UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Karl Jääts

How well could have existing static vulnerability detection tools prevented publicly reported vulnerabilities in iOS open source packages?

Master's Thesis (30 ECTS)

Supervisor: Kristiina Rahkema, MSc

Tartu 15-Aug-2023

How well could have existing static vulnerability detection tools prevented publicly reported vulnerabilities in iOS open source packages?

Abstract:

Preventing vulnerabilities is an ever present and high risk issue in software development that can cause a lot of problems if vulnerabilities are not detected. To prevent vulnerabilities as much as possible many different techniques and approaches have been developed and one of those is vulnerability detection tools. Many such tools have been created but it is unclear how effective the approach is at preventing real world vulnerabilities. In this thesis testing was carried out on publicly reported vulnerabilities in iOS open source packages with the aim of finding out how many of these vulnerabilities could have been prevented by using these tools. Multiple types of security testing tools exist, such as static application security testing (SAST), dynamic security testing (DAST), manual testing and other hybrid approaches. In this thesis SAST tools are used due to their relative ease of use. 5 SAST tools were tested on 81 publicly reported vulnerabilities in 23 packages with 14 out of the 81 vulnerable code segments being flagged by at least one tool. However due to the way these vulnerabilities were reported and the prevalence of false positives it seems that these SAST tools are not good at pinpointing existing vulnerabilities. Instead they help prevent vulnerabilities by directing the developers to write better quality code and notifying them of functions and approaches that are difficult to implement safely so that they know to take extra care or find safer alternatives.

Keywords:

Open-source, vulnerability, CVE, Static Application Security Testing, SAST, vulnerability detection tools

CERCS: P170, Computer science, numerical analysis, systems, control

Kui hästi oleksid olemasolevad staatilised turvavigade tuvastus tööriistad suutnud ennetada avalikult raporteeritud turvavigu iOS avatud lähtekoodiga teekides?

Lühikokkuvõte:

Turvavigade ennetamine on pidev ja väga oluline osa tarkvara arendusest, sest turvavigade olemasolu võib tekitada mitmeid suuri probleeme. Turvavigade ennetamiseks on loodud mitmeid eri meetodeid ja lähenemisi ja üheks neist on turvavigade tuvastus tööriistad. Selliseid tööriistu eksisteerib üsna palju aga pole selge kui efektiivsed nad on päris maailmas eksisteerivate turvavigade ennetamises. Selle väljaselgitamiseks testiti selles lõputöös turvavigade tuvastus tööriistatu iOS avatud lähtekoodiga teekides olevate avalikult raporteeritud turvavigade peal. Selliseid tööriistu on mitut tüüpi, nagu staatiline turvatestimine (SAST), dünaamiline turvatestimine (DAST), manuaalne testimine ja muud hübriid lähenemised. Selles töös kasutatakse SAST tööriistu, eelkõige nende kasutus lihtsuse tõttu. Testiti 5 SAST tööriista 81 avalikult raporteeritud turvaveal, mis esinesid 23-s teegis. Testitud turvavigadest 14 esinesid vähemalt ühe tööriista tulemuste seas, aga nende tööriistade raporteerimisviisi ja valepositiivsete tulemuste rohkuse tõttu paistab, et SAST tööriistad ei ole tugevad olemasolevate turvavigade leidmises. SAST tööriistade kasu turvavigade ennetamisel seisneb pigem arendajate parema kvaliteediga koodi kirjutamisele suunamisel ja arendajate informeerimisel kohtadest ja lähenemistest, mida on keeruline turvaliselt teostada, nii et nad teaksid nendes kohtades olla tähelepanelikud või alternatiivseid lähenemisi otsida.

Võtmesõnad:

Lähtekoodtarkvara, turvanõrkus, CVE, Staatiline rakenduse turvatestimine, SAST, turvanõrkuste tuvastus tööriistad

CERCS: P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

Introduction

Security vulnerabilities are an important issue in software development as they can be exploited by attackers to take down the system that they appear in or in worse cases to execute attackers code or expose sensitive information. So great care needs to be taken to reduce the amount of vulnerabilities as much as possible but they are often very difficult to spot even when specifically looking for them. To help with that several different methods have been developed to try to detect these vulnerabilities early so they are found and fixed well before anyone can exploit them. These methods range from manual testing techniques to automatic testing tools. The testing tools, which this thesis will focus on, can roughly be divided into three groups: Dynamic Application Security Testing (DAST), which test a running program through its frontend or API, Static Application Security Testing (SAST), which scan the code itself and Interactive Application Security Testing (IAST), which observes the system while its running and being interacted with and also references the code if something is found.

1. Problem statement

Despite the many techniques developed for finding and preventing vulnerabilities there are still many of them in modern programs, so the aim of this thesis is to learn a bit more about the causes for this by looking at one method of vulnerability detection, SAST tools, and testing it on some iOS open-source packages that have publicly reported vulnerabilities in noted by Common Vulnerabilities and Exposures (CVE) in them to see if the vulnerabilities can be found using that method. That would show what the current state of SAST tools is, how easy they are to use and what type of vulnerabilities it is good at detecting and which vulnerabilities remain undetected. It would also show if the problem is in the testing methods themselves and they need to be developed further or if the problem is that the existing tools are simply not being used widely enough. For that the following research questions were formulated:

- RQ1: Could these SAST tools have prevented the reported vulnerabilities?
- RQ2: What type of vulnerabilities could have been prevented?

With RQ1 focusing more on how these tools could have prevented any of these vulnerabilities in general and how effective they are at it and RQ2 looking at the distribution of what was detected to discern what type of vulnerabilities SAST tools are good at detecting and what they miss to either highlight the directions these tools should be improved on or make it easier to match SAST tools with other tools that complement them by being good at detecting the types of vulnerabilities SAST tools struggle with.

2. Structure of the thesis

The thesis has been divided into eight sections: Introduction, Background, Related work, Methodology, Results, Discussion and Conclusion.

The Background chapter describes how security vulnerabilities are being documented, what some of the common types of vulnerability detection tools are and how they work. Some

related work is discussed and then the initial dataset of vulnerabilities used for testing in this thesis is introduced and described.

The Methodology section describes how the initial dataset was reduced to better fit the aims of this thesis and make the testing process easier. Then the process of finding the tools that were tested is described and finally testing process itself is covered.

In the Results chapter the results of the testing are described. Every tested tool has its own section where it is described how they work, what the specific process was for testing them, what problems occurred during testing and finally what vulnerabilities were found. In addition the research questions introduced in the Introduction are answered.

The Discussion section focuses on more subjective conclusions based on the results, threats to the validity of this thesis and the testing process and future work that could be done and some lessons and thoughts from testing and looking back at the results.

Finally in the Conclusion section the thesis is summarized and the results concluded.

2. Background

In this section some background information in given on security vulnerabilities. Then the different types of vulnerability detection tools are introduced as well as Open Web Application Security Project. Finally the dataset used in this thesis to test the vulnerability detection tools is introduced.

2.1. Security vulnerabilities in code

A security vulnerability represents an attack vector that a potential attacker could exploit for attacks ranging from Denial-of-service to reading sensitive data or even executing custom code. There are many different types of vulnerabilities that can occur in all layers of the application, from authentication bypasses to memory mismanagement. Vulnerabilities are a constant and unavoidable part of software development and often occur when a developer misses or does not know about some edge case in whatever they are implementing which could be exploited. Due to them being edge cases they are often very difficult for human eyes to spot in the code and thus often come out after the program has already been in use for some time which means they can be exploited and if the project is used as a dependency by other applications then they might also be vulnerable.

Once a vulnerability is publicly disclosed it gets reported to the Common Vulnerabilities and Exposures (CVE) Program which is a project administered by The MITRE Corporation that aims to identify, define, and catalog publicly disclosed vulnerabilities. [1] The CVE Program then assigns the vulnerability a CVE Record with an id and description of the vulnerability. The primary purpose of the CVE Program is to keep track of the vulnerabilities and make sure each has only one CVE assigned to them so people can use it to refer to the vulnerability. It is also useful for notifying developers of a vulnerability in some version of a dependency they are using so they can either update the dependency or otherwise make sure they are not affected by the vulnerability. The CVE system is also used by various cybersecurity products and services to build upon and provide useful features. One example of that is NVD or U.S. National Vulnerability Database which is a vulnerability database expanding on the CVE List database by also including fix information, severity scores and impact ratings [2]. NVD also has more advanced search features compared to CVE List so it is easier to find relevant CVEs and also has extra information on each vulnerability like fix information, severity scores, and impact ratings [2]. In conjunction with the CVE Program a Common Weakness Enumeration (CWE) List has also been developed consisting of common software and hardware weakness types [3]. Each CVE gets assigned one or more CWEs to categorize it and make searching and analyzing them easier.

2.2. OWASP

The Open Web Application Security Project or OWASP is a well-known nonprofit foundation with the goal to improve the security of software that organizes different community-led open-source projects and compiles information about many different aspects of the software security world [7]. To succeed in that goal one of the many things they do is compiling lists of different tools meant to be used to improve the security of applications

including a list of SAST tools [8] which was used in this thesis to select the tools to test. Probably one of the their best known projects is the Top 10 Web Application Security Risks document that they compile every few years about the most critical security risks in web applications [9]. The different OWASP lists were also used to learn about the other types of tools and read about different prevalent vulnerabilities so the OWASP project has been useful as a source of information about the software security world.

2.3. Vulnerability detection tools

Vulnerabilities are a constant problem in software development to the point of seeming unavoidable. There are however ways to minimize the amount of vulnerabilities that get through besides developers paying more attention to security when coding. For one the ability of developers to pay attention and notice problems helped a good deal by focusing on good code style and readability so it would be easier to keep track of and understand what is happening in the code. One way of finding the vulnerabilities that still happen before they reach production is to use vulnerability detection tools.

There are a few different categories of vulnerability detection tools each with many different tools with different levels of efficacy. According to OWASP [3] there are three main types of tools dealing with the application itself: Static application security testing (SAST), Dynamic Application Security Testing (DAST) and Interactive application security testing (IAST). The primary purpose of these tools is finding security vulnerabilities but they can sometimes also find other bugs and SAST tools can often also include various code style recommendations and other notes to improve code quality. The broad differences between the different types of tools are shown in Table 1. Some tools do have the capability to perform more than one of these types of testing but in those cases the different approaches usually operate completely separately from each other. Next each tool category will be described in more detail.

Table 1. Different types of vulnerability detection tools

	SAST	DAST	IAST
Testing technique	White-box testing	Black-box testing	Dynamic testing
Requires source code access	Yes	No	Yes
Requires running app	No	Yes	Yes
Usable from the very start of development	Yes	No	No
Nr of tools listed by OWASP	107	98	4

SAST

SAST or Static application security testing is a white-box testing technique which involves analyzing the source code for issues. [4] As seen in this thesis different SAST tools have different approaches for accomplishing this from using regular expressions to find some patterns in the code to creating a database based on observing the build process to understand how the code fits together in execution and then querying the database. SAST tools are capable of finding simpler issues and flagging specific functions and approaches that might be dangerous and the access to the source code also allows some of the tools to include more broad code analysis to also provide code style recommendations in addition to searching for the security vulnerabilities. However SAST tools struggle with finding more complex issues due to the complexity of understanding the code and finding the various potential edge cases that might have been missed. [4] That means that SAST tools are unable to find some types of vulnerabilities at least using current approaches. It is still important to use SAST tools however as they can be used during development to give nearly instant feedback and can be used straight from the start of development as opposed to the other tools that require the program to be running to work. Additionally some SAST tools have the code quality features that the other types of tools cannot provide.

DAST

DAST or Dynamic Application Security Testing is a black-box testing technique that involves sending various possible inputs to the program while its running and observing the output. [5] These inputs can include various malformed data and formats the program might not expect with the objective being to find cases where the program somehow does not behave as expected or crashes outright which could be exploited by an attacker. [5] The benefit of DAST testing and why it is relatively effective is that it approaches the program from the user or attackers perspective and automatically attempts to try all cases that the attacker might try. Due to it being able to use only the inputs of the program the range of possibilities is not too large and it does not need to deal with the complexities of trying to understand source code. This makes it much easier to make effective tools but the drawback is that DAST tools can only be used once the program is running which means that it cannot be used at the very start of development. It also means that during development the developer needs to complete what they are doing and run the code before getting the feedback from the tool. This also means that DAST testing cannot be used to test something that is not meant to be run on its own like some code libraries.

IAST

IAST or Interactive application security testing is an approach that uses runtime testing techniques which deploying sensors into the application that continually monitor and gather information about the application while it runs and is being interacted with by other tests and users. [6] IAST tools also have access to the source code and so can point to some piece of code when a vulnerability is discovered. [6] Compared to SAST it has similar drawbacks as DAST in that it requires the application to be running but the main difference and benefit

of IAST compared to DAST is that it has access to the source code and so can provide more information about issues which helps with fixing the issue faster.

2.4. Initial dataset

To be able to test the SAST tools a dataset of CVEs about vulnerabilities in open-source packages was required as it was important to test the tools on real world issues to determine their practical efficacy. CVEs were needed to source the vulnerabilities to test and they often include information in their NVD entry that is helpful for testing SAST tools like where in the code the vulnerability exists and how the vulnerability was fixed. The affected packages needed to be open-source to have access to the code to be able to use the SAST tools.

The initial dataset was taken from “Dataset: Dependency Networks of Open Source Libraries Available Through CocoaPods, Carthage and Swift PM” [10].This dataset was originally produced by collecting all open-source packages present in the three package managers used for Swift development and then matching the packages with the vulnerability data from NVD to find the CVEs affecting these packages [10]. The dataset consists of 149 vulnerabilities affecting 41 packages over 1339 package versions. This dataset was chosen due to the wish to consider all available open-source packages from a single ecosystem, in this case iOS development. The dataset was also familiar and easily available and fulfilled all the needs of this thesis.

3. Related work

In this section some similar and related papers are discussed.

Rahkema et al. in “Vulnerability Propagation in Package Managers Used in iOS Development” [11] used the same dataset of CVEs used in this thesis to analyze how vulnerabilities propagate in the package dependency network. They found that most publicly reported vulnerabilities are in packages written in C but vulnerabilities in Objective-C and Swift packages had the highest impact.

Another project that is somewhat related is the CVEfixes project which is a dataset of vulnerabilities in open-source software focused on how the vulnerability was fixed. It is automatically collected from the CVEs present in the NVD with the goal of aiding data-driven security research. The process of vulnerability collection used is described in the paper “CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software” by Guru Bhandari et al. [12] The process broadly consists of automatically collecting the CVE records via JSON vulnerability feeds, collecting the details of the CVEs and cloning the project repositories to extract information about the vulnerable code and the corresponding fixes. The CVEfixes dataset could have been used in this thesis as well as an alternative to the dataset that ended up being used.

There has been some studies done before that are similar to this thesis. One such is Lenarduzzi et al. in “A critical comparison on six static analysis tools: Detection, agreement, and precision” [13] where they analyzed 47 Java projects using 6 static analysis tools including SonarQube. They mostly focused on code quality issues and finding out how much these tools agree with each other and how precise they are primarily through manually assessing the issues to determine if they are false positives or not. They concluded that there is little agreement between the tools and the precision of them is quite low. This thesis focuses more on security related problems and finding vulnerabilities as well as testing C and C++ projects.

Another broader study was done by Elder et al. in “Do I really need all this work to find vulnerabilities?” [14] where they applied four different vulnerability detection techniques including SAST and DAST to an open-source medical records system written in Java and compared the results of these different techniques. For their SAST testing they used three tools two of which are unnamed and the last one being SonarQube. They found most vulnerabilities with the SAST approach though with exploratory manual penetration testing (EMPT) they found more severe vulnerabilities. They also found that the speed of finding vulnerabilities was similar or better on manual techniques compared to automated ones which is contrary to what one might intuitively think. Contrary to most other studies they also found that SAST tools gave very few false positives. The cause of such a significant difference compared to other studies remains a little unclear but might be due to a combination of the tools used and the researchers being very conservative in declaring false positives.

Aloraini et al. in “An empirical study of security warnings from static application security testing tools” [15] also tested SAST tools on active C++ projects. They used six SAST

tools: Parasoft C/C++ test, PVS-Studio Analyzer, Clang Static Analyzer, Cppcheck, Flawfinder and RATS. Their primary goal was to find out if specific types of warnings given by the tools have a higher chance of being false positives. To determine which of the warnings were false positives they used the tools on two versions of each C++ project with one version being from 2012 and the other from 2017. That allowed them to compare the results of both versions and conclude that the warnings found in both must be false positives as if they were true problems they would have been fixed over 5 years of development. They found among other things that PVS-Studio and Parasoft C/C++ test produced the least proportion of false positives. They also found that the distribution of warnings over warning types stayed the same over the 5 years that they think might indicate that the code quality of the projects stayed the same. While the testing work done by Aloraini et al. is very similar to the testing done for this thesis the aim of testing is different. They focused on determining how many false positives occur and in what warning types while this thesis focuses on finding out if the true positive results would have been enough to prevent publicly reported vulnerabilities.

4. Methodology

This section describes the complete methodology used for testing the vulnerability detection tools and the steps taken to refine the dataset as well as how the tools were chosen.

4.1. Narrowing the dataset

The initial dataset, as described in the corresponding background section, is composed of open-source libraries used in iOS development that have vulnerabilities listed in the NVD database. The dataset includes 149 vulnerabilities from 41 packages over 1339 package versions.

As the libraries in the dataset are made for a wide variety of purposes and many of them are also widely used outside of the iOS ecosystem then the languages they are written in also vary. That poses a problem as most vulnerability detection tools support only a small selection of languages and ideally all the tools would be tested on the same dataset. Thus the most common language in the dataset will be selected and only the packages written in that language will be considered. The language the package is written in will be determined by looking at the language subsection on their GitHub repository. Many libraries in the dataset use multiple languages and those will be counted for each language with more than 10% coverage according to GitHub. This might lead to an issue in the libraries containing multiple languages that the vulnerability is not in the language that was selected for testing and thus not in a place the tools would look. In those cases the package will be removed from testing when it is discovered.

When the dataset has been narrowed to a single language, then the NVD entries for the CVEs about the remaining packages will be gone through to determine which of them contain a reference to where in the packages code the vulnerability is, either by naming the function or file in the NVD entry or linking to a commit or issue that specifies it. The purpose of this is to make it easier to determine if the tools found the vulnerability or not as without any reference to where it might be in the code it would be very time consuming to locate.

4.2. Tool selection

In this thesis the Static application security testing (SAST) approach was chosen, mainly due to it not requiring running the packages for testing as opposed to DAST or IAST testing. As the dataset consists of open-source packages used in iOS development then many of them are code libraries that are not meant to be run on their own but used in other applications. That means that some of them cannot be tested with DAST or IAST tools. In addition getting each of the 27 packages to run would be a considerable bit of extra work that is not necessary using SAST tools. IAST testing in particular would have been overly complicated as it involves integrating the sensors into the application and so does not lend itself well to testing many different packages in a row. Therefore the goal is to find SAST tools that could be used on the dataset to potentially find the reported security vulnerabilities in the source code.

To find the SAST vulnerability detection tools the OWASP SAST tools list [8] will be used which as of 30.July.2023 lists 107 SAST tools. The Open Web Application Security Project (OWASP) is a well-known nonprofit project with the goal to improve the security of the web. To succeed in that goal one of the many things they do is compiling lists of tools meant to be used to improve the security of applications with SAST tools being one such category of tools.

Tools that are not freely available will not considered as it would be difficult to gain access to them and the dataset consists of open-source libraries thus using only free tools makes some sense. The tools each support different sets of programming languages so to be able to test all selected tools on the same dataset all tools that do not support the selected language will also discarded.

4.3. Testing process

First all the CVEs about the same package will be gone through in the NVD database to determine if several of them are present in the same version of the package and thus could be tested simultaneously. As the versions listed under the NVD database entry of the CVE are a little unreliable and often include versions well before the point the vulnerability was introduced and sometimes including versions after it was fixed, then the search for the optimal versions to test will be conducted based on the fixing commits. The fixing commits are mostly linked to in the NVD entry or NVD links to an issue report from which the fixing commit can be found. The fixing commit lists in which versions it is included in on GitHub and shows where in the code the vulnerability was. The code just prior to the fix can then be compared to the state of the code in other versions to determine if the vulnerability was already present, erring on the side of caution if some changes have occurred. That way if two or more CVEs target different but close versions of the same package it can be determined if they can be tested on the same run thus reducing the workload.

Then the publicly available source code for the package version will be downloaded. The tool being tested will be run on the package code and the results marked down. Flawfinder, VisualCodeGrepper and Clang-tidy can be run on the source code as is though Clang-tidy might need some extra configuration if errors occur. SonarLint however first requires generating a compile_commands.json file for which the Makefile Tools plugin for Visual Studio Code will be used or if that has problems then the Linux command line tool bear will be used instead. As both basically build the project then all the required dependencies for each package will need to be downloaded first and the configuration scripts like autogen.sh and configure.sh need to be run if present. For that the GitHub readme for each package will be consulted. CodeQL will be run as a GitHub Action so the code will need to be uploaded to GitHub first. The GitHub Action has an autobuild feature which will be tried first but if that fails then a build script will need to be written for each failing package according to the instructions in the package readme file.

After the tool has finished its analyzes the resulting report will be looked through and it shall be determined if the vulnerability in question is among those found by the tool. To speed up looking through the reports the search feature will be used where possible to find

mentions of the file or function that the vulnerability is supposed to be in according to the CVE entry. Some effort will also be put into understanding the vulnerability and the code, including the code immediately preceding and following the target function in code execution. For that the “Find all references”, project search and “Jump to definition” features of Visual Studio Code will be used. The purpose of looking at the project code a little more broadly is that the vulnerability might be detected by the tool a little outside of the function pointed to by the CVE entry, either by the tool detecting some other aspect of the vulnerability compared to the CVE or the CVE itself being mistaken in the location.

The research questions will be answered based on these results. If a vulnerability is determined to be found by a tool then it will be counted as preventable for RQ1 and for RQ2 the types of the found vulnerabilities will be considered.

5. Results

In this section first the outcomes of the dataset narrowing and tool selection are described and then the results of testing the selected tools are given with some description of how each of the tested tools was to use.

5.1. Dataset

First, the language in which a package was written is was determined. The frequency of languages found in the 41 libraries is shown in Table 2.

Table 2. Programming languages by frequency in the dataset

Language	Frequency
C	22
C++	11
Objective-C	8
Swift	6
JavaScript	6
Java	5
Python	4
M4	2
Go	1
C#	1
Assembly	1
CMake	1
TypeScript	1

The most common languages were C and C++ and as most tools that support one of the languages also support the other then it was decided to test packages written in these languages. That left 27 packages as some of them contained both languages.

The dataset contained 112 CVEs about these 27 packages. Next it was determined which CVE entries in the NVD had information about where in program the vulnerability was. A total of 31 CVEs did not have that information and were discarded for the speed and ease of testing. That left 81 CVEs in 23 packages to be tested. The breakdown of these CVEs based on CWE is shown in Table 3 which is based on the CWEs specified in the NVD entry of the CVE with the vulnerability counted for each CWE if several were defined in the NVD. CWEs that appeared less than 3 times are not shown.

Table 3. Vulnerabilities in the dataset by CWE

CWE	Number of vulnerabilities
CWE-125: Out-of-bounds Read	25
CWE-476: NULL pointer dereference	7
CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	7
CWE-787: Out-of-bounds Write	6
CWE-190: Integer Overflow or Wraparound	6
CWE-20: Improper Input Validation	4

5.2. Tool selection

The search started with the 113 SAST tools listed in the OWASP SAST tools list. The tools listed as being under a commercial license were removed leaving 61 tools described as open source or free. Of those 61, only tools that support C and C++ were selected which left 10 tools. While going through the tools however one additional tool(clang-tidy) was found by happenstance and was included despite it not appearing int the OWASP list as it was a tool searching for vulnerabilities in C and C++ code. That left 11 tools:

- SonarQube
- HCL AppScan CodeSweep
- Flawfinder
- CodeQL (GitHub Advanced Security)
- Google CodeSearchDiggity
- LGTM
- Microsoft PREFast
- ParaSoft
- PVS-Studio

- Veracode
- VisualCodeGrepper
- Clang-tidy

However while trying to download, set up and start using these tools it turned out that many of these could not actually be used for various reasons and needed to be discarded as well:

SonarQube does have a free version called the “Community Edition” but that edition does not support C/C++ [16], so SonarQube was replaced with SonarLint which is made by the same company and uses mostly the same backend tools so should find mostly the same things. The main drawback of SonarLint is that the Visual Code plugin, through which it was used, only scans a single file at a time and has no option for scanning the whole project at once.

Two versions of HCL AppScan CodeSweep were considered for testing: the GitHub Action and the Visual Studio Code plugin. The GitHub Action, while free itself, requires an API key for AppScan on Cloud which in turn requires a paid license. The Visual Studio Code plugin was technically usable but it was finding very few issues in anything and including most of the issues it itself claimed to be able to find which indicated that something was perhaps not working properly but there was no indication of what might be going wrong and no information about it online and so it was decided that it would be dropped from testing.

Google CodeSearchDiggity manual that comes with the tool says that the Google API the SAST proportion of the tool was using was discontinued in 2012 and that they are searching for a solution. No solution seems to have been found as it currently is still based on the same Google API and does not work.

Microsoft PREFast also has two versions: a GitHub Action and Visual Studio “analyze” feature. The GitHub Action only supports CMake projects built with the Microsoft Visual C++ Compiler and as some packages in the dataset do not have that option then this cannot be used. The Visual Studio “analyze” feature does work but after quite a bit of trying and even getting a couple of the packages tested it was decided that getting the packages properly set up in Visual Studio was too much work per package due to various compilation problems, various build path problems and Visual Studio being a generally quite unintuitive and cumbersome system. Due to that the testing on Microsoft PREFast was stopped.

ParaSoft has a news post from 2018 that claims to offer licenses to developers who are “an active contributor to an active and vital Open Source project that is recognized within the global Open Source community” judged on a case-by-case basis via email. [17] This thesis falls outside of that definition but an email was sent to ParaSoft anyway enquiring about it to which they responded that ParaSoft is a commercial tool and not free.

PVS-Studio offers two free ways to use their tools. The one for open source projects is the most relevant one for this thesis which has the conditions that the project needs to be

posted on publicly on GitHub or BitBucket [18] and have a link to PVS-Studio in its readme [19]. However that license does not apply to mirrors of projects [18] which is how it would be used in this project. The second option requires writing specific comments at the beginning of every file [20] which considering the number of packages and files is too much work.

Veracode does not appear to have any free version. No reference to one has been found in either their website [21] or documentation [22]. It is also listed under "Commercial" in a different OWASP list [23].

LGTM was discontinued in December 2022 and as it was a web based tool that means that it can no longer be used. Additionally LGTM also used CodeQL and its team moved to GitHub to work on GitHub Advanced Security so the underlying tool will be tested here anyway. [24]

That left 5 tools to be tested on our C/C++ packages:

- SonarLint
- Flawfinder
- CodeQL (GitHub Advanced Security)
- VisualCodeGrepper
- Clang-tidy

6.3. Testing results

Of the 81 CVEs tested 14 were found by at least one tool. However with most of the 14 vulnerabilities found it is debatable if "found" is the right word for them as the same spots are also flagged in versions after being fixed. Also in many cases the intended purpose of these warnings is not necessarily to say that a vulnerability exists in that spot but rather to notify the developers of a potentially dangerous function call or behavior that they should either reconsider or at least be careful with as they are difficult to implement correctly. In all of these packages there are many of these potentially dangerous spots flagged but most of them do not have an actual vulnerability associated with them. Of the 14 found vulnerabilities only CVE-2020-19498 can be definitively said to be a found vulnerability. It involved a division by zero and was found only by Clang-tidy and as opposed to all the other found CVEs did not continue to get flagged after being fixed. The found vulnerabilities and what tools found them can be seen in Table 4.

Table 4. All found vulnerabilities

Package	**CVE**	**CWE**	**Notes**	**Found by**
libevent/libevent	CVE-2016-10195	CWE-125: Out-of-bounds Read	Dangerous function call: memcpy	Flawfinder, VisualCodeGrepper, Clang-tidy
libevent/libevent	CVE-2016-10196	CWE-787: Out-of-bounds Write	Dangerous function call: memcpy	Flawfinder, VisualCodeGrepper, Clang-tidy

libevent/libevent	CVE-2014-6272	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found	Flawfinder, VisualCodeGrepper, Clang-tidy
libevent/libevent	CVE-2015-6525	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found	Flawfinder, VisualCodeGrepper, Clang-tidy
flif-hub/flif	CVE-2018-12109	CWE-787: Out-of-bounds Write	Dangerous function call: fgetc	Flawfinder
leethomason/tinyxml2	CVE-2018-11210	CWE-125: Out-of-bounds Read	Dangerous function call: strlen	Flawfinder, VisualCodeGrepper
libgit2/libgit2	CVE-2018-10887	CWE-190: Integer Overflow or Wraparound, CWE-125: Out-of-bounds Read, CWE-681: Incorrect Conversion between Numeric Types, CWE-194: Unexpected Sign Extension	The real issue is integer overflow but it leads to an out of bounds read via memcpy	Flawfinder, VisualCodeGrepper, Clang-tidy
libgit2/libgit2	CVE-2018-10888	CWE-125: Out-of-bounds Read, CWE-20: Improper Input Validation	Dangerous function call: memcpy	Flawfinder, VisualCodeGrepper, Clang-tidy
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125: Out-of-bounds Read	Dangerous function call: memcpy	VisualCodeGrepper, Clang-tidy
strukturag/libheif	CVE-2020-19498	-	Division by zero, reported with the path the program would take to get there	Clang-tidy
libimobiledevice/libplist	CVE-2017-5836	CWE-415: Double Free	Recommends that dynamic heap memory allocation should not be used	SonarLint
redis/hiredis	CVE-2020-7105	CWE-476: NULL pointer dereference	Recommends that dynamic heap memory allocation should not be used	SonarLint
eclipse/mosquitto	CVE-2021-34431	CWE-401: Missing Release of Memory after Effective Lifetime	Recommends that dynamic heap memory allocation should not be used	SonarLint
webmproject/libwebp	CVE-2016-9969	CWE-415: Double Free	Recommends that dynamic heap memory allocation should not be used	SonarLint

Next the testing results for each tool will be given together with a small description of how they work and how easy they were to set up and use during testing and what notable features they have.

Flawfinder

Flawfinder is an open source command line tool written in Python. It examines the C/C++ source code in a project and detects possible vulnerabilities which it then writes into a html report file. The detected issues are assigned a level between 0 and 5 according to their risk level. Self-described as a simple tool it works by having an internal database of functions that have known problems and doing some simple pattern matching on the source code to find all instances of calls to those functions. [25]

Due to its simple nature it was mostly easy and fast to setup and run, by simply installing it through pip and running it from the command line on the folder containing the project being tested. It did have some problems with a few packages where the tool failed with some errors which required deleting a file or two from the source code, the cause of these errors is not quite clear.

Flawfinder is mostly intended for quick scans of the source code early in development, potentially even before the code can be compiled. It can find and direct the developers attention to some common security issues that can arise with the functions used in the source code. Having found a potentially dangerous function call it assigns it a risk level and adds it to the report with a short description of what the potential issue is and a suggestion for how to avoid it. That speeds up fixing or even preventing these vulnerabilities and informs the developer of dangers they might not have been aware of.

However, as it is just looking for function calls without understanding what is going on around them it simply flags every function in its database like *memcpy* weather or not a vulnerability actually exists there so there are a lot of false positives. That makes it very difficult to find actual problems and difficult to determine if the vulnerability has been fixed or not especially once the codebase gets bigger. That in turn makes it very easy for developers to start ignoring the reported problems to the point that it loses almost all of its usefulness for actually detecting vulnerabilities. Especially as the functions included in Flawfinder are mostly not inherently vulnerable but rather difficult to use correctly which means that fixing the problem if it exists at all means adding checks before the function call and not replacing the function with something else. For some of these functions there are safer alternatives but in none of the CVE-s that were looked through was the actual method changed, the developers always just added checks. Flawfinder does allow for spots that have been verified to not be vulnerable to be ignored by the tool by adding comments before them which makes it a little easier manage false positives but that of course relies entirely on the developers ability to verify that no vulnerability remains.

Flawfinder found 6 vulnerabilities noted in the 81 CVEs tested and partially found 2 others that had multiple locations. The found vulnerabilities are shown in Table 5 and the full testing results are included in Appendix I.

Table 5. Vulnerabilities found by Flawfinder

Package	**CVE**	**CWE**	**Notes**
libevent/libevent	CVE-2016-10195	CWE-125: Out-of-bounds Read	Dangerous function call: memcpy

libevent/libevent	CVE-2016-10196	CWE-787: Out-of-bounds Write	Dangerous function call: memcpy
libevent/libevent	CVE-2014-6272	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found
libevent/libevent	CVE-2015-6525	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found
flif-hub/flif	CVE-2018-12109	CWE-787: Out-of-bounds Write	Dangerous function call: fgetc
leethomason/tinyxml2	CVE-2018-11210	CWE-125: Out-of-bounds Read	Dangerous function call: strlen
libgit2/libgit2	CVE-2018-10887	CWE-190: Integer Overflow or Wraparound, CWE-125: Out-of-bounds Read, CWE-681: Incorrect Conversion between Numeric Types, CWE-194: Unexpected Sign Extension	The real issue is integer overflow but it leads to an out of bounds read via memcpy
libgit2/libgit2	CVE-2018-10888	CWE-125: Out-of-bounds Read, CWE-20: Improper Input Validation	Dangerous function call: memcpy

However as Flawfinder simply flags all instances of the potentially dangerous function calls then all of these found vulnerabilities were hidden among a large amount of false positives, often about calls to the same function in other places where a vulnerability did not exist. For example in the package libevent 475 issues were found by Flawfinder. The issue the developers had missed in almost all of the found vulnerabilities was not necessarily the function call itself but rather insufficient checks before it so it is debatable if these vulnerabilities should in fact be called “found”. It might be more correct to say that Flawfinder warns about using these functions as they are difficult to use safely and not that Flawfinder really found these vulnerabilities.

The fact that only the function call itself is considered means that in all cases Flawfinder still flagged the same spots in package versions where the vulnerabilities had been fixed. That is partially due to that in all cases seen in the testing the fix to the vulnerabilities always consisted of adding more checks before the function call instead of replacing the function with something safer. Perhaps if the developers of these packages had seen warnings about these functions like Flawfinder gives then they might have actually replaced them and then Flawfinder would not have continued to flag those spots.

Flawfinder attributes each found issue to a CWE, but out of the 8 found CVEs the CWE named by Flawfinder and the one written in the NVD entry matched only once, in CVE-2018-11210. Table 5 shows the CWEs according to the NVD entry.

VisualCodeGrepper

VisualCodeGrepper is a standalone tool that has its own GUI though it can be run through the command line as well. As the name suggests it is a similar tool to Flawfinder in that it is fairly simple and uses grep-like text pattern matching to find specific function calls and a few other potentially vulnerable spots, and does not understand the code. Most of these functions that both VisualCodeGrepper and Flawfinder look for by default are taken from the Microsoft banned functions list. VisualCodeGrepper does look for a few more potential issues than Flawfinder and also allows adding of extra user made search patterns through its configuration file. After the scan the found issues are assigned a severity level and the results are shown in the GUI which also allows them to be exported to a file.

As VisualCodeGrepper works mostly the same as Flawfinder then it has mostly the same purpose and problems. It is primarily for quick scans probably early in development with the main purpose being not really finding actual vulnerabilities but rather directing the developers attention towards functions that are potentially dangerous so they can be replaced or at least handled with care. Because of that, from the point of view of actually finding vulnerabilities, it has a lot of false positives and unlike Flawfinder VisualCodeGrepper does not allow for marking these spots to be ignored once they are verified to be false positives. The main advantages compared to Flawfinder are that VisualCodeGrepper attempts to also find some places where there might be some memory mismanagement like not freeing the allocated memory, it finds *todo* and *fixme* comments that might mark unfinished and thus dangerous spots and allows the user to add additional functions and text fragments to the list of patterns being searched for.

It was easy to set up and use, it came with an installer and using it consists of selecting the target directory, clicking scan and then browsing the results. As it only does text pattern matching there was no problems getting it to work with any of the tested packages. The only issue with VisualCodeGrepper was that by default it does not recognize .cc files as C++ files even though they should be equivalent to .cpp files. So by default it leaves them out of its scans but that can be fixed easily by changing the configuration either through the GUI or editing the file.

In testing VisualCodeGrepper found 6 vulnerabilities and partially found another 2 just like Flawfinder. The found vulnerabilities are shown in Table 6 and the full testing results are included in Appendix II.

Table 6. Vulnerabilities found by VisualCodeGrepper

Package	**CVE**	**CWE**	**Notes**
libevent/libevent	CVE-2016-10195	CWE-125: Out-of-bounds Read	Dangerous function call: memcpy
libevent/libevent	CVE-2016-10196	CWE-787: Out-of-bounds Write	Dangerous function call: memcpy
libevent/libevent	CVE-2014-6272	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found

libevent/libevent	CVE-2015-6525	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125: Out-of-bounds Read	Dangerous function call: memcpy
leethomason/tinyxml2	CVE-2018-11210	CWE-125: Out-of-bounds Read	Dangerous function call: strlen
libgit2/libgit2	CVE-2018-10887	CWE-190: Integer Overflow or Wraparound, CWE-125: Out-of-bounds Read, CWE-681: Incorrect Conversion between Numeric Types, CWE-194: Unexpected Sign Extension	The real issue is integer overflow but it leads to an out of bounds read via memcpy
libgit2/libgit2	CVE-2018-10888	CWE-125: Out-of-bounds Read, CWE-20: Improper Input Validation	Dangerous function call: memcpy

Most of the found vulnerabilities were the same as in Flawfinder which is unsurprising as they are very similar tools. The only differences between them was that VisualCodeGrepper did not find CVE-2018-12109 and instead found CVE-2018-16790. In CVE-2018-12109 the potentially dangerous function is *fgetc* which the VisualCodeGrepper does not seem to search for by default. CVE-2018-16790 is another Out-of-bounds Read vulnerability involving the memcpy function which makes it curious that Flawfinder did not find it as well as Flawfinder was flagging a lot of memcpy calls. It is unclear why Flawfinder skipped this one, maybe in a failed attempt to avoid false positives or maybe a bug in the pattern matching.

Like Flawfinder the same spots continued to be flagged even after the vulnerabilities associated with them had been fixed. This is due to the same problems that VisualCodeGrepper searches only for the function call without understanding the code surrounding it and the package developers not replacing the function but rather adding checks to patch up vulnerabilities.

CodeQL

CodeQL is a semantic code analysis engine developed by GitHub that was used in this thesis through GitHub Actions but it can be used locally though the command line as well. It works by first building the code being tested and monitoring the build process which it then uses to extract all the relevant information about the code into a database. Once that is done queries can be made to the database to detect problems and do various analysis tasks on the code. [26]

CodeQL seems to be focusing a lot on users making their own queries to do their own analysis and problem detection but it does come with built-in query pack that can be immediately used. As writing our own set of queries is outside of the scope of this thesis then

built-in queries were used, more precisely the security-and-quality query suite that should include all of the built-in queries. The focus on user created queries might indicate an attempt to facilitate a community through which some highly effective query suites could be sourced but no repository for user created queries to use in this thesis was not found.

Building the code should give it a lot more information and understanding about how the code works and at least in theory allow for much more effective search for vulnerabilities. However at least the built-in queries gave very few results in testing. In fact none of the vulnerabilities looked for in testing were found. Some of that might be from their claimed focus on trying to minimize the amount of false positives detected but if that means that very little of anything is detected then it makes the usefulness of the tool somewhat questionable. Besides many of the found issues are still potentially dangerous spots, not necessarily vulnerabilities. Perhaps the usefulness and potential relies primarily in the framework it gives the user to write their own queries but as that was not done in this thesis it remains unclear how good or easy to write these queries are.

As already mentioned CodeQL did not find any of the vulnerabilities tested for but the testing results for it can be seen in Appendix III. The vast majority of the found issues reported in testing were in the quality portion of the security-and-quality query suite used for testing and most of those simple notes for commented out code and similar. The reason for including the quality queries in testing was that first of all some similar checks were done in Flawfinder and VisualCodeGrepper so for the number to be vaguely comparable it made sense to include them here as well. In addition when first starting testing the default query suite was used and that often did not give any results at all which made it difficult to determine if the tool was working at all especially since getting the tool to work was quite difficult.

In an ideal scenario it is very easy to use, just create a GitHub repository, add the Action through the GitHub menu, choose the appropriate query suite and the autobuild feature of CodeQL should mean that it is able to build and then scan your code automatically. The results will appear as code scanning alerts in the GitHub Security section from where you can view them, jump to the relevant piece of code and mark them as false positives or as “won’t fix”. However in testing the autobuild feature only worked about half the time with the other half requiring writing a build script for the package. These build scripts caused a lot of headaches during testing including giving up on 2 packages after spending a good while trying to get them to compile. Problems with compilers, their versions, dependencies being missing or under different names, some of the ways of building the packages not working when others did (CMake vs Make) etc. OpenSSL versions especially gave trouble in a few packages, in good part because the version being wrong caused an error message that did not indicate in any way what the problem was and only a handful of versions are allowed in Ubuntu. A lot of this is of course due to the nature of testing a large amount of different packages, some with some peculiarities in how they are compiled. When using CodeQL in development one only needs to get it to compile with one project and presumably the developers know well how their code needs to be compiled so it should not really be much of an issue.

Clang-tidy

Clang-tidy is a standalone clang-based linter tool that is integrated into various IDEs and was used in the testing for this thesis through its integration into Visual Code as part of the C/C++ extension. For understanding the code better it needs a compile commands database to be set up for the project which it uses to get build options and other build information about the files. To an extent it does work without compile commands as well but often throws errors and does not detect nearly as many problems.

Clang-tidy searches for some of the same function calls that Flawfinder and VisualCodeGrepper do but analyzes program paths to determine places where a null pointer error or division by zero might occur among other things. It also does some style checking for things like unused variables and other such things. So it is more sophisticated that Flawfinder and VisualCodeGrepper and is able to actually find some concrete issues in the code but most vulnerabilities tested for are still missed.

In testing Clang-tidy found 6 vulnerabilities and partially found another 2, most being the same as the ones found by Flawfinder and VisualCodeGrepper. The found vulnerabilities are shown in Table 7 and the full testing results are included in Appendix IV.

Table 7. Vulnerabilities found by Clang-tidy

Package	**CVE**	**CWE**	**Notes**
libevent/libevent	CVE-2016-10195	CWE-125: Out-of-bounds Read	Dangerous function call: memcpy
libevent/libevent	CVE-2016-10196	CWE-787: Out-of-bounds Write	Dangerous function call: memcpy
libevent/libevent	CVE-2014-6272	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found
libevent/libevent	CVE-2015-6525	CWE-189: Numeric Errors	Dangerous function call: memcpy, CVE contains several spots but only one was found
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125: Out-of-bounds Read	Dangerous function call: memcpy
strukturag/libheif	CVE-2020-19498	-	Division by zero, reported with the path the program would take to get there
libgit2/libgit2	CVE-2018-10887	CWE-190: Integer Overflow or Wraparound, CWE-125: Out-of-bounds Read, CWE-681: Incorrect Conversion between Numeric Types, CWE-194: Unexpected Sign Extension	Dangerous function call: memcpy

libgit2/libgit2	CVE-2018-10888	CWE-125: Out-of-bounds Read, CWE-20: Improper Input Validation	Dangerous function call: memcpy

One of the found vulnerabilities, CVE-2020-19498, involving a division by zero, is of special note as it is the only case in testing any of these tools where a tested vulnerability was clearly and confidently found with none of the usual qualifiers of “potentially dangerous” or similar and no threat of being a false positive. As usual the version of the package where the vulnerability had been fixed was also tested and no false positive was given here, which cannot be said for the other vulnerabilities. Of course the other issues involve a potentially dangerous function call which was not replaced in the fix and so it makes sense that these continue to be reported even though the actual vulnerability no longer exists.

The Visual Code extension for C/C++ was used for testing Clang-tidy as it is integrated into the extension under the “Run code analysis” menu option. This was very easy to set up in itself however to run properly Clang-tidy requires a compile commands database to be set up for the project which it can also do itself from a compile_commands.json file. In some packages some trouble was had to get the package to compile to be able to generate the file. There are a few options for generating the compile_commands.json file, the two used in testing for this thesis were the CMake compile option for it, used in testing mainly through the CMake Tools extension for Visual Code, and the other was the command line tool Bear, which is a tool specifically for generating the compilation database for clang tooling. The need for using two different tools came from the fact that the CMake option was slightly easier to use but not all of the packages were set up to support it. As a Windows computer was used for testing then to make life easier an Ubuntu installation was set up through WSL on which both the compilation and Visual Code were run.

It can be told to analyze all files at once. Once the compile commands database was set up however it was very easy to use by simply telling Visual Code to run analyzes on the active file with an option also provided to run the analyzes on all files. False positives can also be marked as ignored by adding inline comments telling the tool to not check the relevant lines.

SonarLint

SonarLint is a linter developed by SonarSource, the developers of SonarQube. SonarLint checks all the same things as SonarQube and reportedly even uses the same engine for its work, the main difference between them being that SonarLint is focused on instant feedback when writing code and SonarQube gives an overview of the state of the whole project including some history of previous scans. SonarLint seems to mostly work on the same way as Clang-tidy does as it also needs the compile commands file to be generated before it can do its analyzes, but the two tools find quite a bit of different issues in the same code though there is of course some overlap. SonarLint has a much more in depth focus on code style and vulnerability prevention through that with recommendations for not using some ways of doing things like dynamic heap memory allocation. It however

does not flag the potentially dangerous functions many of the other tools were focused on nor does it check for division by zero and null pointer errors like Clang-tidy did.

SonarLint found 4 of the tested CVEs, all of them cases where dynamic heap memory allocation was used and SonarLint recommended that it should not be used. The found vulnerabilities are shown in Table 8 and the full testing results are included in Appendix V.

Table 8. Vulnerabilities found by SonarLint

Package	**CVE**	**CWE**	**Notes**
libimobiledevice/libplist	CVE-2017-5836	CWE-415: Double Free	Recommends that dynamic heap memory allocation should not be used
redis/hiredis	CVE-2020-7105	CWE-476: NULL pointer dereference	Recommends that dynamic heap memory allocation should not be used
eclipse/mosquitto	CVE-2021-34431	CWE-401: Missing Release of Memory after Effective Lifetime	Recommends that dynamic heap memory allocation should not be used
webmproject/libwebp	CVE-2016-9969	CWE-415: Double Free	Recommends that dynamic heap memory allocation should not be used

While the results of the testing of SonarLint are somewhat underwhelming in terms of the vulnerabilities that it found, the issues that it did flag in the code were valid and made sense and few false positives were detected. In general SonarLint seemed like the most useful tool of the five tested though much of this was more focused on writing readable and clean code and not specifically finding vulnerabilities. Readable and clean code should however also lead to less vulnerabilities through the code being more understandable and thus less mistakes being made by the developers.

The four vulnerabilities found during testing continue to be flagged by SonarLint after the actual vulnerability has been fixed as much like previously with other tools detecting the dangerous function calls with the issue being that the developers of the packages are not replacing the underlaying unsafe practice but adding checks to patch up the hole.

In setup SonarLint was much the same as Clang-tidy as SonarLint was also used in testing through the Visual Code plugin and required the same compile_commands.json file to be generated before it would work. The primary difference in the use of SonarLint and Clang-tidy was that SonarLint triggers its analyzes of a file when the file is saved and not through a menu option. The Visual Code plugin for SonarLint also does not allow running SonarLint on the entire project at once though some plugins for other IDEs do. Issues can be marked as ignored similarly to Clang-tidy by adding an inline comment to the relevant places or if the SonarLint instance is connected to a SonarQube installation then it can be marked as ignored through the SonarQube GUI.

7. Discussion

In this section the results are discussed and opinions given on what they might mean. Some other options for finding vulnerabilities are then discussed together with what might be done further to continue the research. Finally some potential issues with the research done are discussed.

7.1. Could these SAST tools have prevented the reported vulnerabilities

The strict answer to RQ1 *Could these SAST tools have prevented the reported vulnerabilities?* is according to our definition 14 out of the 81 vulnerabilities were found by at least one tool and thus could have been prevented. However many of these found vulnerabilities were low confidence and difficult to differentiate from the false positives so the primary usefulness of SAST tools does not seem to be finding existing vulnerabilities.

The vast majority of the vulnerabilities in the dataset are cases where there is some check missing for a little edge case. It seems likely that a good proportion of them can't really be found by a SAST approach due to the complexity that would need to be considered by the tool just being too high. Certainly a pure regex approach like taken by Flawfinder and VisualCodeGrepper seems impossible. Maybe an approach that has compilation information and could understand the code somewhat might be able to come closer or at least they seem to have more potential. This is also exemplified by Clang-tidy being able to find things like null pointer errors and divisions by zero by analyzing the different paths the program might take. However as it currently stands with the tested tools even the tools that have compilation information do not really find that many of the actual vulnerabilities and are best suited for simple errors and code style recommendations.

That being said the tested tools are not useless for reducing the number of vulnerabilities in code. The best of the tested tools found 8 out of 81 CVEs which is still a significant number. Additionally tools like SonarLint improve code readability and quality by giving good recommendations for improving code style and note solutions and functions that are difficult to use safely. Good readable code in turn makes it much easier to avoid making mistakes when implementing places that could produce vulnerabilities. In this way even the simpler tools like Flawfinder and VisualCodeGrepper can be quite useful to use early in development to avoid accidentally using functions that are unsafe and have safer alternatives before getting so far along in development that replacing them would be too much work. Making the developers think more about what they are using from the start of development goes a long way toward preventing these situations where vulnerabilities are easy to come. Fixme and todo comments mark unfinished code which might be vulnerable and so flagging them keeps developers from forgetting about them which can otherwise be easy to do. So these SAST tools can be quite useful for preventing vulnerabilities appearing in the first place but for detecting existing vulnerabilities in a preexisting projects some other approach might be more successful. At least some other approached need to be also used.

It might also be the case that with some programming languages other than C or C++ the SAST approach might work better. C and C++ are lower level languages than most other widely used languages and at least some of the vulnerabilities involved are to do with

memory related problems that would not exist in higher level languages. That being said many of the tested vulnerabilities can exist in any language and the core approach used by the SAST tools will be the same regardless of language. For example SonarQube and by extension SonarLint are some of the most widely used SAST tools for testing Java applications and while the rules it uses to scan code are somewhat different the underlying approach remains the same. Based on the code of the tested packages it also appears that C/C++ have a culture of relatively poor readability code with short non-descriptive variable names, extremely long files and so on, which might mean that it would benefit a good deal from the improving code readability and style aspects of these tools.

As for RQ2 *What type of vulnerabilities could have been prevented?* as the number of found vulnerabilities is low and they are of various types no clear answer was found. The most common CWE among the found vulnerabilities was Out-of-bounds Read (CWE-125) which counts for 5 of the 14 found vulnerabilities and additional 2 are Out-of-bound Write (CWE-787). However 2 of the tested tools (CodeQL and SonarLint) did not find any of these and CWE-125 is also the most common CWE in the dataset comprising 25 of the 81 tested vulnerabilities so these results don't necessarily indicate any specific types of vulnerabilities that SAST tools are good at finding.

7.2. Issues with NVD

NVD seems to have some problems and especially the details added by NVD are not as reliable as they could be. For example the affected versions list is often wrong, either not including enough versions or including too many. Some examples of that are CVE-2017-5835 which is fixed in version 2.0.0 but NVD includes versions up to 2.2.0 and CVE-2019-6285 which only lists 0.6.2 as affected but the problem exists well before that. Of course tracking down the exact range of versions is time consuming so the creators either only list the version the issue was found in or all versions before it. Sometimes the specified function is incorrect. There is also a lot of variation in what links are provided in the NVD. The CWE for a vulnerability also often seems to be assigned based on what the problem seemed to be when the vulnerability was detected and not what it actually turned out to be upon fixing. These issues might be especially relevant for projects like CVEfixes and others that use some automated approach to the CVEs as while the gist of them is usually correct, a lot of the data included is not as reliable as one might hope. This is in large part due to the community driven nature of the project which inevitably causes some as nobody really feel that it is their responsibility and thinks that someone else is going to have a look at it at some point. Also made worse by it being a quite technical thing that requires some concentration to do correctly. Obviously the CVE project does a lot of good and is a good resource for various purposes but there is room for improvement.

CWEs being assigned based on the original error message by which it was found or otherwise guessing might also partly explain why the CWEs from NVD did not match well with the CWEs from Flawfinder. The first being the guessed cause of the vulnerability based on the limited information available shortly after first being discovered and the other an issue in the code that might lead to a vulnerability. The CWEs Flawfinder assigns are of course also not directly applicable to a vulnerability, at least not always as firstly the issues it found

are often not real vulnerabilities and secondly it can be subjective what CWE is correct or at least multiple CWEs can be at work simultaneously.

7.3. Other approaches

There are many other approaches to searching for vulnerabilities besides SAST that would ideally be used in addition to it. The different approaches will find different sets of vulnerabilities so using several will result in the most vulnerabilities being found and fixed.

DAST testing

DAST testing is black-box testing approach so as opposed to SAST testing no code is analyzed but rather the program is run and then various inputs and the outputs analyzed so that potential flaws and vulnerabilities could be found. [27] The main benefit of DAST testing is that as it is done from the point of view of the user it only has the possible inputs to the program to consider. That is a more focused and much less complex issue compared to analyzing and understanding code and thus should allow for more complete coverage and fewer false positives compared to SAST. The main drawback compared to SAST tools is that the program needs to be running for DAST testing which means that more time passes after the code is written before the issues are found and it cannot be used straight from the start of the project. Additionally not all projects are even runnable as for example some code libraries might not function as standalone programs which means that testing them via DAST would require writing some code specifically for it which in turn would mean that the issues found might not be applicable to actual use cases.

Quite a few of the GitHub issues for the tested CVEs mentioned being originally found by fuzz testing which is the part of DAST testing that generates and various random or semi-random data to the program so any missed edge cases would be discovered. This includes malformed data and trying various inputs the program might not be expecting. [28] In a few cases with these tested CVEs however the developers of the package decided that the found issue was an incorrect usage of their package rather than a flaw and refused to fix it in effect calling it a false positive. So in some cases it can be debatable what sort of inputs are in scope and what are not. Particularly packages meant to be used in other programs and not directly accessible to the end user can get away with not being that safe and leaving the edge case checking to the developers using their package. Though there is often little reason not to consider and fix these issues besides time constraints. Part of the reason that fuzz testing was quite prevalent in these open source projects might also be explained by it being a relatively easy way to contribute for people not already involved and familiar with the code as it does not require them to go through and understand the source code.

AI

One emerging way to do SAST testing not looked at in this thesis is also to use AI tools like ChatGPT. ChatGPT itself can try to identify issues in code snippets and more focused AI powered tools are being developed, some like BurpGPT [30] and ChatGPTScanner [31] use ChatGPT but others like SmartScanner [32] or Hacker AI [33] do not. Currently these tools are still very new and require some refinement, but they have potential to surpass the current SAST tools after they have had some time to mature. The reason for not including something

like ChatGPT in this thesis was that firstly it came out after the testing was already underway and as mentioned it is currently still very new and needs some time to really become good. Also ChatGPT is a very general language model not a tool meant for testing code though it can do it to some degree. However there is no doubt that new AI tools focused on testing will be coming out soon if they haven't started to already.

7.4. Threats to validity

It might be that these tested tools actually prevent more vulnerabilities than these testing results suggest. One reason for that is that there is a selection bias in the tested vulnerabilities. It seems reasonable to assume that most if not all of these packages do use some sort of tools to detect problems during development. That means that most simpler and easier to detect problems get detected and fixed well before they reach far enough to get a CVE assigned to them. Some of the packages might even be already using one of the tools used in testing for this thesis so in those cases the testing for this thesis would have produced no positive results despite the tool potentially having already having prevented a number of vulnerabilities.

Another way that the results of the testing might not be completely accurate is that there is a chance that in some cases the tool did find something relating to the vulnerability but it was in some other file not mentioned in either the NVD entry, GitHub issue or fixing commit and not obviously connected and thus was missed when looking through the results. As there was quite a few of the packages to go through it was not possible to become familiar enough with how each of them worked to fully understand where to look for the causes of each vulnerability if it was not said in the available information about the CVE. However that chance is relatively small, especially some effort was made to at least try to understand the code flow and look a little more broadly than just the function specified, so the results should still be fairly accurate in this respect.

7.5. Future work

As SAST tools are by necessity quite language specific then it would be good to do the same sort of testing on a different dataset consisting of some other widely used programming language like Java or JavaScript. Then the results could be compared for a better overview of what SAST tools can do and what they work well with. That could also be a good opportunity to include some new AI powered tools as well to see what they can do and if their findings differ from the existing SAST tools at all. As in this thesis only freely available tools were used then it would be also good to include some commercial programs as they might have more and better features due to having more resources and customers being more vocal if they are unhappy with the tool.

It would also be interesting to try DAST testing tools on the same dataset and compare their results. Due to DAST and SAST being very different approaches it seems likely that they would also find different vulnerabilities. Due to DAST testing only being applicable to programs that can be run on their own and not really being usable for code libraries care would have to be taken in the selection of the dataset to only include packages that are runnable.

Thus the same dataset as used in this thesis would not be usable, at least not fully, though a subset could be selected that would be testable with a DAST approach.

Conclusion

The goal of this thesis was to find out how well could using existing SAST tools have prevented reported vulnerabilities and by extension how effective they are. For that a dataset of 149 CVEs from open-source packages used in iOS development was selected from which 81 CVEs from C and C++ language packages. The OWASP SAST tools list [5] was used as a source of SAST tools from which tools that were free to use and supported C and C++ were selected. This resulted in 5 SAST tools: Flawfinder, VisualCodeGrepper, CodeQL, Clang-tidy and SonarLint.

Each of the 81 CVEs was then tested with each of the 5 tools. This resulted in 14 of the 81 CVEs being found by at least one tool but most of these findings were low confidence and hidden among large amounts of other very similar results without a vulnerability so it would have been very difficult for a developer to actually find and fix these vulnerabilities based on these results. However the tools did give some good recommendations on improving code style and readability and directed attention to functions and techniques used in the code that are difficult to implement safely.

Based on these testing results it seems that the primary value of these SAST tools is not so much detecting existing vulnerabilities but rather notifying the developers of potentially dangerous spots and recommending ways to improve the readability and style of the code. Following these recommendations should lead to better quality code and the developers being more attentive to security issues which in turn should lead to fewer mistakes being made and it being easier to detect existing vulnerabilities.

For future work it would be good to test some tools on a different widely used programming language perhaps including some paid tools and compare the results with the results from this thesis. It would also be interesting to test DAST tools in a similar manner to see how they compare to SAST tools in finding security vulnerabilities.

References

[1] "CVE Overview," The MITRE Corporation, [Online]. Available: https://www.cve.org/About/Overview. [Accessed 24 July 2023].

[2] "FAQs - What is the relationship between CVE and the NVD (U.S. National Vulnerability Database)?," The MITRE Corporation, [Online]. Available: https://www.cve.org/ResourcesSupport/FAQs#pc_introcve_nvd_relationship. [Accessed 24 July 2023].

[3] "About CWE," The MITRE Corporation, [Online]. Available: https://cwe.mitre.org/about/index.html. [Accessed 5 August 2023].

[4] "About the OWASP Foundation," The OWASP Foundation, [Online]. Available: https://owasp.org/about/. [Accessed 25 July 2023].

[5] "OWASP Source Code Analysis Tools," The OWASP Foundation, [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools. [Accessed 6 November 2022].

[6] "OWASP Top Ten," The OWASP Foundation, [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed 25 July 2023].

[7] "OWASP DevSecOps Guideline - Vulnerability Scanning," The OWASP Foundation, [Online]. Available: https://owasp.org/www-project-devsecops-guideline/latest/02-Vulnerability-Scanning. [Accessed 30 July 2023].

[8] "Static Code Analysis," The OWASP Foundation, [Online]. Available: https://owasp.org/www-community/controls/Static_Code_Analysis. [Accessed 30 July 2023].

[9] "Dynamic Application Security Testing," Synopsys, [Online]. Available: https://www.synopsys.com/glossary/what-is-dast.html. [Accessed 30 July 2023].

[10] "Interactive Application Security Testing," Synopsys, [Online]. Available: https://www.synopsys.com/glossary/what-is-iast.html. [Accessed 30 July 2023].

[11] K. Rahkema and D. Pfahl, "Dataset: Dependency Networks of Open Source Libraries Available Through CocoaPods, Carthage and Swift PM," 2022.

[12] K. Rahkema and D. Pfahl, "Vulnerability Propagation in Package Managers Used in iOS Development," 2023.

[13] G. Bhandari, A. Naseer and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their," 2021.

[14] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," 2023.

[15] S. Elder, N. Zahan, R. Shu, M. Metro and V. Kozarev, "Do I really need all this work to find vulnerabilities?," 2022.

[16] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi and Y. Higo, "An empirical study of security warnings from static application security testing tools," 2019.

[17] "SonarQube products," SonarSource, [Online]. Available: https://www.sonarsource.com/products/sonarqube/downloads/. [Accessed 9 April 2023].

[18] "Parasoft Supports Open Source Development Community with Free Access to Parasoft’s Entire Suite of Enterprise-Class Test Automation Software," PARASOFT, 11 June 2018. [Online]. Available: https://www.parasoft.com/news/parasoft-supports-open-source-development-community-free-access-parasofts-entire-suite/. [Accessed 30 April 2023].

[19] "Free PVS-Studio for those who develops open source projects," PVS-Studio LLC, 22 December 2018. [Online]. Available: https://pvs-studio.com/en/blog/posts/0600/.

[20] "Free PVS-Studio license for Open Source," PVS-Studio LLC, [Online]. Available: https://pvs-studio.com/en/order/open-source-license/. [Accessed 30 April 2023].

[21] "Ways to Get a Free PVS-Studio License," PVS-Studio LLC, 11 March 2019. [Online]. Available: https://pvs-studio.com/en/blog/posts/0614/.

[22] "Veracode," Veracode, [Online]. Available: https://www.veracode.com/. [Accessed 30 April 2023].

[23] "Veracode Docs," Veracode, [Online]. Available: https://docs.veracode.com/. [Accessed 30 April 2023].

[24] "Testing Tools Resource," The OWASP® Foundation, [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/stable/6-Appendix/A-Testing_Tools_Resource. [Accessed 30 April 2023].

[25] B. v. Schaik, "The next step for LGTM.com: GitHub code scanning!," GitHub, 15 August 2022. [Online]. Available: https://github.blog/2022-08-15-the-next-step-for-lgtm-com-github-code-scanning/.

[26] "FlawFinder Home Page," [Online]. Available: https://dwheeler.com/flawfinder/. [Accessed 10 June 2023].

[27] "About CodeQL," GitHub, Inc., [Online]. Available: https://codeql.github.com/docs/codeql-overview/about-codeql/. [Accessed 10 July 2023].

[28] "Dynamic Application Security Testing," OWASP® Foundation, [Online]. Available: https://owasp.org/www-project-devsecops-guideline/latest/02b-Dynamic-Application-Security-Testing. [Accessed 21 07 2023].

[29] "Fuzzing," The OWASP® Foundation, [Online]. Available: https://owasp.org/www-community/Fuzzing.

[30] "BurpGPT," Aegis Cyber Ltd., [Online]. Available: https://burpgpt.app/. [Accessed 8 August 2023].

[31] "ChatGPTScanner," [Online]. Available: https://github.com/YulinSec/ChatGPTScanner. [Accessed 8 August 2023].

[32] "Smart Web Application Vulnerability Scanner," [Online]. Available: https://www.thesmartscanner.com/. [Accessed 8 August 2023].

[33] "Hacker AI," [Online]. Available: https://hacker-ai.ai/. [Accessed 8 August 2023].

Appendix

I. Testing results for FlawFinder

Package	CVE	CWE	Version tested	Found vulnerability	Notes
k-takata/onigmo	CVE-2019-16162	CWE-125	6.2.0	No	Found issues: 915
k-takata/onigmo	CVE-2019-16161	CWE-476	6.2.0	No	
libevent/libevent	CVE-2016-10195	CWE-125	2.1.4	Yes	Found issues: 475, Dangerous function call: memcpy
libevent/libevent	CVE-2016-10197	CWE-125	2.1.4	No	
libevent/libevent	CVE-2016-10196	CWE-787	2.1.4	Yes	Dangerous function call: memcpy
libevent/libevent	CVE-2014-6272	CWE-189	2.1.4	Partially	Dangerous function call: memcpy
libevent/libevent	CVE-2015-6525	CWE-189	2.1.4	Partially	Dangerous function call: memcpy
libuv/libuv	CVE-2015-0278	CWE-273	0.10.33	No	Found issues: 311
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125	1.12.0	No	Found issues: 1065
libimobiledevice/libplist	CVE-2017-7982	CWE-190	bbd3379	No	Found issues: 78
libimobiledevice/libplist	CVE-2017-5545	CWE-125	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5209	CWE-125	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5835	CWE-770	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5836	CWE-415	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5834	CWE-125	bbd3379	No	
nanopb/nanopb	CVE-2021-21401	CWE-763	0.4.0	No	Found issues: 135
nanopb/nanopb	CVE-2020-26243	CWE-20, CWE-119	0.4.0	No	
nanopb/nanopb	CVE-2020-5235	CWE-125	0.4.0	No	
libssh2/libssh2	CVE-2019-17498	CWE-190	1.4.3	No	Found issues: 362
libssh2/libssh2	CVE-2019-13115	CWE-190, CWE-125	c07bc647f2	No	Found issues: 579
libssh2/libssh2	CVE-2019-3859	CWE-125	1.4.3	No	
libssh2/libssh2	CVE-2016-0787	CWE-200	d453f4ce3c	No	Found issues: 456
libssh2/libssh2	CVE-2015-1782	CWE-20	1.4.3	No	
jbeder/yaml-cpp	CVE-2019-6285	CWE-674	0.6.3	No	Found issues: 93
jbeder/yaml-cpp	CVE-2019-6292	CWE-674	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-11692	CWE-617	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20573	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20574	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-5950	CWE-119	0.6.3	No	
libimobiledevice/libusbmuxd	CVE-2016-5104	CWE-284	1.0.10	No	Found issues: 27
aomediacodec/libavif	CVE-2020-36407	CWE-787	0.8.1	No	Found issues: 135
flif-hub/flif	CVE-2018-11507	CWE-834	0.3	No	Found issues: 446
flif-hub/flif	CVE-2018-12109	CWE-787	0.3	Yes	Dangerous function call: fgetc

flif-hub/flif	CVE-2018-14876	NVD-CWE-noinfo	0.3	No	
flif-hub/flif	CVE-2017-14232	CWE-399	0.3	No	
flif-hub/flif	CVE-2018-10971	CWE-770	0.3	No	
flif-hub/flif	CVE-2019-14373	CWE-125	0.3	No	
redis/hiredis	CVE-2020-7105	CWE-476	0.14.0	No	Found issues: 73
mailcore/mailcore2	CVE-2021-26911	CWE-295	0.6.4	No	Found issues: 143
dinhviethoa/libetpan	CVE-2020-15953	CWE-74	1.7.2	No	Found issues: 965
dinhviethoa/libetpan	CVE-2017-8825	CWE-476	1.7.2	No	
leethomason/ti-nyxml2	CVE-2018-11210	CWE-125	6.2.0	Yes	Found issues: 61, Dangerous function call: strlen
eclipse/mosquitto	CVE-2017-7655	CWE-476	1.4.14	No	Found issues: 409
eclipse/mosquitto	CVE-2021-34431	CWE-401	2.0.10	No	Found issues: 874
eclipse/mosquitto	CVE-2018-20145	CWE-732	1.5.4	No	Found issues: 487
eclipse/mosquitto	CVE-2017-7654	CWE-401	1.4.14	No	
strukturag/libheif	CVE-2020-23109	CWE-120	1.6.2	No	Found issues: 161
strukturag/libheif	CVE-2020-19498	NVD-CWE-noinfo	commit fd0c01d	No	Found issues: 158
strukturag/libheif	CVE-2020-19499	CWE-125	1.4.0	No	Found issues: 121
strukturag/libheif	CVE-2019-11471	CWE-416	1.4.0	No	
webmproject/lib-webp	CVE-2018-25012	CWE-125	1.0.0	No	Found issues: 243
webmproject/lib-webp	CVE-2018-25013	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2020-36328	CWE-787	1.0.0	No	
webmproject/lib-webp	CVE-2020-36329	CWE-416	1.0.0	No	
webmproject/lib-webp	CVE-2020-36331	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25010	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25011	CWE-787	1.0.0	No	
webmproject/lib-webp	CVE-2018-25009	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25014	CWE-908	1.0.0	No	
webmproject/lib-webp	CVE-2020-36332	CWE-400, CWE-20	1.0.0	No	
webmproject/lib-webp	CVE-2020-36330	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2016-9085	CWE-190	0.5.1	No	Found issues: 182
webmproject/lib-webp	CVE-2016-9969	CWE-415	0.5.1	No	
google/protobuf	CVE-2015-5237	CWE-787	3.1.0	No	Found issues: 661
libgit2/libgit2	CVE-2020-12278	CWE-706	0.27.2	No	Found issues: 1436
libgit2/libgit2	CVE-2020-12279	CWE-706	0.27.2	No	
libgit2/libgit2	CVE-2018-10887	CWE-190, CWE-125, CWE-681, CWE-194	0.27.2	Yes	The real issue is integer overflow but it leads to an out of bounds read via memcpy

libgit2/libgit2	CVE-2018-10888	CWE-125, CWE-20	0.27.2	Yes	Dangerous function call: memcpy
libgit2/libgit2	CVE-2018-15501	CWE-125	0.27.2	No	
akheron/jansson	CVE-2020-36325	CWE-125	2.13.1	No	Found issues: 111
akheron/jansson	CVE-2016-4425	CWE-20	2.7	No	Found issues: 84
aubio/aubio	CVE-2018-14523	CWE-125	0.4.6	No	Found issues: 223
aubio/aubio	CVE-2018-19802	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19801	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19800	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14522	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14521	CWE-119	0.4.6	No	
aubio/aubio	CVE-2017-17554	CWE-476	0.4.6	No	
aubio/aubio	CVE-2017-17054	CWE-369	0.4.6	No	
facebook/folly	CVE-2021-24036	CWE-190, CWE-122	2019.10.28.00	No	Found issues: 1119
facebook/folly	CVE-2019-11934	CWE-125	2019.10.28.00	No	

II. Testing results for VisualCodeGrepper

Package	**CVE**	**CWE**	**Version tested**	**Found vulnerability**	**Notes**
k-takata/onigmo	CVE-2019-16162	CWE-125	6.2.0	No	Found issues: 501
k-takata/onigmo	CVE-2019-16161	CWE-476	6.2.0	No	
libevent/libevent	CVE-2016-10195	CWE-125	2.1.4	Yes	Found issues: 1063, Dangerous function call: memcpy
libevent/libevent	CVE-2016-10197	CWE-125	2.1.4	No	
libevent/libevent	CVE-2016-10196	CWE-787	2.1.4	Yes	Dangerous function call: memcpy
libevent/libevent	CVE-2014-6272	CWE-189	2.1.4	Partially	Dangerous function call: memcpy
libevent/libevent	CVE-2015-6525	CWE-189	2.1.4	Partially	Dangerous function call: memcpy
libuv/libuv	CVE-2015-0278	CWE-273	0.10.33	No	Found issues: 983
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125	1.12.0	Yes	Found issues: 1883, Dangerous function call: memcpy
libimobiledevice/libplist	CVE-2017-7982	CWE-190	bbd3379	No	Found issues: 166
libimobiledevice/libplist	CVE-2017-5545	CWE-125	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5209	CWE-125	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5835	CWE-770	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5836	CWE-415	bbd3379	No	
libimobiledevice/libplist	CVE-2017-5834	CWE-125	bbd3379	No	
nanopb/nanopb	CVE-2021-21401	CWE-763	0.4.0	No	Found issues: 179
nanopb/nanopb	CVE-2020-26243	CWE-20, CWE-119	0.4.0	No	
nanopb/nanopb	CVE-2020-5235	CWE-125	0.4.0	No	
libssh2/libssh2	CVE-2019-17498	CWE-190	1.4.3	No	Found issues: 603
libssh2/libssh2	CVE-2019-13115	CWE-190, CWE-125	c07bc647f2	No	Found issues: 1111

libssh2/libssh2	CVE-2019-3859	CWE-125	1.4.3	No	
libssh2/libssh2	CVE-2016-0787	CWE-200	d453f4ce3c	No	Found issues: 776
libssh2/libssh2	CVE-2015-1782	CWE-20	1.4.3	No	
jbeder/yaml-cpp	CVE-2019-6285	CWE-674	0.6.3	No	Found issues: 142
jbeder/yaml-cpp	CVE-2019-6292	CWE-674	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-11692	CWE-617	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20573	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20574	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-5950	CWE-119	0.6.3	No	
libimobiledevice/li-busbmuxd	CVE-2016-5104	CWE-284	1.0.10	No	Found issues: 52
aomediacodec/li-bavif	CVE-2020-36407	CWE-787	0.8.1	No	Found issues: 273
flif-hub/flif	CVE-2018-11507	CWE-834	0.3	No	Found issues: 577
flif-hub/flif	CVE-2018-12109	CWE-787	0.3	No	
flif-hub/flif	CVE-2018-14876	NVD-CWE-noinfo	0.3	No	
flif-hub/flif	CVE-2017-14232	CWE-399	0.3	No	
flif-hub/flif	CVE-2018-10971	CWE-770	0.3	No	
flif-hub/flif	CVE-2019-14373	CWE-125	0.3	No	
redis/hiredis	CVE-2020-7105	CWE-476	0.14.0	No	Found issues: 187
mailcore/mailcore2	CVE-2021-26911	CWE-295	0.6.4	No	Found issues: 547
dinhviethoa/libetpan	CVE-2020-15953	CWE-74	1.7.2	No	Found issues: 4752
dinhviethoa/libetpan	CVE-2017-8825	CWE-476	1.7.2	No	
leethomason/ti-nyxml2	CVE-2018-11210	CWE-125	6.2.0	Yes	Found issues: 48, Dangerous function call: strlen
eclipse/mosquitto	CVE-2017-7655	CWE-476	1.4.14	No	Found issues: 749
eclipse/mosquitto	CVE-2021-34431	CWE-401	2.0.10	No	Found issues: 1304
eclipse/mosquitto	CVE-2018-20145	CWE-732	1.5.4	No	Found issues: 821
eclipse/mosquitto	CVE-2017-7654	CWE-401	1.4.14	No	
strukturag/libheif	CVE-2020-23109	CWE-120	1.6.2	No	Found issues: 100
strukturag/libheif	CVE-2020-19498	NVD-CWE-noinfo	commit fd0c01d	No	Found issues: 99
strukturag/libheif	CVE-2020-19499	CWE-125	1.4.0	No	Found issues: 27
strukturag/libheif	CVE-2019-11471	CWE-416	1.4.0	No	
webmproject/lib-webp	CVE-2018-25012	CWE-125	1.0.0	No	Found issues: 1550
webmproject/lib-webp	CVE-2018-25013	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2020-36328	CWE-787	1.0.0	No	
webmproject/lib-webp	CVE-2020-36329	CWE-416	1.0.0	No	
webmproject/lib-webp	CVE-2020-36331	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25010	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25011	CWE-787	1.0.0	No	
webmproject/lib-webp	CVE-2018-25009	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25014	CWE-908	1.0.0	No	

webmproject/lib-webp	CVE-2020-36332	CWE-400, CWE-20	1.0.0	No	
webmproject/lib-webp	CVE-2020-36330	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2016-9085	CWE-190	0.5.1	No	Found issues: 1171
webmproject/lib-webp	CVE-2016-9969	CWE-415	0.5.1	No	
google/protobuf	CVE-2015-5237	CWE-787	3.1.0	No	Found issues: 1255
libgit2/libgit2	CVE-2020-12278	CWE-706	0.27.2	No	Found issues: 5205
libgit2/libgit2	CVE-2020-12279	CWE-706	0.27.2	No	
libgit2/libgit2	CVE-2018-10887	CWE-190, CWE-125, CWE-681, CWE-194	0.27.2	Yes	The real issue is integer overflow but it leads to a out of bounds read via memcpy
libgit2/libgit2	CVE-2018-10888	CWE-125, CWE-20	0.27.2	Yes	Dangerous function call: memcpy
libgit2/libgit2	CVE-2018-15501	CWE-125	0.27.2	No	
akheron/jansson	CVE-2020-36325	CWE-125	2.13.1	No	Found issues: 168
akheron/jansson	CVE-2016-4425	CWE-20	2.7	No	Found issues: 173
aubio/aubio	CVE-2018-14523	CWE-125	0.4.6	No	Found issues: 293
aubio/aubio	CVE-2018-19802	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19801	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19800	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14522	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14521	CWE-119	0.4.6	No	
aubio/aubio	CVE-2017-17554	CWE-476	0.4.6	No	
aubio/aubio	CVE-2017-17054	CWE-369	0.4.6	No	
facebook/folly	CVE-2021-24036	CWE-190, CWE-122	2019.10.28.00	No	Found issues: 1569
facebook/folly	CVE-2019-11934	CWE-125	2019.10.28.00	No	

III. Testing results for CodeQL

Package	**CVE**	**CWE**	**Version tested**	**Found vulner-ability**	**Notes**
k-takata/onigmo	CVE-2019-16162	CWE-125	6.2.0	No	Found issues: 135
k-takata/onigmo	CVE-2019-16161	CWE-476	6.2.0	No	
libevent/libevent	CVE-2016-10195	CWE-125	2.1.4	No	Found issues: 4
libevent/libevent	CVE-2016-10197	CWE-125	2.1.4	No	
libevent/libevent	CVE-2016-10196	CWE-787	2.1.4	No	
libevent/libevent	CVE-2014-6272	CWE-189	2.1.4	No	
libevent/libevent	CVE-2015-6525	CWE-189	2.1.4	No	
libuv/libuv	CVE-2015-0278	CWE-273	0.10.33	No	Found issues: 12
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125	1.12.0	No	Found issues: 306
libimobi-ledevice/libplist	CVE-2017-7982	CWE-190	bbd3379	No	Found issues: 17
libimobi-ledevice/libplist	CVE-2017-5545	CWE-125	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5209	CWE-125	bbd3379	No	

libimobi-ledevice/libplist	CVE-2017-5835	CWE-770	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5836	CWE-415	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5834	CWE-125	bbd3379	No	
nanopb/nanopb	CVE-2021-21401	CWE-763	0.4.0	No	Found issues: 14
nanopb/nanopb	CVE-2020-26243	CWE-20, CWE-119	0.4.0	No	
nanopb/nanopb	CVE-2020-5235	CWE-125	0.4.0	No	
libssh2/libssh2	CVE-2019-17498	CWE-190	1.4.3	No	Found issues: 42
libssh2/libssh2	CVE-2019-13115	CWE-190, CWE-125	c07bc647f2	No	Found issues: 54
libssh2/libssh2	CVE-2019-3859	CWE-125	1.4.3	No	
libssh2/libssh2	CVE-2016-0787	CWE-200	d453f4ce3c	No	Found issues: 56
libssh2/libssh2	CVE-2015-1782	CWE-20	1.4.3	No	
jbeder/yaml-cpp	CVE-2019-6285	CWE-674	0.6.3	No	Found issues: 82
jbeder/yaml-cpp	CVE-2019-6292	CWE-674	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-11692	CWE-617	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20573	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20574	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-5950	CWE-119	0.6.3	No	
libimobiledevice/li-busbmuxd	CVE-2016-5104	CWE-284	1.0.10		configure file gets generated with errors, can't get it to work on the Github runner
aomediacodec/li-bavif	CVE-2020-36407	CWE-787	0.8.1	No	Found issues: 115
flif-hub/flif	CVE-2018-11507	CWE-834	0.3	No	Found issues: 265
flif-hub/flif	CVE-2018-12109	CWE-787	0.3	No	
flif-hub/flif	CVE-2018-14876	NVD-CWE-noinfo	0.3	No	
flif-hub/flif	CVE-2017-14232	CWE-399	0.3	No	
flif-hub/flif	CVE-2018-10971	CWE-770	0.3	No	
flif-hub/flif	CVE-2019-14373	CWE-125	0.3	No	
redis/hiredis	CVE-2020-7105	CWE-476	0.14.0	No	Found issues: 13
mailcore/mailcore2	CVE-2021-26911	CWE-295	0.6.4	No	Found issues: 102
dinhviethoa/libetpan	CVE-2020-15953	CWE-74	1.7.2	No	Found issues: 378
dinhviethoa/libetpan	CVE-2017-8825	CWE-476	1.7.2	No	
leethomason/ti-nyxml2	CVE-2018-11210	CWE-125	6.2.0	No	Found issues: 17
eclipse/mosquitto	CVE-2017-7655	CWE-476	1.4.14	No	Found issues: 143
eclipse/mosquitto	CVE-2021-34431	CWE-401	2.0.10	No	Found issues: 238
eclipse/mosquitto	CVE-2018-20145	CWE-732	1.5.4	No	Found issues: 166
eclipse/mosquitto	CVE-2017-7654	CWE-401	1.4.14	No	
strukturag/libheif	CVE-2020-23109	CWE-120	1.6.2	No	Found issues: 104
strukturag/libheif	CVE-2020-19498	NVD-CWE-noinfo	commit fd0c01d	No	Found issues: 106
strukturag/libheif	CVE-2020-19499	CWE-125	1.4.0	No	Found issues: 109
strukturag/libheif	CVE-2019-11471	CWE-416	1.4.0	No	
webmproject/lib-webp	CVE-2018-25012	CWE-125	1.0.0	No	Found issues: 68

webmproject/lib-webp	CVE-2018-25013	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2020-36328	CWE-787	1.0.0	No	
webmproject/lib-webp	CVE-2020-36329	CWE-416	1.0.0	No	
webmproject/lib-webp	CVE-2020-36331	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25010	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25011	CWE-787	1.0.0	No	
webmproject/lib-webp	CVE-2018-25009	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2018-25014	CWE-908	1.0.0	No	
webmproject/lib-webp	CVE-2020-36332	CWE-400, CWE-20	1.0.0	No	
webmproject/lib-webp	CVE-2020-36330	CWE-125	1.0.0	No	
webmproject/lib-webp	CVE-2016-9085	CWE-190	0.5.1	No	Found issues: 55
webmproject/lib-webp	CVE-2016-9969	CWE-415	0.5.1	No	
google/protobuf	CVE-2015-5237	CWE-787	3.1.0	No	Found issues: 366
libgit2/libgit2	CVE-2020-12278	CWE-706	0.27.2	No	Found issues: 248
libgit2/libgit2	CVE-2020-12279	CWE-706	0.27.2	No	
libgit2/libgit2	CVE-2018-10887	CWE-190, CWE-125, CWE-681, CWE-194	0.27.2	No	
libgit2/libgit2	CVE-2018-10888	CWE-125, CWE-20	0.27.2	No	
libgit2/libgit2	CVE-2018-15501	CWE-125	0.27.2	No	
akheron/jansson	CVE-2020-36325	CWE-125	2.13.1	No	Found issues: 103
akheron/jansson	CVE-2016-4425	CWE-20	2.7	No	Found issues: 82
aubio/aubio	CVE-2018-14523	CWE-125	0.4.6	No	Found issues: 185
aubio/aubio	CVE-2018-19802	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19801	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19800	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14522	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14521	CWE-119	0.4.6	No	
aubio/aubio	CVE-2017-17554	CWE-476	0.4.6	No	
aubio/aubio	CVE-2017-17054	CWE-369	0.4.6	No	
facebook/folly	CVE-2021-24036	CWE-190, CWE-122	2019.10.28.00		Can't get it to compile on GitHub
facebook/folly	CVE-2019-11934	CWE-125	2019.10.28.00		

IV. Testing results for Clang-tidy

Package	CVE	CWE	Version tested	Found vul-nerability	Notes
k-takata/onigmo	CVE-2019-16162	CWE-125	6.2.0	No	
k-takata/onigmo	CVE-2019-16161	CWE-476	6.2.0	No	

libevent/libevent	CVE-2016-10195	CWE-125	2.1.4	Yes	Dangerous function call: memcpy
libevent/libevent	CVE-2016-10197	CWE-125	2.1.4	No	
libevent/libevent	CVE-2016-10196	CWE-787	2.1.4	Yes	Dangerous function call: memcpy
libevent/libevent	CVE-2014-6272	CWE-189	2.1.4	Partially	Dangerous function call: memcpy
libevent/libevent	CVE-2015-6525	CWE-189	2.1.4	Partially	Dangerous function call: memcpy
libuv/libuv	CVE-2015-0278	CWE-273	0.10.33	No	
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125	1.12.0	Yes	Dangerous function call: memcpy
libimobi-ledevice/libplist	CVE-2017-7982	CWE-190	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5545	CWE-125	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5209	CWE-125	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5835	CWE-770	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5836	CWE-415	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5834	CWE-125	bbd3379	No	
nanopb/nanopb	CVE-2021-21401	CWE-763	0.4.0	No	
nanopb/nanopb	CVE-2020-26243	CWE-20, CWE-119	0.4.0	No	
nanopb/nanopb	CVE-2020-5235	CWE-125	0.4.0	No	
libssh2/libssh2	CVE-2019-17498	CWE-190	1.4.3	No	
libssh2/libssh2	CVE-2019-13115	CWE-190, CWE-125	c07bc647f2	No	
libssh2/libssh2	CVE-2019-3859	CWE-125	1.4.3	No	
libssh2/libssh2	CVE-2016-0787	CWE-200	d453f4ce3c	No	
libssh2/libssh2	CVE-2015-1782	CWE-20	1.4.3	No	
jbeder/yaml-cpp	CVE-2019-6285	CWE-674	0.6.3	No	
jbeder/yaml-cpp	CVE-2019-6292	CWE-674	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-11692	CWE-617	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20573	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20574	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-5950	CWE-119	0.6.3	No	
libimobiledevice/li-busbmuxd	CVE-2016-5104	CWE-284	1.0.10	No	
aomediacodec/li-bavif	CVE-2020-36407	CWE-787	0.8.1	No	
flif-hub/flif	CVE-2018-11507	CWE-834	0.3	No	
flif-hub/flif	CVE-2018-12109	CWE-787	0.3	No	
flif-hub/flif	CVE-2018-14876	NVD-CWE-noinfo	0.3	No	
flif-hub/flif	CVE-2017-14232	CWE-399	0.3	No	
flif-hub/flif	CVE-2018-10971	CWE-770	0.3	No	
flif-hub/flif	CVE-2019-14373	CWE-125	0.3	No	
redis/hiredis	CVE-2020-7105	CWE-476	0.14.0	No	
mailcore/mailcore2	CVE-2021-26911	CWE-295	0.6.4	No	
dinhviethoa/libetpan	CVE-2020-15953	CWE-74	1.7.2	No	
dinhviethoa/libetpan	CVE-2017-8825	CWE-476	1.7.2	No	

leethomason/tinyxml2	CVE-2018-11210	CWE-125	6.2.0	No	
eclipse/mosquitto	CVE-2017-7655	CWE-476	1.4.14	No	
eclipse/mosquitto	CVE-2021-34431	CWE-401	2.0.10	No	
eclipse/mosquitto	CVE-2018-20145	CWE-732	1.5.4	No	
eclipse/mosquitto	CVE-2017-7654	CWE-401	1.4.14	No	
strukturag/libheif	CVE-2020-23109	CWE-120	1.6.2	No	
strukturag/libheif	CVE-2020-19498	NVD-CWE-noinfo	commit fd0c01d	Yes	Division by zero, reported with the path the program would take to get there
strukturag/libheif	CVE-2020-19499	CWE-125	1.4.0	No	
strukturag/libheif	CVE-2019-11471	CWE-416	1.4.0	No	
webmproject/libwebp	CVE-2018-25012	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25013	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2020-36328	CWE-787	1.0.0	No	
webmproject/libwebp	CVE-2020-36329	CWE-416	1.0.0	No	
webmproject/libwebp	CVE-2020-36331	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25010	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25011	CWE-787	1.0.0	No	
webmproject/libwebp	CVE-2018-25009	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25014	CWE-908	1.0.0	No	
webmproject/libwebp	CVE-2020-36332	CWE-400, CWE-20	1.0.0	No	
webmproject/libwebp	CVE-2020-36330	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2016-9085	CWE-190	0.5.1	No	
webmproject/libwebp	CVE-2016-9969	CWE-415	0.5.1	No	
google/protobuf	CVE-2015-5237	CWE-787	3.1.0	No	
libgit2/libgit2	CVE-2020-12278	CWE-706	0.27.2	No	
libgit2/libgit2	CVE-2020-12279	CWE-706	0.27.2	No	
libgit2/libgit2	CVE-2018-10887	CWE-190, CWE-125, CWE-681, CWE-194	0.27.2	Yes	Dangerous function call: memcpy
libgit2/libgit2	CVE-2018-10888	CWE-125, CWE-20	0.27.2	Yes	Dangerous function call: memcpy
libgit2/libgit2	CVE-2018-15501	CWE-125	0.27.2	No	
akheron/jansson	CVE-2020-36325	CWE-125	2.13.1	No	
akheron/jansson	CVE-2016-4425	CWE-20	2.7	No	
aubio/aubio	CVE-2018-14523	CWE-125	0.4.6	No	
aubio/aubio	CVE-2018-19802	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19801	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19800	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14522	CWE-119	0.4.6	No	

aubio/aubio	CVE-2018-14521	CWE-119	0.4.6	No	
aubio/aubio	CVE-2017-17554	CWE-476	0.4.6	No	
aubio/aubio	CVE-2017-17054	CWE-369	0.4.6	No	
facebook/folly	CVE-2021-24036	CWE-190, CWE-122	2019.10.28.00	No	
facebook/folly	CVE-2019-11934	CWE-125	2019.10.28.00	No	

V. Testing results for SonarLint

Package	CVE	CWE	Version tested	Found vul-nerability	Notes
k-takata/onigmo	CVE-2019-16162	CWE-125	6.2.0	No	
k-takata/onigmo	CVE-2019-16161	CWE-476	6.2.0	No	
libevent/libevent	CVE-2016-10195	CWE-125	2.1.4	No	
libevent/libevent	CVE-2016-10197	CWE-125	2.1.4	No	
libevent/libevent	CVE-2016-10196	CWE-787	2.1.4	No	
libevent/libevent	CVE-2014-6272	CWE-189	2.1.4	No	
libevent/libevent	CVE-2015-6525	CWE-189	2.1.4	No	
libuv/libuv	CVE-2015-0278	CWE-273	0.10.33	No	
mongodb/mongo-c-driver	CVE-2018-16790	CWE-125	1.12.0	No	
libimobi-ledevice/libplist	CVE-2017-7982	CWE-190	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5545	CWE-125	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5209	CWE-125	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5835	CWE-770	bbd3379	No	
libimobi-ledevice/libplist	CVE-2017-5836	CWE-415	bbd3379	Yes	Recommends that dynamic heap memory allocation should not be used
libimobi-ledevice/libplist	CVE-2017-5834	CWE-125	bbd3379	No	
nanopb/nanopb	CVE-2021-21401	CWE-763	0.4.0	No	
nanopb/nanopb	CVE-2020-26243	CWE-20, CWE-119	0.4.0	No	
nanopb/nanopb	CVE-2020-5235	CWE-125	0.4.0	No	
libssh2/libssh2	CVE-2019-17498	CWE-190	1.4.3	No	
libssh2/libssh2	CVE-2019-13115	CWE-190, CWE-125	c07bc647f2	No	
libssh2/libssh2	CVE-2019-3859	CWE-125	1.4.3	No	
libssh2/libssh2	CVE-2016-0787	CWE-200	d453f4ce3c	No	
libssh2/libssh2	CVE-2015-1782	CWE-20	1.4.3	No	
jbeder/yaml-cpp	CVE-2019-6285	CWE-674	0.6.3	No	
jbeder/yaml-cpp	CVE-2019-6292	CWE-674	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-11692	CWE-617	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20573	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2018-20574	CWE-119	0.6.3	No	
jbeder/yaml-cpp	CVE-2017-5950	CWE-119	0.6.3	No	
libimobiledevice/li-busbmuxd	CVE-2016-5104	CWE-284	1.0.10	No	

aomediacodec/libavif	CVE-2020-36407	CWE-787	0.8.1	No	
flif-hub/flif	CVE-2018-11507	CWE-834	0.3	No	
flif-hub/flif	CVE-2018-12109	CWE-787	0.3	No	
flif-hub/flif	CVE-2018-14876	NVD-CWE-noinfo	0.3	No	
flif-hub/flif	CVE-2017-14232	CWE-399	0.3	No	
flif-hub/flif	CVE-2018-10971	CWE-770	0.3	No	
flif-hub/flif	CVE-2019-14373	CWE-125	0.3	No	
redis/hiredis	CVE-2020-7105	CWE-476	0.14.0	Yes	Recommends that dynamic heap memory allocation should not be used
mailcore/mailcore2	CVE-2021-26911	CWE-295	0.6.4	No	
dinhviethoa/libetpan	CVE-2020-15953	CWE-74	1.7.2	No	
dinhviethoa/libetpan	CVE-2017-8825	CWE-476	1.7.2	No	
leethomason/tinyxml2	CVE-2018-11210	CWE-125	6.2.0	No	
eclipse/mosquitto	CVE-2017-7655	CWE-476	1.4.14	No	
eclipse/mosquitto	CVE-2021-34431	CWE-401	2.0.10	Yes	Recommends that dynamic heap memory allocation should not be used
eclipse/mosquitto	CVE-2018-20145	CWE-732	1.5.4	No	
eclipse/mosquitto	CVE-2017-7654	CWE-401	1.4.14	No	
strukturag/libheif	CVE-2020-23109	CWE-120	1.6.2	No	
strukturag/libheif	CVE-2020-19498	NVD-CWE-noinfo	commit fd0c01d	No	
strukturag/libheif	CVE-2020-19499	CWE-125	1.4.0	No	
strukturag/libheif	CVE-2019-11471	CWE-416	1.4.0	No	
webmproject/libwebp	CVE-2018-25012	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25013	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2020-36328	CWE-787	1.0.0	No	
webmproject/libwebp	CVE-2020-36329	CWE-416	1.0.0	No	
webmproject/libwebp	CVE-2020-36331	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25010	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25011	CWE-787	1.0.0	No	
webmproject/libwebp	CVE-2018-25009	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2018-25014	CWE-908	1.0.0	No	
webmproject/libwebp	CVE-2020-36332	CWE-400, CWE-20	1.0.0	No	
webmproject/libwebp	CVE-2020-36330	CWE-125	1.0.0	No	
webmproject/libwebp	CVE-2016-9085	CWE-190	0.5.1	No	
webmproject/libwebp	CVE-2016-9969	CWE-415	0.5.1	Yes	Recommends that dynamic heap memory allocation should not be used
google/protobuf	CVE-2015-5237	CWE-787	3.1.0	No	
libgit2/libgit2	CVE-2020-12278	CWE-706	0.27.2	No	
libgit2/libgit2	CVE-2020-12279	CWE-706	0.27.2	No	

libgit2/libgit2	CVE-2018-10887	CWE-190, CWE-125, CWE-681, CWE-194	0.27.2	No	
libgit2/libgit2	CVE-2018-10888	CWE-125, CWE-20	0.27.2	No	
libgit2/libgit2	CVE-2018-15501	CWE-125	0.27.2	No	
akheron/jansson	CVE-2020-36325	CWE-125	2.13.1	No	
akheron/jansson	CVE-2016-4425	CWE-20	2.7	No	
aubio/aubio	CVE-2018-14523	CWE-125	0.4.6	No	
aubio/aubio	CVE-2018-19802	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19801	CWE-476	0.4.6	No	
aubio/aubio	CVE-2018-19800	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14522	CWE-119	0.4.6	No	
aubio/aubio	CVE-2018-14521	CWE-119	0.4.6	No	
aubio/aubio	CVE-2017-17554	CWE-476	0.4.6	No	
aubio/aubio	CVE-2017-17054	CWE-369	0.4.6	No	
facebook/folly	CVE-2021-24036	CWE-190, CWE-122	2019.10.28.00	No	
facebook/folly	CVE-2019-11934	CWE-125	2019.10.28.00	No	

VI. License

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

I, Karl Jääts,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis "How well could have existing static vulnerability detection tools prevented publicly reported vulnerabilities in iOS open source packages", supervised by Kristiina Rahkema.

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

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

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

Karl Jääts
15/08/2023