 0 | 430 | 1030 | 0.558 | 0 | 1 | 0.558 | **0.716** |
| **Tool 11** | 954 | 7 | 422 | 810 | 0.541 | 0.016 | 1 | 0.541 | **0.7** |
| **Tool 14** | 874 | 0 | 430 | 906 | 0.491 | 0 | 1 | 0.491 | **0.659** |
| **Tool 13** | 699 | 0 | 430 | 921 | 0.431 | 0 | 1 | 0.431 | **0.603** |
| **Tool 7** | 133 | 41 | 388 | 1502 | 0.081 | 0.096 | 1 | 0.081 | **0.147** |
| **Tool 16** | 96 | 13 | 414 | 1534 | 0.059 | 0.03 | 1 | 0.059 | **0.11** |
| **Tool 9** | 92 | 218 | 278 | 1549 | 0.056 | 0.44 | 0 | 0.056 | **0.094** |
| **Tool 10** | 55 | 83 | 352 | 1567 | 0.034 | 0.191 | 0 | 0.034 | **0.062** |
| **Tool 2** | 35 | 13 | 419 | 1589 | 0.022 | 0.03 | 1 | 0.022 | **0.042** |
| **Tool 3** | 6 | 0 | 430 | 1613 | 0.004 | 0 | 1 | 0.004 | **0.007** |
| **Tool 6** | 4 | 12 | 418 | 1615 | 0.002 | 0.028 | 0 | 0.002 | **0.005** |

We have created two categories for the effectiveness of the technologies that focus on F-measure and Recall metrics. Recall metrics must be taken into consideration for web applications as they show a SAST tool's ability to find more vulnerabilities. This metric makes it possible to identify instruments that have the best TP ratio. However, we have chosen to utilise the F-measure statistic for web apps because it is the best effort and indicates that the goal is to find the truest positives while producing the fewest false positives. The results of ranking tools using the BAS metric are similar to ranking using the F-measure and recall. Higher BAS-scoring tools also have a higher F-measure and recall. Commercial SAST tools for Web Applications need to be assessed. By comparing and ranking the outcomes of the Web application SAST tools' execution using a new repeatable technique, the benchmark properties claim that the cost of the benchmark installation is justified. The benchmark is easily portable and produces comparable results when used with the same tool multiple times. Based on the OWASP Top Ten project, this benchmark illustrates the current state of security flaws for web applications and is based on genuine code with a variety of source entries and levels of complexity. Additionally, it enables you to scale them by increasing the variety of vulnerability categories and types. Finally, the benchmark may be quickly conducted with the SAST tools.

# Chapter 6

# Conclusions and Future Works

This thesis presented a study on SAST tools for analysing various Javascript and Python web applications. 16 SAST tools that were used to attack 16 distinct targets were the subject of this investigation. We evaluated the behaviour and efficiency of each tool by focusing on its capacity to identify vulnerabilities in the testing environment as we do not have access to the internals of all tools. First, we have offered some fundamental ideas that are required to comprehend this argument. For instance, the usage of a benchmark and the security of web technologies and why we examined these apps with the SAST tools. Static analysis is another idea that is presented, along with its use. Following this background, we talked about the valuation approach, which is based on the OWASP benchmarking. We specifically discussed how the benchmark suggested by OWASP operates and its features. We then talked about the factors we took into account while selecting the evaluation strategy to evaluate web applications. Then, we defined the applications selected for testing and data gathering regarding the tools. Because we had to choose between web apps that were entirely written in Javascript and those that were developed in Python on the server side and Javascript on the client side, the choice of targets was really important. After the choice of applications, we presented the tools that were used to carry out the analysis and metrics on the applications. The analysis of the results, which concludes the thesis, is both its most significant and central component. In this, initially, we presented the general methodology for tool-based application analysis. The approach for classifying vulnerabilities comes next. The experiment is then described, followed by the presentation of the results along with some interpretations of them. This study pointed out the significant performance differences between various scanners and the apps used. While some tools were not able to identify almost any vulnerability, others were able to identify a significant number of vulnerabilities in

each target web application. None tools identified all vulnerabilities because every tool works on a different level. Then, using particular measures, we compared the security scanners' tools (F-Measure and recall). The tools with high scores based on these two parameters have a large collection of web app vulnerabilities. Despite these facts, there are some advantages to the problems we looked at. It is in the interest of programming language maintainers to offer security solutions to businesses and end users generally as awareness of information security issues is always developing in both the software engineering community and industry management. The OWASP Top 10 Web Application Security Risks project has been revised for 2021 by OWASP. Such guidelines must be given properly and should not be undervalued, as developer community education is essential for the security and calibre of software. Software protection, like most other aspects of information security, is always improving as new methods for identifying and fixing vulnerabilities are developed. Automated and semi-automated procedures are already crucial in software validation for security as software codebases grow bigger and more complex every day. The pattern of these methodologies' growing significance provides clear guidance for research, which will continue to move from human to automated analysis. We frequently employed open-source software as well as freely available tools in our effort to choose tools that are indicative of what the market has to offer. There are also commercial tools used but are free for free open-source projects. Static analysis-based methods have a drawback when a vulnerability occurs during runtime. When an attacker, for instance, employs reflection or can dynamically load classes at runtime, static analysis may not be able to identify some attack vectors. However, this is more typically a limitation of static analysis. Some suggestions for potential extensions of the work arose while the thesis was developed. We report the following:

- Fixing the problems identified in the reports and rerunning all of the analyses to see how the metrics have changed.
- An addition to this work could be to repeat the same experiment but with different tools (including commercial tools).
- An addition to this work could be to repeat the same experiment but with a different programming language (e.g., PHP or C#).
- An addition to this work is to use an approach to design benchmarks for evaluating such SAST considering different levels of criticality.

# Bibliography


[1] As a service — SonarCloud — sonarsource.com. https://www.sonarsource.com/products/sonarcloud/.

[2] bandit: Bandit is a tool designed to find common security issues in Python code. — github.com. https://github.com/PyCQA/bandit.

[3] cinema-plus: Online movie ticket booking web app— github.com. https://github.com/georgesimos/cinema-plus.

[4] The "cloud" at home — github.com. https://github.com/antoniosarosi/home-cloud.

[5] Coverity Scan - Static Analysis — scan.coverity.com. https://scan.coverity.com/.

[6] DeepSource — The Modern Static Analysis Platform — deepsource.io. https://deepsource.io/.

[7] events-manager-io: A basic site for managing event centers and scheduling events. — github.com. https://github.com/appcypher/events-manager-io.

[8] excel-to-json: Excel to json online converter — github.com. https://github.com/filipefilardi/excel-to-json.

[9] Free for open source application security tools — owasp foundation — owasp.org. https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools.

[10] GitHub - insidersec/insider: Static Application Security Testing (SAST) — github.com. https://github.com/insidersec/insider.

[11] Google Code Archive - Long-term storage for Google Code Project Hosting. — code.google.com. https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[12] hack-chat: a minimal, distraction-free chat application — github.com. https://github.com/hack-chat/main.

[13] hellobooks: A single-page library management app — github.com. https://github.com/segunolalive/helloBooks.

[14] Horusec home — horusec.io. https://horusec.io/site/.

[15] Introduction — Fluid Attacks Documentation — docs.fluidattacks.com. https://docs.fluidattacks.com/machine/scanner.

[16] kiptab: Kiptab helps you and your friends keep track of expenses during vacations and other social settings by balancing debts automatically. — github.com. https://github.com/bnan/kiptab.

[17] Kompar — catalog.kompar.tools. https://catalog.kompar.tools/analyzers.

[18] LGTM - Code Analysis Platform to Find and Prevent Vulnerabilities — lgtm.com. https://lgtm.com/.

[19] Mend Free Tools For Developers — mend.io. https://www.mend.io/free-developer-tools/.

[20] mucrypt messenger: A secure and simple end-to-end encrypted chat — github.com. https://github.com/connor-brooks/muCrypt.

[21] N. n. i. of standards and technology. national vulnerability database nvd, 2022. https://nvd.nist.gov/.

[22] Nano-speedtest: Webapp to test speed of nano transactions — github.com. https://github.com/silverstar194/Nano-SpeedTest.

[23] onmyway: Web and mobile application that algorithmically generates a tourism route for a given city and interests. — github.com. https://github.com/bnan/onmyway.

[24] Overview — Pyre — pyre-check.org. https://pyre-check.org/docs/pysa-basics/.

[25] Owasp foundation. https://owasp.org/.

[26] Owasp foundation benchmark. https://owasp.org/www-project-benchmark/.

[27] Precision and recall - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Precision_and_recall.

[28] Reactsocial: Social media — github.com. https://github.com/FLiotta/ReactSocial.

[29] Reshift Security — A Source Code Security Tool for modern developers — reshiftsecurity.com. https://www.reshiftsecurity.com/.

[30] Semgrep — semgrep.dev. https://semgrep.dev/.

[31] Slack-clone: Full-stack live chat application that recreate slack's main features — github.com. https://github.com/yuchiu/Slack-Clone.

[32] Snyk — Developer security — Develop fast. Stay secure. — snyk.io. https://snyk.io/.

[33] Source code analysis tools — owasp foundation — owasp.org. https://owasp.org/www-community/Source_Code_Analysis_Tools.

[34] Stackoverflow-clone-backend: Backend code of the stackoverflow clone project. — github.com. https://github.com/Mayank0255/Stackoverflow-Clone-Backend.

[35] Stackoverflow-clone-frontend: Clone project of a famous Q/A website — github.com. https://github.com/Mayank0255/Stackoverflow-Clone-Frontend.

[36] Storekeeper: Multilingual warehouse/store management software — github.com. https://github.com/andras-tim/StoreKeeper.

[37] video-labeling-tool: A web-based tool for labeling video clips (both front-end and back-end). — github.com. https://github.com/CMU-CREATE-Lab/video-labeling-tool.

[38] websocket-chat: A proof-of-concept chat server/client in websockets — github.com. https://github.com/sjkingo/websocket-chat.

[39] Your Partner in Open Source — Debricked — debricked.com. https://debricked.com/.

[40] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[41] Juan R Bermejo Higuera, Javier Bermejo Higuera, Juan A Sicilia Montalvo, Javier Cubo Villalba, and Juan José Nombela Pérez. Benchmarking approach to compare web applications static analysis tools detecting owasp top ten security vulnerabilities. *Comput. Mater. Continua*, 64(3):1555–1577, 2020.

[42] Paulo Nunes, Ibéria Medeiros, José C Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018.

[43] Team Scan. Scan docs — slscan.io. https://slscan.io/en/latest/.

# Acknowledgements

This thesis would not have been possible without the support of many people who have contributed, with their support, to its realization.

A special thanks go to my supervisor Prof. Riccardo Sisto who followed me, with his infinite availability, in every step of the creation of the paper, right from the choice of the topic.

I am infinitely grateful to my parents who have always supported me, supporting my every decision, right from the choice of my course of study.

A big thank you with affection to my sister Lina and my brother Fernando who supported me at all times during my university career.

A heartfelt thanks to my colleague and roommate Filippo, with whom I shared my entire university career. It is thanks to him that I have overcome the most difficult moments. Without his advice, I would never have made it.

To my friends, my university mates and all those who have crossed their life with mine leaving me something good. Thank you for being my accomplices, each in his way, in this intense and exciting journey, for better or for worse. There are so many memories that go through my head that it is impossible to find the right words to honour them. Thank you for making my achievement truly special!

Finally, I dedicate this thesis to myself, to my sacrifices and the tenacity that allowed me to get here.