Universiteit Utrecht

A Competition Analysis of Software Assurance Tools

Author
Ashot A. Grigorian

Student number:
6501435

Supervisor
Dr. Slinger Jansen

Second Supervisor
Dr. Gerard Wagenaar

A Bachelor's thesis submitted in fulfilling the
requisites for the Bachelor of Science degree in the
Department of Information and Computing Sciences

August 2022

Abstract

In the production process of a software system, it is impossible to manually ensure that no vulnerabilities, license conflicts, or improper code cloning takes place. For this reason, several tools that automate such processes have been introduced in the market as software assurance tools, and enable software-producing organizations to create reliable and secure software systems. In this study, the market for these tools is explored by performing a gap analysis to unearth business opportunities for a new software assurance tool. The market is explored through a systematic snowballing procedure methodology to find more on current literature available in the domain to ultimately compose a requirement comparison matrix that surveys existing tools. Following this study, it is found that although the market is crowded, there are several business opportunities for a new software assurance tool to be introduced. By focusing on aspects that are uncommon in these tools, and therefore expanding the set of functionalities offered by the tool, opportunities arise to outperform the competition. Also offering the tool, or a part of the tool, for no cost would help achieve the successful introduction of a new software assurance tool.

Keywords: Software Assurance Tools, Source Code, Tool Survey, Comparison Matrix, License Tracking, Vulnerabilities.

Contents

Chapter 1. Introduction

Over the past half-century, society has witnessed rapid changes and technological developments. The increased reliance on computers is just a single point in the so-called landscape of change that our world is undergoing [2]. This reliance on computers is known to be the result of technological innovations in telecommunications and computer networking. With current society making more use of computers to share data because it is more time- and cost-effective in comparison to physically sending mail, there is an inevitable increase of data that is stored online. Reportedly, Google, Amazon, Microsoft, and Meta (formerly known as Facebook) share at least 1,200 petabytes of data between them as a result of digitization: *"digitalization embraces the ability of digital technology to capture and assess data to make better business decisions and enable new business models"* [NP4]. That excludes other giant industries in the field of storing and processing information such as GitHub (who surpassed the 10 million repositories in 2013 [NP7] and in 2020 surpassed the 128 million mark [NP13]), Dropbox, Barracuda, and SugarSync, not to mention the servers in bulk in industry and academia [NP15]. With these massive amounts of data being stored online which society relies on, in everyday tasks for a business or private use, it is safe to conclude we find ourselves in an information-based society [20].

These quantities of raw, unprocessed, data, hold minimal value for most. However, most if not all of this raw data is maintained and processed by chunks of code that can operate accordingly for different purposes. With these chunks of code coming together in software products it becomes clear how dependent society is on high-quality, sustainable software to efficiently process the data. Within software production, different approaches are known for producing tools. Most common for this are the waterfall and agile methodologies. With the waterfall method being more sequential and the agile method resulting in an iterative process, different means of building software are available that are minimally look-alike [25].

Nonetheless, whichever method is used for producing the software, an incremental phase of both that cannot be overlooked is assuring the software is of good quality, which is where code assurance comes into play. Code assurance, also known as Software Quality Assurance (SQA) is a supportive process and *"a means and practice of monitoring the software engineering processes and methods used in a project to ensure proper quality of the software"* [7]. Depending on the tool, the SQA process is carried out in certain manners to detect, assess and resolve software defects, commonly referred to as vulnerabilities [1]. By resolving these vulnerabilities and preventing further occurrences in software tools, a desired cyber security level can be reached to prevent malicious users from bypassing security regulations. The SQA is most often carried out by Software Assurance Tools, often referred to as an SAT or SATs within this study. Multiple definitions are in place for SATs with three being mentioned to show their relatedness regarding uncovering weaknesses to assess and eventually increase the security level of a project:

> *1) "Software assurance tools are a fundamental resource for providing an assurance argument for today's software applications throughout the software development lifecycle (SDLC). Software requirements, design models, source code, and executable code are analyzed by tools in order to determine if an application is secure."* [6].
>
> *2) "Software assurance tools – tools that scan the source or binary code of a program to find weaknesses – are the first line of defense in assessing the security of a software project."* [21].

> *3) "Software assurance SwA may be defined as the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in the intended manner. Since modern systems are under constant attack, sufficient SwA is vital."* [33].

Note that the third mentioned definition abbreviates software assurance as SwA, while this study abbreviates software assurance as SA.

The most common vulnerabilities in software development are spread by code cloning, therefore code cloning is an issue to address: *"Vulnerabilities in software will not only lead to security problems of the software itself, but also cause the spread of vulnerabilities through code clones."* [31]. Code cloning represents similarly structured or completely duplicated code [15]. Software tools typically contain between 9%-17% cloned code [15], however, at times it is found that software consists of more than 50% cloned code. Code cloning causes issues such as degraded source code integrity, and code inflation. Another finding shows that the vulnerability density in cloned code is not notably higher as compared to non-cloned code [15]. However, the weaknesses in cloned code are significantly more high-risk as opposed to those in non-cloned code. Such risks may be more redundant code, higher maintenance costs, and fewer modular systems [18].

To tackle vulnerabilities in code, and assure high code quality, it seems intuitive to first tackle the issues that pose a higher priority compared to lower-risk vulnerabilities. Since cloned code contains a higher density of vulnerabilities, tackling the frequency and proportions of cloned code occurrences holds a higher priority. By lowering the density of cloned code, the vulnerability density in a software tool is addressed as well. To be able to lower the density of cloned code, specific software tools are needed to detect such chunks of cloned code. However, within the software industry, such tools are already available. But why is the issue of code cloning still at large unresolved then? Why are the current tools for software assurance not deemed effective enough in tackling the issue of high-risk vulnerabilities? What requirements should code-cloning detection tools meet to successfully tackle the mentioned issue? In this research, we explore the problem where it is insufficiently clear how this niche market can be addressed. Accordingly, the following question is pursued in this case study:

How can software assurance tools be compared?

To find an answer to this research question, an elaborate overview of the research methods applied in this research project is discussed, which includes the procedure, utilized databases, search terms, and inclusion and exclusion criteria. This is based on sub-questions to give the study more direction. Furthermore, the criteria are set in place to find out more about the goals, features, and qualities of SATs after which a comparison matrix is discussed to compare the SATs. Then, SATs are identified and then analyzed individually, before comparing these tools to find strengths and weaknesses compared to their competition.

Chapter 2. Research Methods

2.1 Research Process

The moment various tools are being compared, it should be clear why those exact tools are taken into comparison. Within the context of comparing software tools, a comparison could be made for several attributes such as features, market availability, business models, architectural frameworks, performances, or risks. Studying software assurance tools is no exception. These software products need to be identified using a specific mapping, but it also needs to be clear up front what this mapping is based on, which is where literature research plays its role.

For any product to be compared to its competition, it must be clear what the product is designed for. Accordingly, among others, G2, one of the world's largest tech marketplaces, pursues an answer to the following [NP9]: *"There are tens of thousands of applications on the market. How do you choose the right one?"* At first, the role that an upcoming software assurance tool can fulfill in its respective software ecosystem (SECO) is unclear. To study this role, it is needed to undertake a sequence of research tasks to retrieve the required information for this exploration: It must become clear what is expected of tools or platforms that are designed to detect the high-risk vulnerabilities that come with duplicated code. Among others, this study carries out a systematic snowballing procedure to establish the main goals, challenges, and requirements posed for such software tools that aim to assure code quality. After, aspects of SATs will be discussed to establish a comparison matrix for the review of the selected SATs.

This research study pursues answers to the set of research (sub-)questions mentioned below. Afterward, these are elaborated one by one.

Main Research Question: *"How can software assurance tools be compared?"*

Sub-question 1: *"What role does SearchSECO attempt to play in the software production process?"*

Sub-question 2: *"What identifies competitors of the SearchSECO tool?"*

Sub-question 3: *"What features and qualities does the SearchSECO tool offer that are unique compared to existing software assurance tools?"*

After establishing the goals, challenges, and features for such tools, the features of an existing tool-in-development are evaluated, which in the context of this case study is the SearchSECO tool. By mapping what tasks the SearchSECO tool executes, why and how these are executed, and to what extent the execution of these tasks delivers the desired result, one can analyze its market opportunities. Along with gathering information by performing this process, Subquestion 1 is answered: *"What role does SearchSECO attempt to play in the software production process?"*

Since there is a variety of tools available on the market to improve code quality, uncertainty is present among professionals as to which tools to embrace and which to overlook. For this, SATs must be studied in detail to form an unambiguous understanding of what features and qualities are to be expected in such tools. With this, the competition of SearchSECO is defined. More specifically, formulating a distinction between direct, and non-competition helps to point in the direction of mapping competitive tools. In other words, the target is to identify competitors of SearchSECO for which Subquestion 2 arises: *"What identifies competitors of the SearchSECO tool?"* This is also promoted by introducing inclusion and exclusion criteria in Subchapter 2.5.

After having identified the direct competition of SearchSECO, the process of analyzing these separately as done with SearchSECO starts, before comparing these tools against each other. With this comparison, the goal is to distinguish strengths from weaknesses. With this, Subquestion 3 is formulated: *"What features and qualities does the SearchSECO tool offer that are unique compared to existing software assurance tools?"*. To put the process in an orderly fashion the following steps, as listed below, are executed sequentially. Of this list, the first-mentioned step, namely, Step 0) represents this current chapter. Furthermore, Steps 3) and 4) are performed simultaneously.

```
0) Select research methods
1) Conduct a literature review on software assurance tools
2) Identify features and developments within SearchSECO
3) Establish inclusion and exclusion criteria
4) Define when a tool is a competitor to SearchSECO
5) Determine the competitors of SearchSECO
6) Map features and developments within competitors
7) Compare competing tools with one another and SearchSECO
8) Identify strengths and weaknesses of SearchSECO
```

After these steps are completed, the strategic position of SearchSECO is discussed to indicate the market opportunities that lay ahead for SearchSECO. To discuss this, the strengths and weaknesses are mapped to point out the attributes which the SearchSECO tool stands out with, or where adjustments could be beneficial. These are based on the comparison which is materialized by mapping the features and developments of competitors.

2.2 Systematic Snowballing Procedure

The technique used within this study to review literature is the systematic snowballing procedure. This procedure helps with *"mechanically searching individual databases for the given topic"* [29]. The initial step is to find a starting set of scientific papers which can be used for the systematic snowball procedure [34]. The difficulty of this step is identifying whether certain papers find themselves inconjunct or disjunct clusters of papers to not obstruct the effect of the snowballing procedure. Furthermore, the initial set of papers should not be too small, and not too large either. By adjusting the keywords used in the search, the number of results can be adjusted accordingly for a diverse set of papers. The search terms of this study are specified in Subchapter 2.4 and are used to form the initial set of papers with which the systematic snowballing procedure is carried out. By applying these means a literature review can be conducted to base the inclusion and exclusion criteria on, which are used for the tool review.

2.3 Database

The online search engines used in the process of conducting this review consisted of those identified as relevant to the field of education, information technology, and cyber security: Google (search engine of Google LLC), and Google Scholar (a freely accessible internet search engine that indexes the full text or metadata of academic literature in a range of publication formats and disciplines, as part of Google LLC). Even though both Google and Google Scholar are mentioned, it is mostly Google Scholar that is utilized for more academic search results. In addition to these two mentioned web search engines, academic closed database search systems are also used. Such databases identified for this review include SpringerLink and IEEE Xplore.

2.4 Search Terms

As aforementioned, what is demanded of solutions intended for uncovering the high-risk vulnerabilities associated with cloned code must be made apparent. The literature study is no different from this. Therefore, search terms are defined beforehand to find out more about the goals, challenges, and features of code cloning detection software. For each of these three subtopics a specific search query is composed, to be able to address each topic separately, referred to as SQ1, SQ2, and SQ3 respectively. By using the results of the queries, and applying the systematic snowballing procedure, the triplet of topics is mapped in a reproducible manner.

The first search query is applied to map the goals, or rather objectives of such tools, which focuses on tackling the previously introduced high-risk vulnerabilities in a more in-depth manner. This subject is addressed by using terms such as Goal, Vulnerability, Code Duplication, Tool, and Security. To prevent the exclusion of relevant articles that phrase these terms differently, the most similar synonyms of these terms were also added in the query as identified by (digital) urban dictionaries [NP37]. However, upon testing this query on the mentioned databases, a significant portion of the results did not pursue the subject this study focuses on. Consequently, specific terms have been excluded that hold no value for this literature framework to refine the search.

SQ1: The main objectives of software assurance tools

("Goal" OR "Objective" OR "Tackle") AND ("Vulnerability" OR "Exposure" OR "Intrusion" OR "Threat" OR "Openness") AND ("Code Duplication" OR "Code Cloning") AND "Detection" AND ("Tool" OR "Software" OR "Product" OR "System") AND ("Security" OR "Digital" OR "Computer" OR "Cyber") -Psychology -Communication -Biology -Behaviour

The second query is composed similarly to the query for studying the main objectives and is based on the following four search terms: Challenges, Code, Duplication, and Tool. Therefore, the same terms are excluded and the query is partially identical. For finding a starting set of literature on the main challenges for the tools in question, the following search query is applied:

SQ2: The main challenges of software assurance tools

("Challenge" OR "Task" OR "Obstacle" OR "Problem" OR "Issue") AND ("Code Duplication" OR "Code Cloning") AND "Detection" AND ("Tool" OR "Software" OR "Product" OR "System") AND ("Security" OR "Digital" OR "Computer" OR "Cyber") -Psychology -Communication -Biology -Behavior

At last, the functional, and non-functional requirements within code duplication detection tools are discussed. This subtopic discusses, for the most part, the back-end requirements of this category of tools. To find suitable studies that explore the required technologies for successful code duplication detection, the following keywords are used: Feature, Code, Duplication, and Tool. By adhering to the same criteria as the other two discussed queries, the following query is the result:

SQ3: The main requirements of software assurance tools

("Feature" OR "Functionality") AND "Code Assurance" AND ("Tool" OR "Software" OR "Product" OR "System") AND ("Security" OR "Digital" OR "Computer" OR "Cyber") -Psychology -Communication -Biology -Behavior

2.5 Inclusion and Exclusion Criteria for the Tool Survey

With the previous subchapters discussing the literature study, this subchapter aims at defining criteria to which tools have to adhere to, to be included in the tool review. The criteria mentioned in this subchapter are further elaborated on in Chapter 6 based on findings from the literature. Still, this subchapter provides an overview beforehand to indicate the direction taken within this project.

At first, the three inclusion criteria are mentioned and described. Then, four exclusion criteria are mentioned. While the first two exclusion criteria are related to being able to study the tools, the latter two exclusion criteria focus on selecting the appropriate tools.

To be included in the review of SATs, **(i1)** the tool must be similar to SearchSECO. This similarity is defined by both the requirements identified for SATs, and by the main goals to achieve for SATs, specifically SearchSECO. Based on the attributes of SearchSECO, which are described in Chapter 5, similar tools can be retrieved and compared. **(i2)** The selected tools for comparison should also be easily and directly available, along with being **(i3)** fully available on the market with documentation. These criteria *(i2)* and *(i3)* are in place to be able to completely assess the SAT within a reasonable amount of time, which is defined as approximately half of one week per tool.

As a consequence of convenience sampling, the criterion is further specified to **(e1)** exclude tools not available in the market for software assurance in the Netherlands, and **(e2)** exclude tools that cannot be tested without a sales representative; to prevent external influences.

Further, following a study by Bendinskas et al. on mature software processes [5], they claim that *"the higher maturity results in cost savings, reduced time-to-market, enhanced quality, and improved predictability in meeting cost and schedule estimates."* This claim derives from the work by the Software Engineering Institute (SEI) [27], which defined five maturity levels (CMM) and to what extent these affect the costs of development [5]. Bendinskas et al. defined how long it takes to transition between the five maturity levels [5]: level 1 to 2 in 22 months; level 2 to 3 in 19 months; level 3 to 4 in 25 months; level 4 to 5 in 13 months. However, Eickelmann [11] shows that no change in whether the trends of cost as a percent of development for prevention, appraisal, internal and external failure, and total cost of software quality decrease or increase can be noticed after turning to maturity level 3. Therefore, after reaching maturity level 3, these trends are reinforced, but not changed, enhancing predictability and maturity.

To reach maturity level 3, a software product would need to be approximately 22+19 months old, in other words, it would be aged 3.4 years. For convenience, this is rounded to 3.5 years. To create a representative view of the software assurance tool market, it is more appropriate to consider matured products. The less matured tools with renewed techniques and methods are yet to prove to be considered as competition for the other tools in the market of SATs for the forthcoming term. This study, therefore, **(e3)** does not take into account SATs aged less than 3.5 years.

At last, **(e4)** tools that are not in direct competition are left out of the survey. The tools taken into comparison should be considered direct competition within the SECO of SearchSECO. For what is considered direct or indirect competition, the following definition is used: *"Theorists have started to explore competitive behavior that minimizes rivalry and head-to-head confrontation—a competitive style referred to as "indirect competition.""* [8]. In other words, tools minimizing rivalry and confrontation within the software ecosystem are not tools to take into consideration.

2.6 Comparison Matrix

To carry out the tool review, a matrix will be composed to compare the different aspects of the tools taken into comparison. Chapter 4 focuses on different aspects of software assurance tools, which are used to create a framework for the matrix used in this study. Based on the literature study carried out, a substantiation is given for the composition of the comparison matrix. As a dependency of the matrix, it becomes clear how to determine when a certain requirement is met. The literature helps to identify whether a particular tool, for a particular requirement, receives a checkmark or not. However, it could also suggest using another scale, with multiple levels where a distinction is made between not, partially, or fully meeting a criterion. In other words, the chosen matrix and the documentation of that matrix support the decision-making process when there is a lack of certainty about when a condition is satisfactorily met. A listing of the matrix itself is given accordingly, in Chapter 4.6. Accordingly, each aspect taken into comparison must be studied for every included tool. This analysis will be carried out using either documentation of the tool or testing the tool itself.

Chapter 3. Goals and Challenges

For studying software assurance tools (SATs), this chapter elaborates on the main goals and challenges, with Chapter 4 focusing on the requirements of such tools respectively. Along with the requirements, research will be performed on existing matrices to compare the features. First, an initial set of papers is discussed which is retrieved by its respective search query as visualized previously using search queries 1, 2, and 3. Along with the findings of the initial set of papers, findings from the systematic snowballing procedure are covered to attain a sound understanding of the matter.

While engineers and researchers must be protected from plagiarism, end-users must be protected from harmful and malicious intentions. The same holds for application and software project development. In other words, one should not be enabled to take away from the man-hours put in by authentic developers, which is where SATs become operative. In the past years, code, application development, and mobile phone sales have seen inflation [15][10]. To grasp the risks that code cloning carries, besides plagiarizing the work of others, the three laws of software development as defined by Holzmann must be introduced [14]:

1) Software tends to grow with time, whether or not there is a rational need for it;
2) All nontrivial code has defects;
3) The probability of non-trivial defects increases with code size.

Holzmann has substantiated the first law by describing the growth of the scripts for the commands *true* and *false* in Unix® and Unix-based systems. As visualized in Figure 1, the size of one of the most simple commands for computer systems has grown exponentially, without changing functionality.

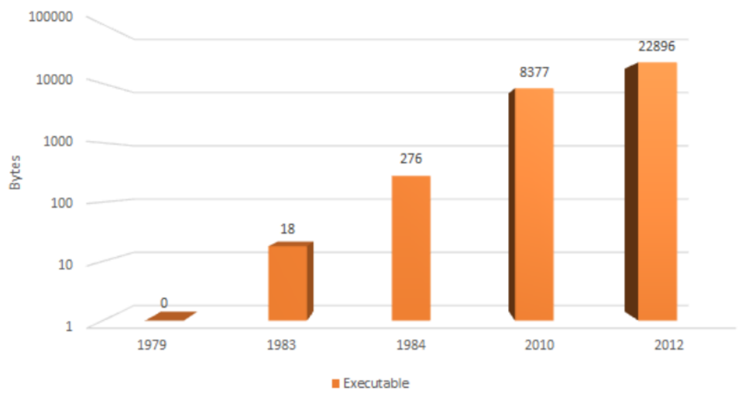


Figure 1. The size of /bin/true in Unix® and Unix-based systems [14]

This increase in executable code for the commands *true* and *false* does not give other results of implementation, yet more code is used. Although for other code, this does potentially make a difference since for certain features, it becomes unclear what the purpose is of certain parts of code to have it successfully implemented. Where this is the case, the code is considered *"dark code"*. Yet, the code cannot be removed without disrupting the implementation [14]. Instances of clear code cloning, and occurrences of dark code, may hurt the overall quality, maintainability, and readability of software [4][S2]. In 2011, members of the University of California presented DNADroid [10], *"a tool that detects copying, or "cloning", by robustly computing the similarity between Android applications"*. This tool aims at helping create *"a healthy market environment to* *encourage developers to continue creating applications."* They state that robust techniques are required to tackle code cloning in Android applications. To do so, the following definition is given as to what a code clone stands for:

Definition of *"Clone"*: *Clones occur when two applications (1) have similar code but (2) have different ownership* [10]. This definition will be used going forward in this report.

Code duplication is as much of an issue to end-users as it is to original application owners. Here, both are discussed, starting with the latter. As a result of cloning, developers could lose out on potential earnings [10]. First, the application, if available for a small payment, might be *"cracked"* and published for no cost, or re-published with certain edits made to important libraries redirecting the cash flow towards the plagiarist. Even republishing while crediting the original owner might have the owner lose out on earnings. This can be realized by adjusting the original developer's client ID. Often, these changes are made in a combination with dark code to increase the difficulty of removal [14]. All of these official permissions or permits to do, use, or own the application are mentioned within its license. Breaking the terms of such licenses is considered illegal, and with application and advertisement revenue both belonging to the application owner, this is a significant issue to tackle.

For end-users, however, the danger could be argued to be at least as bad or even worse. These vulnerabilities could lead to the implementation of malware or ransomware in the application, which spreads to the device used to run the code. This problem of software vulnerabilities is magnified by the life cycle of these exploits. That is, when a provider is to release a patch after the detection of a weakness in the native application, it still consumes time before this is fully deployed to all end-users using the application [S1]. The delay between the release of the patch and reaching the user increases when more are involved in the application development. For example [22]: *"attackers were able to successfully deploy the "Dogspectus" ransomware to a large number of Android smartphones in 2016 by exploiting a two-year-old vulnerability in the kernel (CVE-2014-3153) located in the firmware."* This was made possible by the fact that Google uses the Linux kernel to create the Android OS, and vendors use this OS to create firmware for smartphones. When the vulnerability was detected in the Linux kernel, there was a significant lag before this vulnerability was fixed. Therefore, a goal of SATs would be to have a short period in which security patches can be deployed to reach the end-users to minimize the damage that vulnerability exploits potentially lead to.

While the frequency and the risk level of vulnerabilities increase as detected by software assurance tools [10], so do new types of detection evasion techniques emerge [S3]. As coined by Sun Tzu (a Chinese general, military strategist, writer, and philosopher) [NP1]: *"If you know yourself but not the enemy, for every victory gained you will also suffer a defeat."* To paraphrase and put it into context for this research, If SAT developers fail to keep up, and cloned or vulnerable code can stay undetected, in a matter of time more harm will be done. Therefore, this issue needs to be addressed now. SAT developers have to keep up with malicious intenders, which is realized by developing techniques to be effective in resisting the emerging types of detection evasion techniques. If the developers fail to keep up, and the malicious plagiarists can stay undetected, in a matter of time new detection evasion techniques will arise. Tackling the detection evasion techniques is considered one of the main goals for SATs. Along with discussing these techniques, the challenges when it comes to resisting detection evasion techniques are also of value, together with the features that are required to do so within SATs.

As aforementioned, it remains a goal to fight the evasion techniques that plagiarists and those who mean harm apply. A plagiarist will most likely modify the cloned code to evade any form of detection, and consequently, challenges lay ahead for SATs. The previously introduced tool, DNADroid, is designed to resist the following five evasion techniques [10]:

1) High-level modifications: modified packages, classes, methods, and variable names as well as the addition or deletion of classes and methods.
2) Method Restructurings: relocating methods between connected classes, splitting these methods into multiple methods, or reverse.
3) Control Flow Alterations: Swapping if and else branches or for loops and while loops in code.
4) Addition/Deletion: Adding and deleting parts to or of the original code to make retracing harder.
5) Reordering: Reordering code segments.

However, with not all tools being designed for tackling the evasion techniques listed above, it remains a challenge to resist these for all software projects. But why take on the challenge and try to fight developers who clone code? To understand this, it must become clear that software requires maintenance periodically, or a system redesign for more optimal results. However, with code clones, lots of redundant code is included, which complicates the maintenance. Even redesigns could become complicated when there is no clear view of how features were implemented previously [26] [S4].

Chapter 4. Requirements for Software Assurance Tools

To discuss features for applications, it must become clear beforehand what is meant by the noun 'Feature'. Rather than following in the footsteps of many others, this study distinguishes functional and non-functional requirements by using the definition of S. Farshidi et al.: *"A software tool includes two primary sets (Qualities and Features). The set Qualities is a set that contains software quality attributes, and the set Features is a set that consists of domain features"* [12]. From this, one can extract that there are two primary sets of requirements: the functional (features), and non-functional (qualities).

In the context of this study, a feature is defined to be a functional requirement necessary for either managing a software project within the SAT, identifying the source code within the project, or constructing the source code model. Within this work, the terms 'Feature', 'Functionality', and 'Functional requirement' will be used synonymously. Nonetheless, not only functionalities are of importance to studying a software tool. Along with the functional requirements, attention will be paid to the qualities of the software, such as the number of different programming, scripting, and markup languages a certain SAT can handle.

Depending on the software assurance tool, different means are used to reach the goal of assuring code is of good quality and meets the lawful terms and conditions the original code owners state. But which steps lay ahead for software assurance tools? What features do these tools have in common?

Proposed by Craft et al. is a novel software application, the Source Code Assurance Tool to *"assist a system analyst in the software assessment process"* [9]. Rather than discussing how the Source Code Assurance Tool is designed to achieve its goals, its draft plan with various steps to take is discussed extensively; the goals set and different requirements defined by Craft et al. for the Source Code Assurance Tool provide more context to work with when compared to the means to reach those exact goals. Within this study, a job is defined as work that needs to be done; this definition holds for analyzing a software project and assessing its quality and risks. This analysis includes multiple tasks, i.e. multiple pieces of work that need to be done. Among others, Craft et al. included tasks on how to identify code [9]. Discussing the requirements proposed in this step will be followed by the requirements for its source code model, transforming code using the source code model, and further qualities an SAT would benefit from.

For each requirement, multiple outcomes are described with the first outcome being the least advanced outcome, and the last outcome being the more advanced outcome, and therefore likely to be more preferred. Nevertheless, the later requirements require more enhanced engineering and would end up more time-consuming, therefore being the more costly option to take into consideration when designing a software assurance tool. Consistently, this order is maintained for the forthcoming functionalities and their respective outcomes.

4.1 Identifying code

To be able to identify source code certain prerequisites should be stipulated beforehand. Still, in the broader process of securing a program-based system the assessment of code is only a single piece to consider. Craft et al. mention [9] that as soon as issues are spotted, it is the program analyst's and developers' responsibility to find and define alternative strategies for protecting the software system. One strategy for accomplishing this is by creating multiple, differing configurations of the same project, each aimed at solving different security issues. These configurations can then be compared to assess the pros and cons of certain configurations to compose an improved configuration. Although not every software assurance tool works in this manner because of its time-consuming and therefore costly aspects, it is most effective to analyze a certain issue using this form of triangulation where multiple configurations of the same software project are analyzed.

Within each system configuration, multiple source code files are present. These can be considered units of analysis, also referred to as program units. When certain files are compiled together, these files form a single program of a system. With the identification of system configurations, program units within those configurations, and the source code files within the program units, one can dig into the metalevel a certain SAT can handle. Not only does this affect the complexity, but also the usability of the system. Being enabled to work with certain configurations of a software project rather than the software project as a whole helps with interoperating with parts of configurations to work towards an improved implementation. This leads to a series of identifications:

- **Identifying system configurations in the project:**

Two levels of complexity can be identified regarding system configurations when wanting to implement this feature into the SAT: **1)** The SAT is not able to identify and distinguish different system configurations. **2)** The SAT supports adding, deleting, and renaming one or more configurations to a specific project.

In practice, the outcomes for program units and source code files are effectively identical to those of configurations, with the only difference being the hierarchical level one is working with. So to say, the configurations include the program units, and the program units include the source code files. Most important regarding these identifications is the ability or inability of the SAT to work on different levels of a software project; therefore the outcomes of the following two requirements are identical to the requirement of the system configurations. Important to note is that alterations on any of the three levels affect the analysis the SAT carries out, since, for example, a minimalistic alteration on the level of source code files affects a system configuration as a whole.

Regarding program units and source code files, the same levels can be achieved for system configurations. Either it is, or is not possible for the SAT to identify and distinguish, and therefore analyze different program units and source code files.

- **Identifying program units in the system configurations.**
- **Identifying source code files in the program units.**

- **Compatibility with software and platforms:**

When working on the implementation of the previous features, where the SAT can distinguish configurations, program units, and individual files from one another, one could go even further. Nowadays, certain code processing and version-control software and platforms, such as Microsoft Visual Studio or GitHub can already distinguish these [9]. While compatibility with platforms is a quality rather than a feature, it is quite important to take into account when designing an SAT. However, it can also be decided **1)** not to implement compatibility with software and platforms designed to work with system configurations. On the other hand, SAT developers could choose to work on **2)** making the SAT compatible with other software and platforms.

4.2 The source code model

After working on or skipping the trajectory of distinguishing system configurations from program units and individual files, the tool must continue with building the source code model. The source code model stands for the illustration of the source code, i.e. individual files, in a generalized manner. From this point onwards, the source code does not need to be comprehensive any longer. One does not need to understand what the original code represented. Rather, methods and classes are rewritten in such a manner that these can be compared to other methods and classes in search of similarity in terms of functionality and purpose. To reform source code, multiple aspects of the code should be taken into account to enable the SAT to both rewrite the code and then analyze the project. For constructing a source code model of a software project, the following aspects are identified by Craft et al. [9]:

- **Extracting the code's structure:**

At first, it should become clear how the code is structured. With software code being written using a combination of classes and methods which cooperate, it is important to find how these cooperate to understand the structure of the code. In object-oriented coding, applying inheritance is a common practice, i.e. using and referring to parent and child classes [24]. However, with this combination of classes, the role a single class plays in its program unit becomes unclear. By extracting the code's structure, more clarity is achieved. Craft et al. brought up the following issue: *"When maintaining multiple configurations (or when the code from which one configuration was derived), how does the source code modeling utility know what has changed and what has not since the last time that the model of the subject source code was built?"* With this issue directly concerning the extraction of code's structure, it must be included. This issue could have two outcomes depending on the SAT: **1)** The SAT is not designed to maintain multiple configurations or generically extract the code's structure to identify instances of vulnerabilities. **2)** The SAT extracts the code's structure to identify instances of vulnerabilities generically.

- **Ability to diff multiple code structures:**

Concerning the extraction of code's structure, Craft et al. brought up the following: *"Should one of SAT's features be the ability to visually diff two models?"* With this, the SAT would either have or not have the ability to compare files to determine how or whether they differ. Not only would this assist with the analyses of different system configurations, program units, or source code files for the same project, but also with the comparison of different projects of different authors. This feature results in shifting the SAT towards the purpose of tackling code plagiarism. With tackling code plagiarism not being the main purpose of SATs, not all SATs will have spent resources on this functionality. However, since this feature brings additional value to the SAT, this will be included in this study alongside its two defined outcomes: **1)** The SAT is not able to visually differentiate between two models. **2)** The SAT can visually differentiate between the two models.

4.2.1 Identifying dependencies

For constructing a source code model, one should go beyond looking into the code's structure. Accordingly, a deep dive into the dependencies of the software project is supposed to shine a light on the work. With Westland extensively studying dependencies in project management life cycles [32], it is natural to involve his work and discuss his rationale on what constitutes dependencies in software before discussing what internal and external dependencies are and to what extent these can or should be identified by SATs. Westland said [32]: *"Dependencies are logical relationships between phrases, activities or tasks which influence the way that a project will be undertaken. Dependencies may be internal to the project (between project activities) or external to the project (between project activity and business activity)".*

The definition Westland has given to dependencies [32], is in line with that of Craft et al. who put emphasis on internal dependencies such as parent-child or sibling relationships within the code to *"identify the structural components of the code and the relationships that exist between these components"* [9], and external dependencies to *"... assist the analyst in identifying whether or not the element contains any interfaces that extend beyond the program unit that contains the element. For those external interfaces that do exist, SAT permits the user to document the nature of the interface in terms of various qualifiers (e.g., operating system calls or external messaging)"* [9].

- **Identifying internal dependencies:**

As discussed previously along with the work of Westland [32] and Craft et al. [9], the internal dependencies include parent-child and sibling relationships. After extracting the code's structure, the next step to going more in-depth for the software project analysis is identifying the internal relationships within the code. Either **1)** The SAT is not adequate to identify internal dependencies, or **2)** the SAT allows for the detailed analysis of the code's internal dependencies.

- **Identifying external dependencies:**

While working on his book on project management life cycles, Westland identified external dependencies as the following [32]: *"External dependencies are activities within the project which are likely to impact on or be impacted by an activity external to the project."* To create an understanding of how external dependencies appear in code, Code Snippet 1 is included. This abstraction of source code, as illustrated by Joukov et al. [17], shows two instances of external dependencies, namely for variables **f0** and **f1**. These are externally dependent since the values are subject to the value taken from the file linked via the code to this matching variable.

```
1 | FileInputStream f0, f1;
2 | f0 = new FileInputStream("/data/matrix");
3 | if (version == "old")
4 | name = "matrix.old";
5 | else
6 | name = "matrix";
7 | f1 = new FileInputStream("/data/" + name);
```
Code Snippet 1. Two examples of code-embedded file dependencies are visible on lines 2, 7 [17]

Craft et al. [9] defined four types of external dependencies and their according requirements. However, distinguishing between these types does not add value in this study, since it is more relevant whether or not an SAT supports this, instead of discussing to what extent. It is being analyzed whether **1)** external dependencies cannot be identified by the tool, **2)** external dependencies can be identified, but the corresponding files cannot be grouped automatically, or **3)** external dependencies can be identified and consequently the correct files are grouped. This grouping is done to represent a collection of cooperating source code files or program units.

4.3 Transforming code using the source code model

- **Modeling component vulnerabilities:**

Once a comprehensive model for the SAT is developed, the time comes to put the tool into practice. With the goal of SATs being to ensure good quality of code, it is important to define in what manners vulnerabilities appear in the software. This is realized most effectively by modeling component vulnerabilities. Not only do these vulnerabilities appear in source code files, but also system components, i.e. program units. For the modeling of vulnerabilities, Craft et al. [9] defined that *"the primary focus is on what could go wrong if variables within the program took on unexpected values or if events occur out of order."* Following this, two sorts of values are identified: variables representing an enumeration or a range of values. For both these sorts of values, it should be explicitly defined what is considered a normal or abnormal value or range. Different techniques are available for modeling component vulnerabilities, such as *"state transition diagrams, portions of state transition diagrams, and portions of data flow diagrams"* [9]. A software assurance tool can either **1)** not model component vulnerabilities or **2)** the software assurance tool can model component vulnerabilities.

- **Source lexing:**

As stated in the previous issue of concern, to create a practical tool that detects code vulnerabilities, it should be clear in what form vulnerabilities appear. For tools that analyze hundreds if not thousands of source code files, a generic solution must be applied to not overflow the database(s) of the SAT where known vulnerabilities are stored. The generic solution presented in this work, based on the work of Russel et al. [30] is to use source lexing, otherwise known as carrying out a lexical analysis [3]. Source lexing is the technique where one only captures the generic meaning of words or text, which in this context would be source code. Vulnerabilities can therefore be transformed into a generic piece of text using source lexing to then be saved in the SAT's database. All code from the source code files will then be transformed as well using the same source lexing technique. After the lexing, the source code text will be put up against the vulnerabilities known in the database to see if any matches are found. In case matches are found, these will be flagged by the SAT. Within this research, no distinction is made between different source Lexington utilities, however, different Lexington software could generate different (slightly) differing results - these differing results however will be the result of either false positives, or false negatives which are accounted for in the false detection rate of the corresponding SAT. To analyze source lexing for different SATs, the following two outcomes are identified: **1)** The SAT in question does not use the technique of source lexing as its primary technique for identifying vulnerabilities in source code. **2)** The SAT in question uses source lexing as the primary technique to find and flag vulnerabilities.

- **Data curation:**

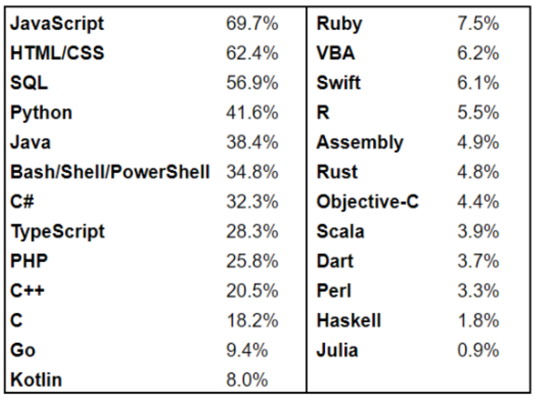
As identified by Russel et al. [30], it is important to remove duplicates from the same software components. They state that along with duplicates, it is important to design a strict duplication removal process where near-duplicates are excluded as well. This process can then be analyzed in detail using its results; for example by introducing a passing curation variable, reflecting the number or percentage of functions remaining from the total number of functions pulled from source code files. While this seems more like an optional attribute, it does contribute to the overall quality of an SAT by improving its code processing rather than working with duplicates that hold no additional value when being left in. Therefore the SAT either **1)** does not apply data curation, or **2)** does apply data curation. The value this attribute holds is as follows: By having the software assurance tool carry out more tasks to then visualize the results to the user, a better understanding of the analyzed software can be formed with less input and effort required from the user.

4.4 Identifying qualities

Along with the (primarily,) functional attributes of SATs formerly discussed, multiple qualities make or break whether an SAT is useful to put into practice or not. From existing literature [16] software qualities to consider are extracted, which are further explored and expanded on by discussing these aspects from the scope of an SAT researcher. These qualities are discussed further in this subchapter, mentioning the diverse range of computer languages, referred to as digital languages, along with the meaningfulness of the generated analysis reports and the bill of materials.

- **Support for digital languages:**

For building software, some knowledge is needed to manage its life cycle. With applications being built with high- or low-code tools, knowledge and understanding of a specific, or diverse set of digital languages to some extent is necessary. With different programming languages having different purposes and being used for a variety of applications, some programming languages are deemed generally more useful, in comparison to others. As a result, certain languages are more commonly used, while others are deemed more challenging to learn and apply. In 2020, the Stack Overflow Developer Survey [NP33] considered programming, scripting, and markup languages to create a ranking of the most popular languages. Using 47,184 responses from professional developers, it was found that for the eighth year in a row JavaScript is the most popular used language in the world (69.7%). Closely behind are HTML and CSS (62.4%), and further behind are SQL (56.9%), Python (41.6%), and Java (38.4%). The significant importance of these languages is visualized in Figure 2.



JavaScript	69.7%	**Ruby**	7.5%
HTML/CSS	62.4%	**VBA**	6.2%
SQL	56.9%	**Swift**	6.1%
Python	41.6%	**R**	5.5%
Java	38.4%	**Assembly**	4.9%
Bash/Shell/PowerShell	34.8%	**Rust**	4.8%
C#	32.3%	**Objective-C**	4.4%
TypeScript	28.3%	**Scala**	3.9%
PHP	25.8%	**Dart**	3.7%
C++	20.5%	**Perl**	3.3%
C	18.2%	**Haskell**	1.8%
Go	9.4%	**Julia**	0.9%
Kotlin	8.0%		

Figure 2. Top 25 commonly used digital languages following
a Stack Overflow Developer Survey [NP33]

This information becomes relevant when surveying different tools since this provides more insight into the practical use of SATs. With this study performing quantitative analysis, there is no perfect answer as to which supported combination of programming languages is best. However, with there being a ranking available with the most used languages of professional developers [NP33], certain levels can be defined. The ranking created by Stack Overflow recorded a list of 25 different languages, which can be considered when grading a tool's support for a diverse set of languages within the tool survey. Since not all SATs offer support for the same set of languages, a predefined set of outcomes is not feasible. Rather, a separate table will be used where a checkmark is given when a language is supported by the SAT in question. By setting multiple possibilities for the languages that are supported, multiple checkmarks can be attributed to the same tool. Note that the languages that fall outside the Stack Overflow ranking present a more niche market to support, possibly offering more business opportunities to expand. Also note that the languages must be fully supported within the tool, up to a level where it is feasible to work for developers. By using this method, a large number of checkmarks can be gained making the overview more obscure, however, it helps in easily distinguishing tools that support more languages than others.

- **Establishment of report:**

After the analysis has been carried out for the project, a report must be established with the key takeaways mentioned. This can be seen as a conclusion to the software project analysis, including summarizing aspects such as the number of methods or projects checked, and the number of matches with vulnerabilities present in the database. Without creating the report, there is no point in conducting the analysis. Nonetheless, this report includes various aspects, depending on the scope of the SAT developer. This aspect of the tool requires more inspection from a qualitative point of view rather than a quantitative one, but, also to stay in accordance, a simplistic distinction can be made between tools where **1)** analysis reports are not generated versus tools where **2)** analysis reports are generated.

4.5 Miscellaneous

When any tool, product, or even entity is similar to others in terms of what it provides for its user, one will either overlook the product or choose the least costly alternative. To stand out in its respective market, attention should be given to noteworthy features or other qualities that immerse the user into using the product rather than switching to its competition.

- **License tracking:**

The most eye-catching of these qualities, where developers go out of their way to include unexpected functionality for SATs, is tracking the licenses of software since for license tracking, separate applications are available: license analysis tools. Often, software consists of multiple program units, of which not all units are built by the same developer, or under the same company's supervision. In those cases, software license tracking is a useful addition. Regarding license tracking, it should be discussed whether **1)** the tool does not provide license tracking, or **2)** the tool does provide license tracking, and if so, to what extent.

- **Software Bill of Materials (SBOM):**

Another noteworthy aspect of the software is its bill of materials. The bill of materials is described as follows [NP39]: *"A software Bill of Materials (SBOM) is a list of all the open-source and third-party components present in a codebase (...) also lists the licenses that govern those components, the versions of the components used in the codebase, and their patch status (...) to quickly identify any associated security or license risks."* With this definition, it is possible to study whether the SATs can study the SBOM of software projects that are analyzed by the SAT. SATs are either **1)** not able to map the SBOM of software projects or **2)** can map the SBOM of software projects.

After finishing the analysis of each tool, an overview is given of the expenses that come along when choosing a certain tool. These are expenses such as the available editions or tiers to choose from and what features and qualities these include for the end-users.

4.6 Introducing the SAT Comparison Matrix

For making a comparison between SATs, there needs to be an immovable basis to work with. In other words, it should be clear what exactly is being compared before a matrix can be materialized to address the matter. For this, the bullet points addressed in this chapter are used. The blank SAT comparison matrix is presented in Table 1. With the SearchSECO tool taking center stage in this project, it is already noted as the first tool in the matrix. As will be done for each tool of this survey, a general description of the software assurance tool or platform will be given at first. Afterward, an elaborate description is provided of its features and qualities following the order of the SAT comparison matrix. With these descriptions, scores are assigned to the different aspects of the tools to then be used in the comparison matrix. Chapter 11 contains the SAT comparison matrix after surveying the SearchSECO tool along with its competition that is identified using the defined inclusion and exclusion criteria. While carrying out the tool survey, additional tables are added for an overview of the supported digital languages, and the metadata of the different tools including the year of establishment and payment plans.

Table 1. SAT Comparison Matrix Template
for software assurance tools and platforms

SAT Comparison Matrix		**Software Assurance Tools and Platforms**				
		SearchSECO	**Tool B**	**Tool C**	**Tool D**	**Tool E**
1. Identifying code	**1.1 Identifying system configurations in the project**					
	1.2 Identifying program units in the system configurations					
	1.3 Identifying source code files in the program units					
	1.4 Compatibility with software and platforms					
2. The source code model	**2.1 Extracting the code's structure**					
	2.2 Ability to diff multiple code structures					
	2.3 Identifying internal dependencies					
	2.4 Identifying external dependencies					
3. Transforming code using the source code model	**3.1 Modeling component vulnerabilities**					
	3.2 Source lexing					
	3.3 Data curation					
4. Identifying qualities	**4.1 Support for digital languages**	**See additional table**				
	4.2 Establishment of report					
5. Miscellaneous	**5.1 License tracking**					
	5.2 Software Bill of Materials (SBOM)					

Chapter 5. Tool Review: SearchSECO

Using the literature as a guide, it is viable to determine the extent to which the SAT comparison matrix, as detailed in Subchapter 4.6, corresponds with SearchSECO. Afterward, these steps will be reproduced for its competitors. To understand more about SearchSECO a general overview of the software project is given first, before discussing its more technical details.

The cyber security project SecureSECO (est. 2020) intends to make the worldwide software ecosystem a safer place, by maintaining a distributed ledger of facts about software that is used in the field. The data that is collected and maintained in the ledger can be used to prevent vulnerabilities in a software configuration from being abused by malicious attackers. The SecureSECO software is managed in a Distributed Autonomous Organization (DAO) and wishes to provide metadata on software trustworthiness as a common. The SecureSECO project is a collaboration between academic, corporate, and endorsement partners, consisting of five companies and five universities. By performing academic research, participating in hackathons, and providing academic research data for other research groups about software, over 10 researchers and tool developers collaboratively contribute to the vision of a safer and more secure worldwide software ecosystem.

Three products are under development in SecureSECO. 1) *TrustSECO*: a distributed ledger that is used to store trust data about software packages, to support the trust that customers of the package managers have. 2) *UseSECO*: an infrastructure that registers the use of open-source package usage in a distributed ledger, to increase trust, transparency, and security of the software ecosystem. 3) *SearchSECO*: a hash-based index for code fragments that enables searching source code at the method level in the worldwide software ecosystem. The latter product, SearchSECO, takes center stage in this bachelor project.

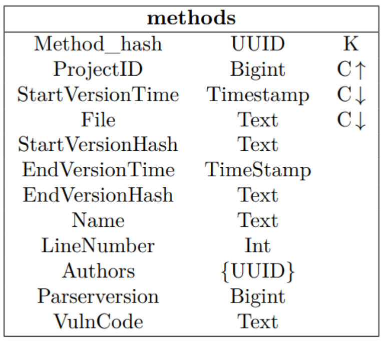
While the intent is to deliver an innovative new infrastructure, there are many related (commercial) services that perform similar tasks as the software in the SearchSECO product. This research investigates which products and services companies already provide and what services are related to SearchSECO. With this project best structured as a tool review, an inventory is made of existing tools and services in the field of code assurance and compared against the SearchSECO product. The absolute goal is to map whether SearchSECO solves a problem in the market that software-producing organizations are willing to pay for. Currently, it is insufficiently clear whether SearchSECO does exactly that. Therefore, the inventory of tools must be made, after which each tool is analyzed and compared against SearchSECO to seek noteworthy aspects of SearchSECO that help the tool stand out in its domain.

In principle, SearchSECO offers three main features:
• Check for clone detection concerning plagiarism;
• Check for clone detection concerning vulnerabilities;
• Check for license violations with the license analysis tool Tortellini [NP38].

For studying the features and qualities of SearchSECO, the official SearchSECO documentation and dashboard are consulted [NP22, NP23]. The review will be conducted as follows: first, information will be sought regarding the overarching sets of requirements as elaborated on in Chapter 4. Afterward, using this information, the different requirements of each set will be discussed concerning the gained knowledge regarding the tool and its capabilities. In case any requirement is insufficiently documented, the undertaken steps to study this instance are discussed to improve the repeatability of this study and discuss possible gaps within the SAT.

5.1 SearchSECO: Identifying code

Within the comparison matrix, it is defined that there are different meta-levels one could investigate, with either a broader or more narrow scope of analysis. There is the source code, which is the code itself as programmed by developers, there are program units which are multiple code files together cooperatively carrying out certain tasks interdependently, and there are the system configurations as a whole, containing all program units. The SearchSECO documentation [NP23] mentions that *"The Spider downloads repositories from a codebase (currently only GitLab and GitHub)"*. From this, one can extract that in the context of this documentation, repositories are a directory or storage space where projects can live. Once the repository is downloaded, the following information, as presented in Figure 3, is extracted and stored:



methods		
Method_hash	UUID	K
ProjectID	Bigint	C↑
StartVersionTime	Timestamp	C↓
File	Text	C↓
StartVersionHash	Text	
EndVersionTime	TimeStamp	
EndVersionHash	Text	
Name	Text	
LineNumber	Int	
Authors	{UUID}	
Parserversion	Bigint	
VulnCode	Text	

Figure 3. Project data regarding methods,
as collected by the SearchSECO tool [NP23]

Regarding the Spider, which is used for extracting project data, it is mentioned that *"Functions requiring a git repository to test aren't tested"* showing incompleteness. This Spider is managed by the Controller, which can carry out four relevant commands as extracted from the SearchSECO documentation [NP23]:

• Check: *"The check command will parse the most recent version of a given git repository, and finds all matches between the methods found in that repository and the methods stored in the database."* With this, different system configurations can be identified. Here, it is the Database API that identifies known methods with the same hash as the hashes sent to the Database API.

• Upload: The upload command processes all tags of a given repository, and uploads the methods found to the database. This includes metadata including licenses, authors, and the software project version. While the Upload command is carried out on projects used to fill in the database rather than projects to be checked on source code integrity for security assurance, this command ensures that SearchSECO is capable of collecting this metadata from its scanned projects.

• CheckUpload: *"The CheckUpload command both checks the most recent version for matches with the database and uploads every version of the given repository. This is mostly an ease-of-use command and is implemented by simply running both the check and upload commands. The output of the check command gives an overview of the encountered matches."* Showing similarity with the Check command, this function appears to identify different configurations within repositories.

However, the CheckUpload command then continues to check the most recent version rather than all identified configurations.

• Start: This command starts a worker node. This node requests a job from the Database API. There are 3 types of jobs the Database can assign: A Crawl job, a Spider/Parse job, and a NoJob response. These three jobs hold no further purpose for the scope of this study.

SearchSECO: 1.1 Identifying system configurations in the project: 2/2
As previously presented, the Check and CheckUpload commands work on identifying the most recent version of a given git repository. Through reasoning, this means that both commands can identify all versions of a given git repository, and from thereon identify the most recent version of a project. However, since it is documented that the latest version is checked, it leaves out the possibility of checking older versions. While this is intuitive since older versions are less likely to be in use, it constrains the tool and its possibilities.

SearchSECO: 1.2 Identifying program units in the system configurations: 1/2
SearchSECO does not assign resources to map program units within the system configurations. Rather, functions are taken directly from the files within the system configurations, and *"to prevent the matching of trivial clones, any function which contains less than 50 characters (after abstraction) or six lines are excluded."* In other words, the SearchSECO tool does not group related functions together and instead takes the large functions contained within the configuration to analyze. As a result, this requirement is insufficiently covered within SearchSECO.

SearchSECO: 1.3 Identifying source code files in the program units: 1/2
As mentioned regarding program units, also no distinction is made for source code files. Rather, there is only a distinction between different functions, since it is checked for each function whether it exceeds the set number of 50 characters or six lines. If so, then the function is included in the analysis. However, by focusing on a function or method level of vulnerability detection SearchSECO goes further than is specified within the comparison matrix.

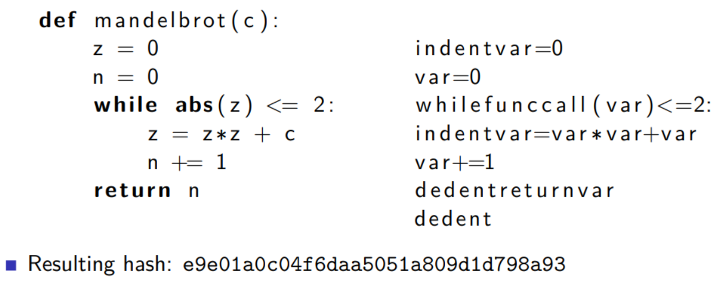
SearchSECO: 1.4 Compatibility with software and platforms: 2/2
Previously brought forward was the SearchSECO documentation [NP23] mentioning that *"The Spider downloads repositories from a codebase (currently only GitLab and GitHub)"*. With this compatibility, automation is possible regarding the extraction of repository data. The tool sufficiently provides details on the origin of the repositories, and constraints the compatibility to the two mentioned sources, namely GitLab and GitHub.

5.2 SearchSECO: The source code model

To find more on the extraction of the code's structure, the SearchSECO dashboard [NP22] and official presentation [NP14] are consulted. The presentation explains that the parser abstracts each method of either over 50 characters or over six lines, after which the abstracted representation is hashed. The parser currently offers support for C, C++, C#, Java, Python 3, and JavaScript. For files of C, C++, C#, and Java, the parser uses source markup language (srcML) to convert the code as presented in the source code files to an XML format. This decision is substantiated by its easy identification of methods, as well as variables and function calls. The parser currently calls srcML using a (hidden) command-line interface, rather than calling srcML directly, which could hold advantages. No problems have been encountered. To increase efficiency, multi-threading is used for the srcML to process input and output simultaneously. After going through the srcML, the output is sent to the XMLParser identifying the boundaries of files and methods and sending the valid methods to the AbstractSyntaxToHashable class. For both Python and JavaScript, custom parsers are used to abstract the source code. These custom parsers are built using ANTLR [NP2].

ANTLR is defined as follows [NP2]: *"ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees."* Primarily, the ANTLR parser uses lexing and parsing grammar to abstract the code.

The abstracted data, of all of the parser's supported languages, are turned into a string on which abstraction is applied, to then be hashed using the MD5 algorithm. Figure 4 visualizes three benchmarks: 1) Source code, 2) Parsed code, and 3) The resulting hash. The advantage of hashing is that the original text is presented with a shorter, fixed-length key, simplifying the process to locate and use.



```
def mandelbrot(c):
    z = 0                   indentvar=0
    n = 0                   var=0
    while abs(z) <= 2:      whilefunccall(var)<=2:
        z = z*z + c         indentvar=var*var+var
        n += 1              var+=1
    return n                dedentreturnvar
                            dedent
```

■ Resulting hash: `e9e01a0c04f6daa5051a809d1d798a93`

Figure 4. Example of a parsed method within SearchSECO [NP14]

SearchSECO: 2.1 Extracting the code's structure: 2/2
Following the SearchSECO documentation [NP23], *"a project can be updated based on a previous version of a project (which should already be in the database, of course). This is done by providing the previous version of the project together with a list of unchanged files. The methods in these unchanged files are then updated accordingly by updating their end version, and other, non-key, attributes."* meaning that updates of project repositories are taken into account when analyzing software projects.

SearchSECO: 2.2 Ability to diff multiple code structures: 2/2
The SearchSECO tool does not provide the ability to diff multiple code structures. Nonetheless, when changes are made within the repository of a tool that has been analyzed before, the newest version is put through the tool. Following this process, the old version is updated with the newer end-version, mapping the differences for the user. By way of explanation, it is not explicitly mentioned that the tool holds this ability, rather, following reasoning one can extract this information.

SearchSECO: 2.3 Identifying internal dependencies: 1/2
SearchSECO: 2.4 Identifying external dependencies: 1/3

Within the official documentation of SearchSECO, it is at no point mentioned that any form of dependencies is identified. Similarly, internal dependencies, alongside external dependencies, are not identified.

5.3 SearchSECO: Transforming code using the source code model

Using the aforementioned parser generator ANTLR [NP2], parse trees are both generated and walked using a LL parser [23]. Figure 5 presents a rather large parse tree that is formed using the parser, following a simple, two-line method in Python 3. By having a LL parser, it is ensured to have fewer conflicts arise since it is mostly based on hand inventing. This same parser is responsible for both lexing and taking care of grammar, using a grammar file. Next, the parse tree is walked through and tokenized, to transform the data of the tree into a string, before hashing.

```
def double(n):
    return n * 2
```

```
(file_input
 (stmt
  (compound_stmt
   (funcdef def
    (name double)
    (parameters (
     (typedargslist
      (tfpdef
       (name n))) )) :
    (funcbody
     (suite \n indent
      (stmt
       (simple_stmt
        (small_stmt
         (flow_stmt
          (return_stmt return
           (testlist
            (test
             (or_test
              (and_test
               (not_test
                (comparison
                 (expr
                  (xor_expr
                   (and_expr
                    (shift_expr
                     (arith_expr
                      (term
                       (factor
                        (power
                         (atom_expr
                          (atom
                           (name n))))) *
                       (factor
                        (power
                         (atom_expr
                          (atom 2))))))))))))))))))) \n)) dedent))))) <EOF>)
```

Figure 5. A small Python 3 method with its complete parse tree [NP23]

SearchSECO: 3.1 Modeling component vulnerabilities: 2/2
This requirement states that vulnerabilities should be able to be modeled using a predefined modeling language, such as *"state transition diagrams, portions of state transition diagrams, and portions of data flow diagrams"* [9], meaning that source code is transformed to parse trees.

SearchSECO: 3.2 Source lexing: 2/2
Covered in both Subchapter 5.2 and Subchapter 5.3 are the functionalities of the parser. Using the parser, as generated by and using ANTLR, lexing is applied to the source code before analyzing, following the SearchSECO documentation [NP23]: *"An example of a lexer rule is the following: WHILE : ' while ' ; This simply implies that whenever we encounter the string 'while' in the source, this should be tokenized as WHILE."*

SearchSECO: 3.3 Data curation: 2/2
While the Cassandra database [NP3] stores duplicates of the data on different nodes, this is more of a fundamental decision to prevent data loss at the time of node failures. More relevant is the method of functioning of the crawler. Once the crawler finishes crawling pages on GitHub, using the GitHub API, it returns the most recently identified project ID. This number prevents duplicate projects in the job queue by having worker nodes take care of this number and the queue. Regarding the JSON adapter, functions are in place to check if a variable exists, such as isNull(), isEmpty(), and contains(). While the crawler and JSON adapter concern the database entries, helper functions are set in place, such as repeatedGet() and exists(), which prevent code duplication and therefore code inflation. These functions make clear that the SearchSECO tool tackles code duplicates in an effective approach to reduce redundant code.

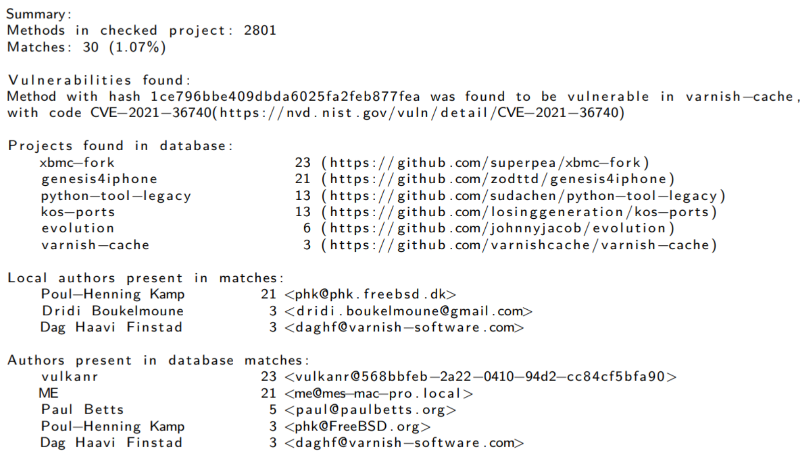
5.4 SearchSECO: Identifying qualities
Following the technical requirements (i.e. the features) come general qualities. As discussed in the SAT comparison matrix, these are the support for computer languages, including different programming, scripting, and markup languages a certain SAT can handle. Alongside this, an elaborate description is given of how final reports are generated by the SearchSECO tool after an analysis has been completed.

SearchSECO: 4.1 Support for digital languages

The following statement, as extracted from [NP23], summarizes what is needed to rate the requirement of to which extent the SAT supports digital languages: *"The Parser can currently parse C, C++, C#, and Java code using srcML, while parsing Python and JavaScript code using a custom parser built using ANTLR."* With JavaScript holding the number one position in the language ranking discussed in Figure 2, the SearchSECO tool already shows to provide a valuable quality. Alongside, there are Python (4th), Java (5th), C# (7th), C++ (10th), and C (11th). Whilst all of these supported languages earn the SAT checkmarks in this review, leading to **six** checkmarks, it is as significant to note that three of the five most common digital languages are supported. Consequently, support for a large proportion of the languages sought within the market for SAT tools is covered.

SearchSECO: 4.2 Establishment of report: 2/2

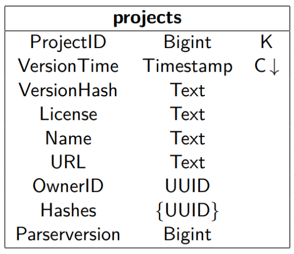
Following the presentation on SearchSECO [NP14], it becomes evident that an extensive analysis report is generated afterward. This includes repositories in which the job and report are stored, along with more specific details such as the number of methods in the project, the number of matches with the project, detected vulnerabilities, and author details. This is visualized in Figure 6.



```
Summary:
Methods in checked project: 2801
Matches: 30 (1.07%)

Vulnerabilities found:
Method with hash 1ce796bbe409dbda6025fa2feb877fea was found to be vulnerable in varnish-cache,
with code CVE-2021-36740(https://nvd.nist.gov/vuln/detail/CVE-2021-36740)

Projects found in database:
    xbmc-fork                    23 (https://github.com/superpea/xbmc-fork)
    genesis4iphone               21 (https://github.com/zodttd/genesis4iphone)
    python-tool-legacy           13 (https://github.com/sudachen/python-tool-legacy)
    kos-ports                    13 (https://github.com/losinggeneration/kos-ports)
    evolution                     6 (https://github.com/johnnyjacob/evolution)
    varnish-cache                 3 (https://github.com/varnishcache/varnish-cache)

Local authors present in matches:
    Poul-Henning Kamp        21 <phk@phk.freebsd.dk>
    Dridi Boukelmoune         3 <dridi.boukelmoune@gmail.com>
    Dag Haavi Finstad         3 <daghf@varnish-software.com>

Authors present in database matches:
    vulkanr                  23 <vulkanr@568bbfeb-2a22-0410-94d2-cc84cf5bfa90>
    ME                       21 <me@mes-mac-pro.local>
    Paul Betts                5 <paul@paulbetts.org>
    Poul-Henning Kamp         3 <phk@FreeBSD.org>
    Dag Haavi Finstad         3 <daghf@varnish-software.com>
```

Figure 6. Post-analysis report by SearchSECO [NP14]

However, by also identifying other attributes of the project, the report becomes more elaborate: Figure 7 shows what attributes of projects are stored, where the emphasis is on the IDs, namely the ProjectID, OwnerID, and the generated hashes which can be used for later data retrieval.



projects		
ProjectID	Bigint	K
VersionTime	Timestamp	C↓
VersionHash	Text	
License	Text	
Name	Text	
URL	Text	
OwnerID	UUID	
Hashes	{UUID}	
Parserversion	Bigint	

Figure 7. Stored attributes of projects when analyzed by SearchSECO [NP14]

While analysis reports being generated offer something extra to the tool, intuitively speaking, analysis reports are likely to be generated after conducting an analysis.

5.5 SearchSECO: Miscellaneous qualities and features

At last, regarding the review of the SearchSECO tool, miscellaneous requirements are discussed: License tracking, and the software bill of materials. In other words,

SearchSECO: 6.1 License tracking: 2/2

At times, cases come forward where code smells, such as license violations, are the center of attention [28] [S5]. This is considered another goal for SATs, however, not as often focused on as necessary, with research finding that this issue is still at large. Golubev et al. studied a dataset of 23,378 projects on potential code borrowing and license violations in Java projects on GitHub [13]. These projects had been selected since they had at least 50 stars and at least one line of Java code and it was later found that 9.4% of them potentially violate original licenses. With over 128 million public repositories on GitHub, this poses a threat to a large number of software project owners. With this threat being tested using those specific requirements for projects included in the study of Golubev et al., there is likely a significant number of projects that are excluded but put at risk. Considering this could lead to lots more violations of licenses for projects that have less than 50 stars or do not include Java code, attention should be paid to the problem of license violation [1]. While license tracking is more commonly performed by software license tracking tools, SATs such as SearchSECO take it upon themselves to fight code duplicates that are not authorized by license to use the code in that certain manner.

According to the SearchSECO documentation [NP23], the Crawler is instructed to retrieve metadata for the repository, like the license and default branch, in case a license is available. The license is included in the project data using text by extracting this from git, as shown in Figure 7. As previously mentioned in the list of three main features offered by SearchSECO (Chapter 5) license analysis tool Tortellini is used [NP38]. While SearchSECO conducts license tracking, with 2 mentions and 4 contributors as mentioned on the official website [NP38], Tortellini is not a commonly implemented license analysis tool.

SearchSECO: 6.2 Software Bill of Materials (SBOM): 1/2

The SearchSECO tool cannot extract the SBOM of the tools it analyzes.

Access to and usage of the SearchSECO SAT is offered fully for free.

Regarding the first subquestion defined in Subchapter 2.1, the following answer is formed regarding the goals of SearchSECO, after having analyzed its do-hows, per the SAT matrix:

The answer to Sub-question 1:
SearchSECO attempts to deliver an innovative infrastructure for SATs where the expected requirements are implemented using efficient algorithms, while also paying attention to unique, unexpected qualities such as license tracking.

Chapter 6. Competition Identification

Having identified SearchSECO and having it put up against the matrix, it is appropriate to continue with identifying its competitors. Discussed within Subchapter 2.5 are numerous criteria which promote the identification of competition. The criteria mentioned in that subchapter are further elaborated on, following the gathered information from Chapter 5. Joined, these criteria and information answer the second subquestion.

The answer to Sub-question 2:
(i1) To be included in the review of SATs, the tool must be similar to SearchSECO. This similarity is defined by both the requirements identified for SATs, and by the main goals to achieve for SATs, specifically SearchSECO. Based on the attributes of SearchSECO, which are described in Chapter 5, similar tools can be retrieved and compared. Therefore, SATs which search through source code at the method level in the worldwide software ecosystem in a self-managed manner are sought.
(i2) The selected tools for comparison should be easily and directly available.
(i3) The selected tools are fully available on the market with documentation.
These criteria **(i2)** and **(i3)** are in place to be able to completely assess the SAT within a reasonable amount of time, which is defined as approximately half of one week per tool.

(e1) Exclude tools not available in the market for software assurance in the Netherlands.
(e2) Exclude tools that cannot be tested without a sales representative. The tools must be available for usage or testing without contacting the product owners to plan demos.
(e3) Exclude SATs aged less than 3.5 years. At the moment of writing, this research does not consider SATs established or after the year 2019. This does not take into account newer releases and is confirmed by either its official first release date or by using the search-by-date filter provided by Google, showing that releases were carried out before the year 2019.
(e4) Exclude SATs that cannot be considered direct competition.

To identify the competition of SearchSECO, different tools will be listed that are perceived as software assurance tools, to check whether these meet the inclusion and exclusion criteria. First, a short, official introduction of the product is given, clarifying why the tool is considered an appropriate competition for SearchSECO within this study **(i1)**. Additionally, it is checked whether these tools are available for direct use, along with documentation explaining what the tool offers, to check whether the tool meets criterion **(i2)** and criterion **(i3)**. After, the tools mentioned in the list are checked on adherence to the exclusion criteria. In case a certain tool does not adhere to the exclusion criteria, this tool will be filtered out of the review, while elaborately mentioning why the tool did not comply with the set terms. For the exclusion criteria, it will be checked whether the tool is restricted to a set of countries, and if so, whether or not the tool is available in the Netherlands **(e1)**. Then, it is checked whether the tool is available to test, for free, without a sales representative overlooking every action taken while carrying out the study **(e2)**. Then, the founding year is taken into account, which is mentioned next to the product for each product along with its owner(s), to exclude the tools built within the last 3.5 years **(e3)**. At last, tools that pose minimal rivalry with SearchSECO are also excluded **(e4)**.

• Semgrep, 2015, product of r2c [NP25]:
"Semgrep is a fast, open-source, static analysis tool for finding bugs and enforcing code standards at editor, commit, and CI time." **(i1)**. With the documentation of Semgrep being provided on the official website, Semgrep can easily and directly be analyzed **(i2)**. With Semgrep being open-source software and also available for free on GitHub [NP20], the tool adheres to the third

inclusion criteria **(i3)**. Still, the tool does have multiple tiers for customers to choose from [NP25] The tool shows support for a wide and diverse set of languages to production-level support with few known bugs. Although its GitHub page poses the first release dating back to the year 2019, using the search-by-date filter provided by Google it becomes evident that Semgrep has been carrying out releases since 2015. The tool is therefore considered mature enough for this review. Through GitHub, the tool is also accessible for free in the Netherlands.

• VUDDY, 2015, a product of IoTCube.com [NP12]:
"IoTcube provides easy-to-use analysis for discovering security vulnerabilities in IoT devices." Furthermore, VUDDY, also known as Hmark, is available on GitHub, where the following is mentioned: *"VUDDY is an approach for scalable and accurate vulnerable code clone detection. This approach is specifically designed to accurately find vulnerabilities in massive code bases (e.g., Linux kernel, 25 MLoC)."* **(i1)** Along with the tool, its documentation [19] can be found via GitHub **(i2) (i3)**. With the tool being available on GitHub, it is accessible in the Netherlands without a sales representative being present. With the foundation of the tool dating back to 2015, the tool is considered sufficiently mature to be included in the survey.

• Black Duck, 2002, a product of Synopsys, Inc. since 2017 [NP28]:
"Black Duck software composition analysis solutions and open-source audits give you the insight you need to track the open-source in your code, mitigate security and license compliance risks, and automatically enforce open-source policies using your existing DevOps tools and processes" **(i1)**.

Statements such as the following suggest the tool is not easily available to study: *"Contact us for the most current list of supported languages and platforms."* Nonetheless, Synopsys provides its community with a platform [NP35] dedicated to answers to FAQs, knowledge articles, tutorials, and documentation, adequately presenting information for the survey **(i2) (i3)**. With no regional restrictions mentioned or posed when accessing its documentation, tutorials, and articles, it becomes clear the tool is available in the Netherlands. While the tool consists of many modules, an analysis can be conducted on the modules relevant to the scope of this study. Additionally, demos can only be requested through a sales representative by an official company, which would obstruct the inclusion of Black Duck in this research study since the tool does not adhere to the exclusion criterion **(e2)**, but with this disobedience being limited to demos, it remains feasible to include the tool in the survey. Furthermore, the tool is to be considered an excellent candidate for the tool survey, since the tool was founded in 2015, and can be considered direct competition, having no link with SearchSECO other than a competitive one.

• Sigrid, 2015, a product of The Software Improvement Group [NP26]:
"Expose hidden risks and opportunities in your source code to assure full control over your digital transformation" **(i1)**. Furthermore, the tool is available via GitHub [NP29] **(i2)**. Although a list containing frequently asked questions along with their respective answers is provided [NP29], documentation on realizing the features, qualities, and methods is lacking **(i3)**. While the tool does not comply with criterion **(i3)**, the tool is available in the Netherlands without a sales representative and dates back to the year 2015. Furthermore, The Software Improvement Group is a corporate partner of SecureSECO. With SecureSECO being the umbrella company of SearchSECO [NP16], this relation suggests that Sigrid is not in direct competition with SearchSECO following the aforementioned definition of competition **(e4)**.

• Snyk, 2015, product of Snyk [NP27]:
"Find and automatically fix vulnerabilities in your code, open-source dependencies, containers, and infrastructure as code — all powered by Snyk's industry-leading security intelligence" **(i1)**. Upon further inspection, Snyk offers four different products: Snyk open-source, Snyk Code, Snyk

Container, and Snyk Infrastructure as Code. By providing four different, specialized products, rather than a single, overarching product, studying the tool becomes more complicated. Alongside this, this leads to falling short on the criteria **(i2)** and **(i3)**, consequently suspending further analysis of the tool.

• ScanCode, 2015, a product of NexB [NP21]
"ScanCode is the most effective and efficient open-source tool for Software Composition Analysis (SCA), used and trusted by the Linux kernel maintainers as a code scanning engine" **(i1)**. However, the tool is also presented as follows: *"a tool to scan code and detect licenses, copyrights and more"* showing that ScanCode is a license tracking tool rather than a software assurance tool. Further analysis is therefore suspended.

• SonarQube, 2006, product of SonarSource SA [NP30]
Sonar, SonarSource, SonarLint, SonarQube, and SonarCloud are different products, but all are trademarks of SonarSource SA. SonarSource provides two similar products: SonarCloud and SonarQube. Regarding SonarCloud, the following is mentioned: *"SonarCloud is a cloud-based code analysis service designed to detect code quality issues in 25 different programming languages, continuously ensuring the maintainability, reliability, and security of your code"* Also, it is mentioned that SonarCloud is a service and cloud-based static analysis tool for CI/CD workflows. SonarQube, on the other hand, provides a self-managed static analysis tool for continuous codebase inspection: *"SonarQube empowers all developers to write cleaner and safer code."* **(i1)**. Of these two tools, SonarQube shows the most similarity with SearchSECO since SearchSECO and SonarQube both present a non-cloud-based model, therefore this investigation continues with SonarQube. With the SonarQube Community Edition being available for free, and SonarQube being entirely free for all open-source projects, the tool is available for testing. While all other SonarQube editions (*"Developer Edition"* and *"Enterprise Edition and Data Center Edition"*) are commercial and require a paid license, documentation is fully and directly available nonetheless **(i2)** **(i3)**. With the tool being available for installation without a sales representative in the Netherlands and dating back to 2006, the tool complies with the set criteria. By having no relation with SearchSECO, SonarQube poses direct competition and is to be included.

Having identified six tools using both the inclusion and exclusion criteria, four tools remain:
• Semgrep
• VUDDY
• Black Duck
• SonarQube

In the following chapters, the remaining tools will be analyzed separately using the SAT Comparison Matrix, as carried out for SearchSECO in Chapter 5. At last, all reviewed tools will be compared which results in a filled-in form of the matrix to then be concluded on.

Chapter 7. Tool Review: Semgrep

When discussing Semgrep, three core topics have to be covered: its ecosystem, findings, and rules. The Semgrep ecosystem consists of three parts: Semgrep, Semgrep CI, and Semgrep App. Semgrep runs offline on uncompiled code, with no code leaving the device. Semgrep CI offers continuous integration, and Semgrep App offers a dashboard to visualize findings. A finding is the result of analyses of Semgrep and Semgrep CI. Findings are generated when a Semgrep rule matches a piece of source code. After matching, a finding can make its way through the Semgrep ecosystem. Semgrep rules are specific patterns based on which findings Semgrep reports in code. These findings may help catch issues such as security, performance, and correctness. Rules are stored in the open-source Semgrep Registry that enables you to scan code without the need to write anything custom.

As defined in this tool survey, a job is an overarching name for the set of tasks to be carried out for a software project. Semgrep shares this definition and offers to add Semgrep to the project repository. With Semgrep, more specifically Semgrep CI, being added to the repository, rather than adding the repository to Semgrep, there is no possibility to distinguish different analyses using Semgrep.

Observe that since the continuous integration happens continuously while working on the software project, opening an existing job is not an option unless the report is exported and stored elsewhere. The dashboard does not mention the date at which a certain finding happened, nonetheless it shows what current fixes are needed, including categories to sort these findings. Several categories are defined: Code injection, Cookie flag, Cross-site request forgery (CSRF), Active debug code, Cryptography, Deserialization, Path traversal, Regex, Open redirect, and Command injection [NP24]. This dashboard is part of the Semgrep App, which can be connected to Semgrep CI. The findings that Semgrep visualizes to its users have four states: open, fixed, muted, and removed. While the option is not given to remove all findings at once, it is possible to ensure that a specific finding's rule is not enabled on the repository anymore. The advantage of Semgrep CI is that these findings pop up live while working on the code.

7.1 Semgrep: Identifying code

Semgrep only identifies what the user specifies for Semgrep to identify, using the (custom-made) rules. With Semgrep being added to a repository, it in itself means the tool does support identifying system configurations, but not program units or source code files. Semgrep presents itself as being at its best when used to continuously scan code. For this continuous scanning, the continuous integration version of Semgrep, namely Semgrep CI, is needed. Semgrep CI is compatible with some of the more popular software project platforms, such as GitHub, GitLab, Slack, and Jira. Members of the Semgrep community also managed to integrate Semgrep into the following, less widely known CI providers: Bitbucket, Pipelines, Bitrise, Buildbot, Buildkite, CircleCI, Codefresh, Jenkins, TeamCity CI, Travis CI [NP24].

7.2 Semgrep: The source code model

With Semgrep striving for simplicity by delivering a lightweight and fast static analysis, a few design trade-offs are made [NP8]:

• No path sensitivity: All potential execution paths are considered, although some may not be feasible.
• No pointer or shape analysis: Individual elements in arrays or other data structures are not tracked.

• There is no proper field sensitivity at present but may be developed in the future.
• No soundness guarantees: Semgrep ignores the effects of eval-like functions on the program state. It doesn't make worst-case sound assumptions, but rather "reasonable" ones.

Together, these trade-offs show that the tool does not extract the code's structure, to deliver the lightweight and fast static analysis promised to its users. Rather, Semgrep tracks the data for a few specifics, namely constant propagation and taint checking. With the findings of Semgrep defined in four states as mentioned, Semgrep does not offer to differ between multiple code structures. A finding is tied to a certain part of the code, rather than a part of the code along with its version or date. Regarding dependencies, Semgrep is currently testing out the *project-depends-on* key to specify third-party dependencies. Unfortunately for its users, it is mentioned there's no expectation of stability across releases yet. There are also no indications that Semgrep rules can manage internal dependencies.

7.3 Semgrep: Transforming code using the source code model

The vulnerabilities found in the code are modeled in the Semgrep App dashboard. The app supports scanning and visualizing the OWASP Top 10 vulnerabilities. Semgrep mentions the OWASP Top 10 to be an industry-recognized report of top web application security risks and provides a ruleset to scan for these risks. This ruleset can both be run locally (Semgrep) and continuously integrated (Semgrep CI). For transforming the code to a generic abstract syntax tree form, source lexing is used in Semgrep before parsing. For this, Semgrep uses the *Lexer_python.mll* submodule as a lexer, and the *Parser_python.mly* submodule as the parser. However, no form of data curation is carried out by Semgrep.

7.4 Semgrep: Identifying qualities

For a set of languages, Semgrep provides production-level support with few known bugs. Users are expected to give feedback as the developers are actively looking for bug reports. The support team of the tool responds generally within 24 hours regarding support for the production-level languages. Alongside these languages are their respective rankings as compared to the Stack Overflow Developer Survey (Figure 2): JavaScript (1st), Python (4th), Java (5th), C# (7th), TypeScript (8th), Go (12th), Ruby (14th), Scala (21st), JSON, JSX, and TSX [NP34]. With the latter three mentioned (JSON, JSX, and TSX) being file formats and language extensions rather than languages themselves, these are not taken into account within the review.

After analyzing the code, certain findings are noted based on the implemented rules rather than a report. Using those findings one can analyze the quality of the software project.

7.5 Semgrep: Miscellaneous qualities and features

As can be derived from the categories in which findings are sorted, Semgrep does not offer license tracking for code since it does not belong to any of the categories that findings can be placed in. Furthermore, the SBOM of software projects is not extracted either.

Semgrep has three tiers for its customers to choose from [NP25]:
• Community (free) which is for general-purpose security scanning.
• Team (US$40 monthly per developer) which is for the enforcement of company-specific coding standards, private rules, and the analysis of findings.
• Enterprise (custom pricing) which gains more individual attention by Semgrep as custom solutions, including deployment into virtual private clouds (VPCs) with dedicated technical support is offered.

Based on the study carried out on Semgrep, the following scores are assigned regarding the features and qualities mentioned in the SAT Comparison Matrix:

Semgrep: 1.1 Identifying system configurations in the project: 2/2
Semgrep: 1.2 Identifying program units in the system configurations: 1/2
Semgrep: 1.3 Identifying source code files in the program units: 1/2
Semgrep: 1.4 Compatibility with software and platforms: 2/2

Semgrep: 2.1 Extracting the code's structure: 1/2
Semgrep: 2.2 Ability to diff multiple code structures: 1/2
Semgrep: 2.3 Identifying internal dependencies: 1/2
Semgrep: 2.4 Identifying external dependencies: 1/3

Semgrep: 3.1 Modeling component vulnerabilities: 2/2
Semgrep: 3.2 Source lexing: 2/2
Semgrep: 3.3 Data curation: 1/2

Semgrep: 4.1 Support for digital languages
Semgrep: 4.2 Establishment of report: 1/2

Semgrep: 5.1 License tracking: 1/2
Semgrep: 5.2 Software Bill of Materials (SBOM): 1/2

Chapter 8. Tool Review: VUDDY

VUDDY, commonly referred to and known as hmark, is proposed since it provides a scalable approach for the detection of vulnerable code clones. hmark is referred to as the implementation of VUDDY. VUDDY can detect vulnerabilities in large programs efficiently and accurately. The scalability is made possible by using function-level analyses and length-filtering techniques.

8.1 VUDDY: Identifying source code

The following definitions are given to explain what VUDDY does and does not take into account when analyzing code (quote) [19]:
• Token: This is the minimum unit the compiler can understand. For example, in the statement int i = 0; five tokens exist: int, i, =, 0, and ;.
• Line: This represents a sequence of tokens delimited by a new-line character.
• Function: This is a collection of consecutive lines that perform a specific task. A standard C or C++ function consists of a header and body. A header includes a return type, function name, and parameters, and a body includes a sequence of lines that determine the behavior of the function.
• File: This contains a set of functions. A file may contain no functions. However, most source files usually contain multiple functions.
• Program: This is a collection of files.

While code cloning can occur within any of the listed units, VUDDY carries out analyses on the function level of code. Nonetheless, as Figure 8 visualizes, VUDDY can distinguish separate code files in a project.

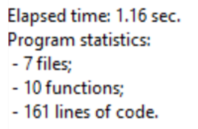

```
Elapsed time: 1.16 sec.
Program statistics:
 - 7 files;
 - 10 functions;
 - 161 lines of code.
```

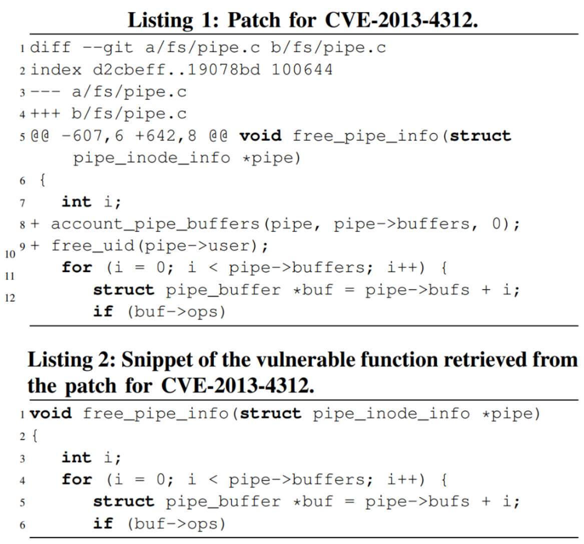
Figure 8. Statistical overview after scanning a software project

When one has an interest in using VUDDY, then they will have to download the software project to the local storage, to upload it from there into the tool [NP11]. As a result, the tool is not made compatible with other software or platforms.

8.2 VUDDY: The source code model

VUDDY carries out five steps to detect and analyze threats [19]: The first of these steps is function retrieval. VUDDY uses a robust parser to retrieve functions from a given program (or software project). Afterward, abstraction is applied to formal parameters, local variable names, data types, and function calls depending on the identified functions. With this, no dependencies are identified. Then, a so-called fingerprint is generated. This fingerprint is a string representing the combination of the normalized function body string's length and the hash value of this string. By continuously doing this, a “fingerprint dictionary” is created, containing the new and previously generated fingerprints. Then, VUDDY iterates over every key in a source dictionary, looking for the existence of the length of the preprocessed function. If unsuccessful, there is no clone detected, however, if successful, the tool continues with looking up the hash values included in the fingerprints created after abstraction in its database. By assigning hash values to individual files, vulnerability patches can be mapped and made comprehensible by adding the hash code of the old file next to the patched file in the patch report.

As an example, d2cbeff is registered as the hash value of the file in need of a patch, with 19078bd being the hash value for the same file, but patched. The report after the patch would then show a code line similar to "index d2cbeff..19078bd 100644", as shown in Figure 9. This shows that adjustments are taken into account for source code files of a software project. The aforementioned hash value can be used to retrieve the old, vulnerable function by querying "git show d2cbeff" to the cloned Git object. By being able to obtain the old file and see what is achieved with the patch, it is possible to differentiate between different code structures.



Listing 1: Patch for CVE-2013-4312.

```
diff --git a/fs/pipe.c b/fs/pipe.c
index d2cbeff..19078bd 100644
--- a/fs/pipe.c
+++ b/fs/pipe.c
@@ -607,6 +642,8 @@ void free_pipe_info(struct
    pipe_inode_info *pipe)
 {
   int i;
+  account_pipe_buffers(pipe, pipe->buffers, 0);
+  free_uid(pipe->user);
   for (i = 0; i < pipe->buffers; i++) {
      struct pipe_buffer *buf = pipe->bufs + i;
      if (buf->ops)
```

Listing 2: Snippet of the vulnerable function retrieved from the patch for CVE-2013-4312.

```
void free_pipe_info(struct pipe_inode_info *pipe)
{
   int i;
   for (i = 0; i < pipe->buffers; i++) {
      struct pipe_buffer *buf = pipe->bufs + i;
      if (buf->ops)
```

Figure 9. Example patch overview [19]

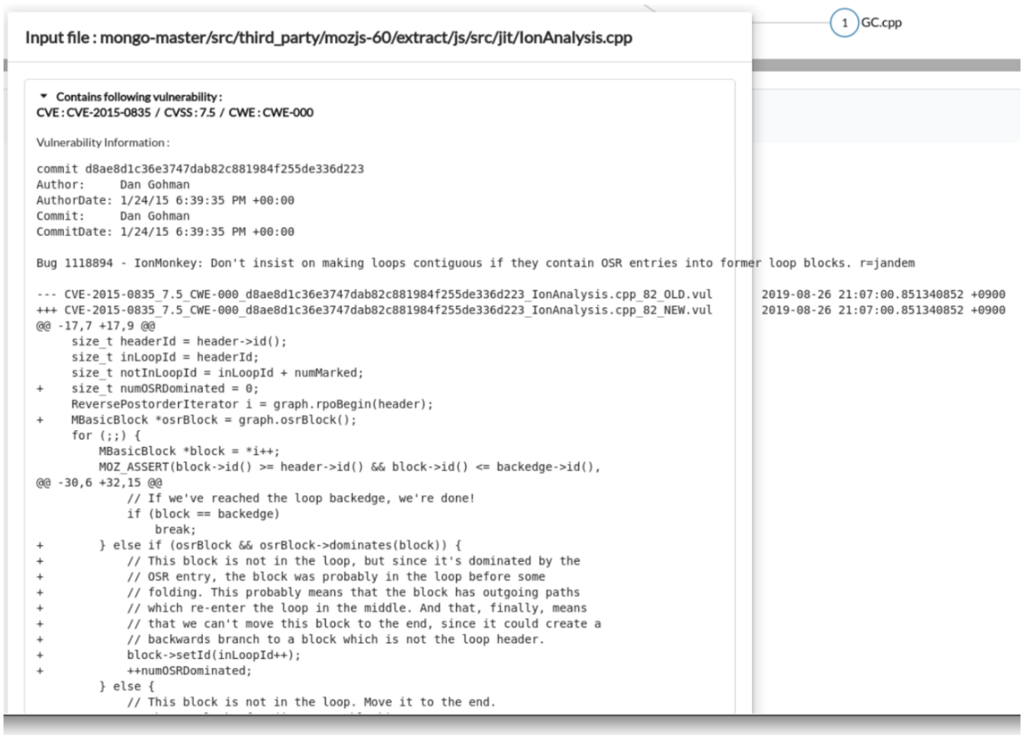
8.3 VUDDY: Transforming code using the source code model

After carrying out the steps of abstraction and the creation of the fingerprint, an overview is given of the original and preprocessed abstracted methods, therefore mapping the vulnerabilities. Using the fingerprint dictionary, VUDDY iterates over every key in a source dictionary. By iterating over every key, the tasks of the lexer are carried out. Furthermore, no data curation is applied to the content of the methods.

8.4 VUDDY: Identifying qualities

Following VUDDY contributor Seulbae Kim [NP32], currently, *"Vuddy only supports C and C++, but there's an extension of vuddy that supports Java"*. Although Java is not supported in the base program, it is included in this study since the tool offers support through the extension. With Java being ranked 5th, C++ ranked 10th, and C ranked 11th, VUDDY does not offer a wide range of language support in vulnerable code clone discovery. Next to the support of these languages, VUDDY can translate input source code to signatures [NP5]. This translation applies to programs of Javascript (1st), Python (4th), and Go (12th), however, these are not included as supported languages by the tool in this survey, since these signatures focus on an aspect other than identifying vulnerabilities.

With the visualizations of Figure 9 and Figure 10, it is evident that the VUDDY tool establishes reports after the job is carried out.



Input file : mongo-master/src/third_party/mozjs-60/extract/js/src/jit/IonAnalysis.cpp

▾ Contains following vulnerability :
CVE : CVE-2015-0835 / CVSS : 7.5 / CWE : CWE-000

Vulnerability Information :

```
commit d8ae8d1c36e3747dab82c881984f255de336d223
Author:     Dan Gohman
AuthorDate: 1/24/15 6:39:35 PM +00:00
Commit:     Dan Gohman
CommitDate: 1/24/15 6:39:35 PM +00:00

Bug 1118894 - IonMonkey: Don't insist on making loops contiguous if they contain OSR entries into former loop blocks. r=jandem

--- CVE-2015-0835_7.5_CWE-000_d8ae8d1c36e3747dab82c881984f255de336d223_IonAnalysis.cpp_82_OLD.vul      2019-08-26 21:07:00.851340852 +0900
+++ CVE-2015-0835_7.5_CWE-000_d8ae8d1c36e3747dab82c881984f255de336d223_IonAnalysis.cpp_82_NEW.vul      2019-08-26 21:07:00.851340852 +0900
@@ -17,7 +17,9 @@
     size_t headerId = header->id();
     size_t inLoopId = headerId;
     size_t notInLoopId = inLoopId + numMarked;
+    size_t numOSRDominated = 0;
     ReversePostorderIterator i = graph.rpoBegin(header);
+    MBasicBlock *osrBlock = graph.osrBlock();
     for (;;) {
         MBasicBlock *block = *i++;
         MOZ_ASSERT(block->id() >= header->id() && block->id() <= backedge->id(),
@@ -30,6 +32,15 @@
             // If we've reached the loop backedge, we're done!
             if (block == backedge)
                 break;
+        } else if (osrBlock && osrBlock->dominates(block)) {
+            // This block is not in the loop, but since it's dominated by the
+            // OSR entry, the block was probably in the loop before some
+            // folding. This probably means that the block has outgoing paths
+            // which re-enter the loop in the middle. And that, finally, means
+            // that we can't move this block to the end, since it could create a
+            // backwards branch to a block which is not the loop header.
+            block->setId(inLoopId++);
+            ++numOSRDominated;
         } else {
             // This block is not in the loop. Move it to the end.
```

Figure 10. Example vulnerability detection overview [NP10]

8.5 VUDDY: Miscellaneous qualities and features

VUDDY does not track the licenses and does not extract the SBOM of the analyzed software projects either.

The VUDDY tool is available free of charge to all of its users.

Based on the study carried out on VUDDY, the following scores are assigned regarding the features and qualities mentioned in the SAT Comparison Matrix:

VUDDY: 1.1 Identifying system configurations in the project: 1/2
VUDDY: 1.2 Identifying program units in the system configurations: 1/2
VUDDY: 1.3 Identifying source code files in the program units: 2/2
VUDDY: 1.4 Compatibility with software and platforms: 1/2

VUDDY: 2.1 Extracting the code's structure: 2/2
VUDDY: 2.2 Ability to diff multiple code structures: 2/2
VUDDY: 2.3 Identifying internal dependencies: 1/2
VUDDY: 2.4 Identifying external dependencies: 1/3

VUDDY: 3.1 Modeling component vulnerabilities: 2/2
VUDDY: 3.2 Source lexing: 2/2
VUDDY: 3.3 Data curation: 1/2

VUDDY: 4.1 Support for digital languages
VUDDY: 4.2 Establishment of report: 2/2

VUDDY: 5.1 License tracking: 1/2
VUDDY: 5.2 Software Bill of Materials (SBOM): 1/2

Chapter 9. Tool Review: Black Duck

9.1 Black Duck: Identifying source code

Within the Black Duck ecosystem multiple solutions, or analysis modules, are available. One of these is the component analysis with which different system configurations, or build systems as referred to by Synopsys, are identified [NP6]. The tool does not select a certain build by default but does offer working with multiple workspaces to not entangle different analyses. Black Duck Component Scanning is the module of Black Duck that offers an automated way to find the open-source software components that make up a software project. In the terms of this research project, the term “software components” is interpreted as a synonym for program units. While scanning open-source software components results solely in external components, the tool is capable of differing between components. Therefore, the support of the tool regarding this matter is confirmed. With snippets (further described in Subchapter 9.2) being taken from source code, it is evident that Black Duck offers source code file identification. Black Duck states to be compatible with GitHub, however, no other compatibilities are mentioned.

9.2 Black Duck: The source code model

To identify vulnerabilities in code, Black Duck analyzes snippets: the first 2MB of data of a file. Snippets can be manually adjusted between 1-16MB and typically need five to seven lines of code to find a match [NP6]. Using snippets, rather than complete files, productivity improves but the reliability of the tool is counteracted. Before scanning snippets, though, the matching service first attempts to find exact file matches. If matches are identified for the file, a heuristic is run to select the best match as the likely source along with a listing of the other matches. While Black Duck supported individual file matching, which is matching files to components based upon a checksum match to the one file, this was not always accurate and produced a significant amount of false positives, therefore this functionality was disabled. However, it is optional for users to re-enable this to have a more complete overview of the SBOM when analyzing a software project. Apart from the aforementioned, the documentation [NP6] does not provide further details on how the snippet or component identification is eventually carried out.

The Black Duck SAT offers a comparison of the signatures of software components within a software project. Rapid Scan, another Black Duck component, can also compare its results to an existing project version. Nonetheless, this comparison is only at the top level. While subprojects are compared, their components are not automatically compared. Still, it is identified for components whether these are added, modified, new, removed, or replaced, and optional for the user to compare scanned code with matched open-source code. A variety of matches can occur when the tool is scanning a software project. The signature, similar to the signature described in Subchapter 8.4, looks at the directory and file structures to match the "signatures" stored in its database, known as the KnowledgeBase [NP6]. Using the built-in package manager, external tools are set in place to examine system configurations to find declared dependencies and their matching open-source components in the KnowledgeBase. After the scan, "Transitive Dependencies" and "Direct Dependencies" are displayed [NP6]. Black Duck lists the number of matches for each type of dependency. The tool might display multiple matches, representing and depending on what different paths exist within the dependency tree.

9.3 Black Duck: Transforming code using the source code model

For each component, a vulnerability table is created. This table lists a set of attributes for each vulnerability, including an identifier, risk score, and current vulnerability status among others. To find and quickly fix vulnerabilities, the SAT deploys critical risk metrics, vulnerability-specific technical insight, exploit details, and impact analysis [NP36]. However, no lexer or a submodule that focuses on data curation is applied.

9.4 Black Duck: Identifying qualities

Black Duck supports a wide range of digital languages: JavaScript (1st), SQL (3rd), Python (4th), Java (5th), C# (7th), TypeScript (8th), PHP (9th), C++ (10th), C (11th), Go (12th), Kotlin (13th), R (17th), Objective-C (20th), Scala (21st), and PERL (23rd).

Alongside this, Black Duck creates a digital report for all relevant actions taken within the different modules of the tool. A report is generated for the notices (which include open source components, versions, licenses, and copyright statements), the project version, and vulnerabilities regarding remediation, status, and updates. At last, a report is established regarding the SBOM.

9.5 Black Duck: Miscellaneous qualities and features

The Black Duck Component Scanning technology determines the open-source components included in a software project. It can identify exact matches along with fuzzy matches of open-source components. With this technology, also relevant metadata (vulnerability, license, and project status) are identified and cataloged. Using the Component Scanning technology, the SBOM of a specific project can be automatically generated [NP6]. After, Black Duck's Rapid Scanning can be used to determine if the use of certain software components violates licenses, and provides the functionality to compare the SBOM of a current and previous version of a software project. This is however limited to SBOMs dating back a maximum of seven days.

Black Duck consists of multiple modules. The Software Composition Analysis tool is set in place to help manage security, quality, and license compliance risks that come from using open-source or third-party code. Black Duck SCA offers two paid plans [NP36]:
• Security Edition: $500/member, 20-50 members
• Professional Edition: Custom pricing; this edition offers more features regarding open source vulnerability detection and license compliance checking.

Based on the study carried out on Black Duck, the following scores are assigned regarding the features and qualities mentioned in the SAT Comparison Matrix:

Black Duck: 1.1 Identifying system configurations in the project: 2/2
Black Duck: 1.2 Identifying program units in the system configurations: 2/2
Black Duck: 1.3 Identifying source code files in the program units: 2/2
Black Duck: 1.4 Compatibility with software and platforms: 2/2

Black Duck: 2.1 Extracting the code's structure: 1/2
Black Duck: 2.2 Ability to diff multiple code structures: 2/2
Black Duck: 2.3 Identifying internal dependencies: 2/2
Black Duck: 2.4 Identifying external dependencies: 3/3

Black Duck: 3.1 Modeling component vulnerabilities: 2/2
Black Duck: 3.2 Source lexing: 1/2
Black Duck: 3.3 Data curation: 1/2

Black Duck: 4.1 Support for digital languages
Black Duck: 4.2 Establishment of report: 2/2

Black Duck: 5.1 License tracking: 2/2
Black Duck: 5.2 Software Bill of Materials (SBOM): 2/2

Chapter 10. Tool Review: SonarQube

10.1 SonarQube: Identifying source code

SonarQube offers continuous code inspection across the project branches and pulls requests by integrating the tool into an existing workflow. To make use of the tool and analyze code a SonarQube scanner must be installed and configured. This scanner has two methods in which it can be integrated into the workflow of a project [NP31]: it can either run on a build, also known as a branch (i.e., a certain system configuration) or be run by continuous integration. The (free and open-source) Community Edition of SonarQube offers the analysis of a single branch, while the paid editions offer the analysis of multiple branches. When analyzing a branch, by default, only the recognized and supported source code files are loaded into the job. Which files are (not) included depends on the selected SonarQube edition. Consequently, more files will be recognized during the analyses if carried out by the more cost-expensive editions. SonarQube does not offer the identification of different program units, however.

The Community Edition of SonarQube offers no compatibility with external platforms and tools. Unlike the Community Edition, with the Developer Edition, the user is granted pull request decoration for GitHub, GitLab, Bitbucket, and Azure DevOps. Therefore, it is noted that the tool is compatible with other software and platforms.

10.2 SonarQube: The source code model

While SonarQube carries out analyses, it does not test the coverage of the analysis to tell what percentage of source code is covered by test cases. And while the SAT directly supports coverage for data formats of the supported languages, it also supports importing a generic format to rewrite code in a custom manner [NP31]. By integrating third-party libraries, the SonarQube tool offers the identification of transitive dependencies [NP19], which by definition hold both internal and external dependencies. With the user being enabled to carry out multiple analyses, these results can be compared to that of other analyses. However, having the analyses of different projects in one place is a feature of the Enterprise Edition.

10.3 SonarQube: Transforming code using the source code model

Within SonarQube, there is a distinction between security hotspots and vulnerabilities. Security hotspots are seen as non-critical threats which do not affect the security level of a software project in an analysis. Still, these hotspots need to be reviewed by developers to ensure the safety of the project. Furthermore, some vulnerabilities need to be fixed immediately since these impact the security level. Rather than modeling vulnerabilities, a report is generated that shows how the software project scores regarding the number of vulnerabilities detected and the security score given to the code. SonarQube does not apply any form of source lexing or data curation.

10.4 SonarQube: Identifying qualities

SonarQube offers quite a wide range of support for digital languages: JavaScript (1st), HTML/CSS (2nd), Python (4th), Java (5th), C# (7th), TypeScript (8th), PHP (9th), Go (12th), Kotlin (13th), Ruby (14th), and Scala (21st). Furthermore, SonarQube supports Flex, VB.NET, and XML, however, these are not included in the Stack Overflow survey ranking, and therefore not further mentioned. With the Enterprise Edition, Apex, COBOL, PL/I, RPG, and VB6 are also supported.

Reports in PDF format are available for the Enterprise Edition and Data Center Edition [NP17]. These reports are stated to give a *"periodic, high-level overview of the overall code quality and security of your projects, applications, or portfolios"* [NP17]. This is either sent on a daily, weekly, or monthly basis, with the latter being the default. One of the reports that SonarQube generates is the security reports that include both the security hotspots and vulnerabilities. This report shows the result of the analysis where the source code is compared to the following popular and open-source vulnerabilities. OWASP Top 10 2021, OWASP Top 10 2017, CWE Top 25 2021, CWE Top 25 2020, and CWE Top 25 2019.

10.5 SonarQube: Miscellaneous qualities and features

The tool does not provide license tracking or extraction of the SBOM.

With SonarSource developing SonarQube, the pricing of the tool is found via SonarSource. The following is mentioned regarding the different editions of the tool (quote) [NP18]: *" Commercial Editions (Developer, Enterprise, and Data Center) are priced per instance per year and based on your lines of code (LOC). An instance is an installation of SonarQube. You pay per instance for a maximum number of LOC to be analyzed. Developer Edition pricing starts at $150/yr for a maximum of 100,000 LOC and can extend to $65K/yr for a maximum of 20M LOC. Enterprise Edition pricing starts at $20K/yr for a maximum of 1M LOC and can extend to $240K/yr for a maximum of 100M LOC."* The Data Center Edition starts at $130K/yr, being the more expensive solution, however, there is also a free open-source Community Edition.

Based on the study carried out on SonarQube, the following scores are assigned regarding the features and qualities mentioned in the SAT Comparison Matrix:

SonarQube: 1.1 Identifying system configurations in the project: 2/2
SonarQube: 1.2 Identifying program units in the system configurations: 1/2
SonarQube: 1.3 Identifying source code files in the program units: 2/2
SonarQube: 1.4 Compatibility with software and platforms: 2/2

SonarQube: 2.1 Extracting the code's structure: 2/2
SonarQube: 2.2 Ability to diff multiple code structures: 2/2
SonarQube: 2.3 Identifying internal dependencies: 2/2
SonarQube: 2.4 Identifying external dependencies: 1/3

SonarQube: 3.1 Modeling component vulnerabilities: 1/2
SonarQube: 3.2 Source lexing: 1/2
SonarQube: 3.3 Data curation: 1/2

SonarQube: 4.1 Support for digital languages
SonarQube: 4.2 Establishment of report: 2/2

SonarQube: 5.1 License tracking: 1/2
SonarQube: 5.2 Software Bill of Materials (SBOM): 1/2

Chapter 11. Tool Comparison: Overview

With the individual tool reviews being completed at this stage of the research project, an overview is given of the main findings in combination with an extensive review where the different SATs are being compared. At first, the completed SAT Comparison Matrix (Table 2) of this research will be discussed by going over the sets of requirements laid out in the matrix. Table 2 is filled in using the collected data of the individual tool analyses. Then, the support for digital languages (Table 3) is discussed along with its inferences. To complete the comparison of tools, the different pricing plans and editions for the SATs included in this study are presented.

Table 2. SAT Comparison Matrix for SearchSECO, Semgrep, VUDDY, Black Duck, and SonarQube

SAT Comparison Matrix		Software Assurance Tools and Platforms				
		SearchSECO	**Semgrep**	**VUDDY**	**Black Duck**	**SonarQube**
1. Identifying code	**1.1 Identifying system configurations in the project**	2/2	2/2	1/2	2/2	2/2
	1.2 Identifying program units in the system configurations	1/2	1/2	1/2	2/2	1/2
	1.3 Identifying source code files in the program units	1/2	1/2	2/2	2/2	2/2
	1.4 Compatibility with software and platforms	2/2	2/2	1/2	2/2	2/2
2. The source code model	**2.1 Extracting the code's structure**	2/2	1/2	2/2	1/2	2/2
	2.2 Ability to diff multiple code structures	2/2	1/2	2/2	2/2	2/2
	2.3 Identifying internal dependencies	1/2	1/2	1/2	2/2	2/2
	2.4 Identifying external dependencies	1/3	1/3	1/3	3/3	3/3
3. Transforming code using the source code model	**3.1 Modeling component vulnerabilities**	2/2	2/2	2/2	2/2	1/2
	3.2 Source lexing	2/2	2/2	2/2	1/2	1/2
	3.3 Data curation	2/2	1/2	1/2	1/2	1/2
4. Identifying qualities	**4.1 Support for digital languages**	See additional table				
	4.2 Establishment of report	2/2	1/2	2/2	2/2	2/2
5. Miscellaneous	**5.1 License tracking**	2/2	1/2	1/2	2/2	1/2
	5.2 Software Bill of Materials (SBOM)	1/2	1/2	1/2	2/2	1/2

First, the scope of analysis when it comes to identifying what the tool is working with differs from the reviewed tools. A tool such as VUDDY does not offer to identify a system configuration. Rather, it identifies that files or a group of files are uploaded instead of seeing these files as a unit, or as a configuration. This is in contrast to the other tools, where the tool is either integrated into the project for the tool to work in a CI manner, or the project repository is linked to the tool. Program units form the next metalevel of analysis after system configurations. Following the data presented in Table 2, it becomes evident that no tool other than Black Duck offers to analyze the program units. In this aspect, Black Duck shows to go further than the other SATs studied with its Component Scanning module. Regarding the source code file identification, SearchSECO and Semgrep show a lack of functionality. Both tools, however, focus on a function (or method) level of analysis. The tools, therefore, have a different focus, invalidating the perspective of the matrix on this matter. This is further discussed in Chapter 13. Furthermore, VUDDY is the only tool reviewed that does not show compatibility with third-party software or platforms such as GitHub and GitLab. So far, Black Duck is the stand-out tool, as expected when comparing tool costs, with VUDDY being the least developed of the five tools.

To create a model of the source code, SearchSECO, VUDDY, and SonarQube offer reforming source code to a generic structure. Using this generic structure, the retrieval of vulnerabilities is more simplified in comparison to Semgrep and Black Duck which do not apply an extraction of the code structure. While Black Duck does not create a generic structure of the code, it offers a visual comparison between structures. In this aspect, Semgrep is the only tool to not offer the functionality to compare different code structures of a software project. Regarding the dependencies, Black Duck and SonarQube are the only tools capable of identifying both internal and external dependencies. The other tools show a lack of functionality regarding identifying internal and external relations in code. The lack of identifying dependencies can presumably be traced back to the lack of identifying program units. Within program units, it is important to include dependencies to have certain methods and functions work as a collective.

Regarding modeling the component vulnerabilities, all tools except SonarQube have some method to identify and then visualize the vulnerabilities. SonarQube, however, does not present the vulnerabilities. Rather, it generates a report that includes the number of open vulnerabilities along with the security rating it gets assigned after analysis. Both Black Duck and SonarQube do not rework the code of a software project like the other tools do use source lexing. Not using lexers is not necessarily a flaw in the tools, but rather a different method of achieving the same goal, which is detecting vulnerabilities. However, this could be presumed to be a weaker form of detecting these threats in code since generic forms of code might result in fewer false-negative responses. That is fewer responses where there is no vulnerability detected because the detected vulnerability contains other lexical elements than the source code. Nonetheless, this is not factual and would require further studying, as elaborated on in Chapter 13. Therefore, creating generic forms of code results in a higher accuracy of vulnerability detection. Concerning data curation, SearchSECO is the only tool that explicitly mentions offering functionality to prevent overload of the system by using key identifiers to prevent the same projects from being analyzed more than once.

With the reports being generated after analyses, various types of reports are constructed with different tools. A tool such as Black Duck generates reports after every relevant action is taken for the user. Meanwhile, a tool such as SearchSECO only reports after the analysis is finalized, which includes the main information necessary to improve the security level of a project. On the other hand, the Semgrep tool does not create a final report but does inform its users of findings continuously, achieving the same goal of tagging vulnerabilities.

From the list of tools included in this survey, only SearchSECO and Black Duck offer the feature of tracking licenses. This feature is not a standard feature expected within SATs to perform tests to check compliance with security standards. Therefore, the tools offer noteworthy functionality to consider when in doubt about choosing an SAT. At last, The Black Duck tool is the only tool reviewed that extracts the SBOM of software projects analyzed.

Not only does Black Duck score high in most categories, but it also achieved a score higher than all other tools or a high score equal to at most one other tool in five categories. This makes Black Duck the most advanced tool included in the tool survey. On one of those categories, license tracking, it is SearchSECO that equals the score of Black Duck. This provides opportunities for SearchSECO by offering similar services when compared to those of the high-end, and high-cost tool that is Black Duck.

With only a single tool other than SearchSECO offering license tracking, it is considered a unique feature. Another unique feature is the curation of data. No other tool offers this feature. While the curation of data happens for jobs that are used to fill the database, the tool is capable of preventing the same analysis from taking place multiple times. This opens up opportunities for the tool to surpass competitors. To reap the benefits of this opportunity the unique identifier of software projects can be used, so using the ProjectID attribute (Figure 3) which is extracted from projects by the tool. Rather than reanalyzing a software project that has been analyzed before, this ID can be used to retrieve previous analysis reports to check for matches in a former analysis, based on the values of the ProjectID, StartVersionTime, and EndVersionTime attributes. By testing for a match between these attributes, it is made sure that the system configurations are the same. By doing this, the tool can be prevented from carrying out analyses more than once on the same software project.

Along with the different features and qualities, the supported languages for each SAT are visualized. However, this includes only the languages that are part of the Stack Overflow top 25 survey (Figure 2). Certain tools (i.e. Black Duck and SonarQube) offer a wider range of support for digital languages and environments, but these are not included in Table 3. Nonetheless, while leaving those cases out of the review, Black Duck and SonarQube prove to offer a greater variety of support in comparison to the other languages. The VUDDY tool scores the worst on this aspect, by being the only tool to not support at least two languages from the top nine digital languages. Despite the small variance, i.e. the SearchSECO tool supports the second least number of languages, four of the top seven digital languages are supported to a method level of analysis. Depending on the user's needs, SearchSECO could still offer opportunities. Note that non-supported languages of the top 25 ranking as found by the Stack Overflow survey are excluded from Table 3, along with supported languages that rank outside of the top 25.

Table 3. Support for top 25 digital languages for SearchSECO, Semgrep, VUDDY, Black Duck, and SonarQube. Non-supported languages are removed.

SAT Comparison Matrix		**Software Assurance Tools and Platforms**				
Support for top 25 digital languages		**SearchSECO**	**Semgrep**	**VUDDY**	**Black Duck**	**SonarQube**
JavaScript	#1	Y	Y	-	Y	Y
HTML/CSS	#2	-	-	-	-	Y
SQL	#3	-	-	-	Y	-
Python	#4	Y	Y	-	Y	Y
Java	#5	Y	Y	Y	Y	Y
C#	#7	Y	Y	-	Y	Y
TypeScript (TSX)	#8	-	Y	-	Y	Y
PHP	#9	-	-	-	Y	Y
C++	#10	Y	-	Y	Y	-
C	#11	Y	-	Y	Y	-
Go	#12	-	Y	-	Y	Y
Kotlin	#13	-	-	-	Y	Y
Ruby	#14	-	Y	-	-	Y
R	#17	-	-	-	Y	-
Objective-C	#20	-	-	-	Y	-
Scala	#21	-	Y	-	Y	Y
Perl	#23	-	-	-	Y	-

At last, there is Table 4 which presents the different editions or plans that the tools offer. The tools SearchSECO and VUDDY both present the tool to their users without a pricing plan. By offering the tools for no cost, a significant advantage is created over a tool such as Black Duck, which in its least cost-expensive form costs around 500 USD per user, with a minimum of 20 and a maximum of 50 users. While Semgrep and SonarQube offer a free edition to their respective communities, both free editions are limited in functionality when compared to the other plans. With the Semgrep tool offering a paid edition for 40 USD per month per developer, this edition is more affordable than the SonarQube plans which depend on the size of software projects rather than the number of developers working on the project.

Table 4. Pricing for SearchSECO, Semgrep, VUDDY, Black Duck, and SonarQube.

SAT Comparison Matrix	**Software Assurance Tools and Platforms**				
Metadata	**SearchSECO**	**Semgrep**	**VUDDY**	**Black Duck**	**SonarQube**
Year of establishment (est.)	**2020**	**2015**	**2015**	**2002**	**2006**
Pricing plans or editions Note: Prices are in USD.	**No pricing plan**	**Community:** Free **Team:** $40/ month/ developer **Enterprise:** Custom pricing; individual attention	**No pricing plan**	**Security:** $500/ member, 20-50 members **Professional:** Custom pricing	**Community:** No pricing plan **Developer:** $150/year - $65K/year **Enterprise:** $20K/year - $240K/year **Data Center:** $130K/year

With the individual tools reviewed in an individual setting along with a comparison of the data coming forward from these individual analyses, an answer can be formulated to Sub-question 3. This sub-question aimed at mapping aspects that can be considered unique compared to tools already on the market for some time.

The answer to Sub-question 3:
The SearchSECO tool offers the license tracking quality which, of the identified competition, only the Black Duck tool can offer. Despite SearchSECO already holding a competitive advantage by offering the license tracking feature for no additional costs, the Tortellini tool is unproven in its domain. Further research on the Tortellini tool and its license tracking competitors could strengthen the position of SearchSECO's position on license tracking. Along with the license tracking, the tool holds the opportunity to gain a further advantage over competitive tools. This advantage can be realized by applying the data curation techniques implemented for database jobs on regular software project analysis jobs.

Chapter 12. Conclusion

To conclude this research project, first, a summary is given of the undertaken steps and the main findings that have come forward during the study. Then, the main research question is answered. At last, the discussion and limitations can be found in Chapter 13.

Initially, literature research was conducted to give meaning to what the goals and challenges are for software assurance tools. Its main focus is assuring that code is of good quality, i.e. knowing what the code is for, ensuring there are no duplicates, and minimal to no occurrences of dark code. Software assurance tools are needed to uncover weaknesses to assess and eventually increase the security level of a project. In other words, these tools are used to help make the software more secure by scanning for vulnerabilities that developers have missed when building a software project. Then, fifteen aspects of software assurance tools were mapped following different research studies. These studies were found by applying a systematic snowballing procedure. Using these aspects a software assurance tool comparison matrix was established. With the SearchSECO tool taking center stage in this project, the tool was surveyed using the fifteen aspects identified. Using the findings from the literature and the conducted survey, inclusion and exclusion criteria were determined to identify the competition of SearchSECO. For the four identified competitive tools, a similar study was carried out. The collected data from these surveys ensured that the matrix was completed.

By studying the results, i.e. studying the completed comparison matrix and the additional results, the main research question can be answered. The main research question was formulated as follows: *"How can software assurance tools be compared?"* The corresponding answer is expressed as follows:

The answer to the Main Research Question:
Software assurance tools can be compared by studying the level(s) of analysis a tool works with. The level of analysis affects the techniques that are implemented within the tool and focus on different kinds of vulnerabilities. Focusing on a method level of analysis detects other threats versus a tool analyzing program units where dependencies, and therefore other kinds of vulnerabilities, are identified. Alongside this, it is important to check whether the tools create a generic structure of source code in their pursuit of finding vulnerabilities. Creating this generic structure depends on using techniques such as source lexing. Furthermore, a significant quality to keep track of during a comparison of software assurance tools is checking which digital languages are supported by the tool, with certain languages holding more value than others. Comparing these results with the available price plans of the respective tools would be the final obstacle before making a selection.

Chapter 13. Discussion and Limitations

Within this chapter, a discussion of the research is given along with certain limitations of the work. Discussing the limitations is not only mapping how these affect the current work but also how future work in this field of study can be refined to give more accurate and usable results.

First, the extent to which the snowballing procedure is applied in Chapter 3 is questionable. The ratio of literature references versus references that are the result of the systematic snowballing procedure is unbalanced. Possibly, further carrying out the systematic snowballing procedure and therefore expanding the literature research, other features and qualities of SATs could have been identified. Chapter 4 discussed the attributes of SATs, and levels were set to map whether or not certain attributes can be found in the different tools. However, this form of examination accounts for whether or not these attributes are included, rather than also discussing to what extent these are part of the focus, which is more subjective and opinionated with a significant part of this study being qualitative research. Alongside this, an expansion of Chapter 6 could lead to the identification of more SATs that are deemed eligible for this survey. A significant number of the tools identified were identified beforehand, rather than carrying out an own exploration during this study.

Additionally, the studies of the individual tools are not as rigorous as some might expect for such a tool evaluation. By studying documentation, presentations, dashboards, and platforms rather than the tools themselves, the tools are studied from different perspectives. However, developers and potential SAT users that require assistance with putting the tools into practice would face the tools from a similar perspective.

As mapped within the VUDDY documentation (see Subchapter 8.1), SATs can focus on different levels of analysis. These levels of analysis were defined as follows: token, line, function, file, and program. Tools such as SearchSECO and VUDDY have gone further than is specified within the comparison matrix by focusing on a function level of vulnerability detection. As a result, however, these tools score visibly lower than the other tools included in the tool survey. To be inclusive of these alternatives, requirements 1.1, 1.2, and 1.3 of the comparison matrix could be reformed to “Depth of analysis”. Here, for each tool, the different levels of analysis can be stated.

A question that arises when studying vulnerability detection is how creating a generic code structure to detect vulnerabilities differs from not applying a generic structure to source code. As mentioned in Chapter 11, not using lexers to create a generic structure is not necessarily a flaw in the tools, but rather a different method of achieving the same goal. Whether and to what extent this influences the rates at which vulnerabilities are detected is unclear. Further research could be conducted regarding this topic to study the same code on two groups of tools: one group where lexers and generic structures for code are applied, and another group where this is not the case.

A senior expert in this field of software security would have different views or methodologies in comparison to a bachelor’s student. Even starters in this section of IT with (little) practical experience, but experience in using the tool(s), would presumably have other views compared to a researcher newly exploring this field in a mostly theoretical manner. Therefore, time and resources could play a factor in future research. In case more time and resources are allocated to studying software assurance tools, then all individual tools included in a survey can be tested. Furthermore, a substantial part of the comparison matrix can be derived back to the work and chain of thoughts of Craft et al. Nonetheless, other researchers may think differently about their work, which this work builds on and perhaps include or exclude certain features and qualities when studying SATs.

Bibliography

Within the bibliography, there is a distinction between the three groups of references. Each group is distinguished by its unique identifiers, with the count for each group starting from 1. For background literature references, only the reference number is noted. For literature references that are the result of the systematic snowballing procedure, the letter "S" is added before the reference number. At last, for non-peer-reviewed literature and sources, the letters "NP" are added before the reference number.

Background Literature [x]

1. Alhazmi, O. H., Malaiya, Y. K., Ray, I. (2007). Measuring, analyzing, and predicting security vulnerabilities in software systems. Computers &#38; Security, 26(3), 219–228. https://doi.org/10.1016/j.cose.2006.10.002
2. Alvarez. (2002, January 7). Mapping the Information Society literature. Retrieved March 12, 2022, from https://firstmonday.org/ojs/index.php/fm/article/download/922/844?inline=1
3. Anatomy of a Compiler and The Tokenizer. (n.d.). Retrieved May 3, 2022, from http://www.cs.man.ac.uk/~pjj/farrell/comp3.html
4. Bari, M. A., & Ahamad, D. S. (2011). Code Cloning: The Analysis, Detection, and Removal. International Journal of Computer Applications, 20(7), 34-38.
5. Bendinskas, V., Mikaliūnas, G., Mitašiūnas, A., & Ragaišis, S. (2005). Towards mature software process. Information technology and control, 34(2).
6. Black, P. E., Fong, E., Okun, V., & Gaucher, R. (2008). Software assurance tools: Web application security scanner functional specification version 1.0. Special Publication, 500-269.
7. Bourque, Pierre; Fairley, Richard E., eds. (2014). Guide to the Software Engineering Body of Knowledge (SWEBOK Guide): Version 3.0 (PDF). IEEE Computer Society. ISBN 978-0-7695-5166-1. Archived (PDF) from the original on 15 May 2020. Retrieved 14 March 2022
8. Chen, M. J., & Wilson, T. Indirect Competition: Strategic Considerations.
9. Craft, R. L., ESPINOZA, J., & CAMPBELL, P. L. (2001). Source Code Assurance Tool: Preliminary Functional Description (No. SAND2001-3092). Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia National Lab.(SNL-CA), Livermore, CA (United States).
10. Crussell, J., Gibler, C., Chen, H. (2012). Attack of the Clones: Detecting Cloned Applications on Android Markets. In: Foresti, S., Yung, M., Martinelli, F. (eds) Computer Security – ESORICS 2012. ESORICS 2012. Lecture Notes in Computer Science, vol 7459. Springer, Berlin, Heidelberg. https://doi-org.proxy.library.uu.nl/10.1007/978-3-642-33167-1_3
11. Eickelmann, N. (2004). Measuring maturity goes beyond process. IEEE Software, 21(4), 12–13. doi:10.1109/ms.2004.21
12. Farshidi, S., Jansen, S., de Jong, R., &#38; Brinkkemper, S. (2018). A decision support system for software technology selection. Journal of Decision Systems, 27(sup1), 98–110. https://doi.org/10.1080/12460125.2018.1464821
13. Golubev, Y., Eliseeva, M., Povarov, N., & Bryksin, T. (2020, June). A study of potential code borrowing and license violations in java projects on Github. In Proceedings of the 17th International Conference on Mining Software Repositories (pp. 54-64).
14. Holzmann, G. J. (2015). Code Inflation. IEEE Softw., 32(2), 10-13.
15. Islam, M. R., Zibran, M. F., & Nagpal, A. (2017, November). Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (pp. 20-29). IEEE.
16. Jang, J., Agrawal, A., & Brumley, D. (2012, May). ReDeBug: finding unpatched code clones in entire os distributions. In 2012 IEEE Symposium on Security and Privacy (pp. 48-62). IEEE.
17. Joukov, N., Tarasov, V., Ossher, J., Pfitzmann, B., Chicherin, S., Pistoia, M., & Tateishi, T. (2011, May). Static discovery and remediation of code-embedded resource dependencies. In 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops (pp. 233-240). IEEE.

18. K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," Proceedings of the Fourth Working Conference on Reverse Engineering, 1997, pp. 44-54, doi: 10.1109/WCRE.1997.624575.
19. Kim, S., Woo, S., Lee, H., & Oh, H. (2017, May). Vuddy: A scalable approach for vulnerable code clone discovery. In 2017 IEEE Symposium on Security and Privacy (SP) (pp. 595-614). IEEE.
20. Kolers, P. A. Duchnicky, R. L., and Ferguson, D. C. (1981). Eye movement measurement of readability of CRT displays. Human Factors, 23,517-527.
21. Kupsch, J. A., Heymann, E., Miller, B., & Basupalli, V. (2017). Bad and good news about using software assurance tools. Software: Practice and Experience, 47(1), 143-156.
22. Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., & Hu, J. (2016, December). Vulpecker: an automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications (pp. 201-213).
23. LL(1) parsers. (n.d.). Retrieved May 23, 2022, from http://www.cs.ecu.edu/karl/5220/spr16/Notes/Top-down/LL1.html
24. M. Orrú, E. Tempero, M. Marchesi and R. Tonelli, "How Do Python Programs Use Inheritance? A Replication Study," 2015 Asia-Pacific Software Engineering Conference (APSEC), 2015, pp. 309-315, doi: 10.1109/APSEC.2015.51.
25. McCormick, M. (2012). Waterfall vs. Agile methodology. MPCS, N/A.
26. Pierro, G. A., & Tonelli, R. (2021, March). Analysis of source code duplication in Ethereum smart contracts. In 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 701-707). IEEE.
27. Process Maturity Profile. Software CMM® 2004 MidYear Update. Software Engineering Institute, Carnegie Mellon University, August 2004.
28. Rahman, S. (2021). Aspect of Code Cloning Towards Software Bug and Imminent Maintenance: A Perspective on Open-source and Industrial Mobile Applications (Doctoral dissertation, University of Saskatchewan).
29. Rashid, M., Clarke, P. M., & O'Connor, R. V. (2019). A systematic examination of knowledge loss in open-source software projects. International Journal of Information Management, 46, 104-123.
30. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., ... & McConley, M. (2018, December). Automated vulnerability detection in source code using deep representation learning. In 2018 17th IEEE international conference on machine learning and applications (ICMLA) (pp. 757-762). IEEE.
31. Song, X., Yu, A., Yu, H., Liu, S., Bai, X., Cai, L., & Meng, D. (2020, December). Program Slice based Vulnerable Code Clone Detection. In 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) (pp. 293-300). IEEE.
32. Westland, J. (2006). The project management life cycle: a complete step-by-step methodology for initiating planning, executing and closing the project.
33. Wheeler, Reddy, Fong. (2018, July 2). Securely Using Software Assurance (SwA) Tools in the Software Development Environment. DTIC. https://apps.dtic.mil/sti/citations/AD1103836
34. Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14. https://www.wohlin.eu/ease14.pdf

Snowballing Literature [Sx]

1. A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In Proc. SP, pages 692–708. IEEE, 2015.
2. B. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", in Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, (1995)

3. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 872–881. ACM (2006)
4. Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in Ethereum. In 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pages 9–15. IEEE, 2020.
5. Wahidur Rahman, Yisen Xu, Fan Pu, Jifeng Xuan, Xiangyang Jia, Michail Basios, Leslie Kanthan, Lingbo Li, Fan Wu, and Baowen Xu. Clone detection on large scala codebases. In 2020 IEEE 14th International Workshop on Software Clones (IWSC), pages 38–44. IEEE, 2020.

Non-peer-reviewed Literature and Sources [NPx]

1. A quote from The Art of War. (n.d.). Retrieved March 28, 2022, from https://www.goodreads.com/quotes/17976-if-you-know-the-enemy-and-know-yourself-you-need
2. ANTLR. (n.d.). Retrieved May 22, 2022, from https://www.antlr.org/
3. Apache Cassandra (https://cassandra.apache.org/): a free and open-source, distributed, wide-column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Documentation can be found at https://cassandra.apache.org/doc/latest/
4. Asite, A. A. (n.d.). Digitization, Digitalization, and Digital Transformation – What's the Difference? Retrieved March 20, 2022, from https://www.asite.com/blogs/digitization-digitalization-and-digital-transformation-whats-the-difference
5. Automated Vulnerability Analysis System (2017). SOSCON. Retrieved June 30, 2022, from https://www.sosconhistory.net/2017/pdf/day2_1530_1.pdf
6. Black Duck User Guide. (n.d.). Synopsys. Retrieved July 5, 2022, from https://testing.blackduck.synopsys.com/doc/pdfs/user_guide.pdf
7. Build software better, together. (n.d.). GitHub. Retrieved June 18, 2022, from https://github.com/
8. Engine overview. (n.d.). Semgrep. Retrieved June 16, 2022, from https://semgrep.dev/docs/writing-rules/data-flow/overview/#design-trade-offs
9. G2. (n.d.). About. G2 Culture. Retrieved April 21, 2022, from https://company.g2.com/about
10. IoTcube. (n.d.). CSSA. Retrieved July 9, 2022, from https://iotcube.net/userguide/manual/hmark
11. IoTcube. (n.d.). CSSA. Retrieved June 18, 2022, from https://iotcube.net/process/type/wf1
12. IoTcube. (n.d.). CSSA. Retrieved May 30, 2022, from https://iotcube.korea.ac.kr/
13. Kashyap, N. (2020, March 4). GitHub's Path to 128M Public Repositories. Towards Data Science. https://towardsdatascience.com/githubs-path-to-128m-public-repositories-f6f656ab56b1
14. Matser, M. (2022, February 15). SearchSECO Presentation (Slides). SearchSECO.
15. Mitchell, G. (n.d.). How much data is on the internet? BBC Science Focus Magazine. Retrieved March 12, 2022, from https://www.sciencefocus.com/future-technology/how-much-data-is-on-the-internet
16. Partners – SecureSECO. (n.d.). Retrieved May 30, 2022, from https://secureseco.org/about/partners/
17. PDF Reports. (n.d.). SonarQube Docs. Retrieved July 10, 2022, from https://docs.sonarqube.org/latest/project-administration/portfolio-pdf-configuration/
18. Plans &; Pricing. (n.d.). Sonar. Retrieved July 5, 2022, from https://www.sonarsource.com/plans-and-pricing/
19. Plugin basics. (n.d.). SonarQube Docs. Retrieved July 10, 2022, from https://docs.sonarqube.org/latest/extend/developing-plugin/
20. returntocorp. (n.d.). GitHub - returntocorp/semgrep: Lightweight static analysis for many languages. Find bug variants with patterns that look like source code. GitHub. Retrieved May 31, 2022, from https://github.com/returntocorp/semgrep/
21. ScanCode. (2021, August 6). NexB. https://nexb.com/scancode/

22. SearchSECO Dashboard. (n.d.). Grafana. Retrieved May 16, 2022, from https://secureseco.science.uu.nl
23. SecureSECO. (n.d.). GitHub - SecureSECO/SearchSECOController at development. GitHub. Retrieved May 16, 2022, from https://github.com/SecureSECO/SearchSECOController/blob/development/Documentation/Documentation.pdf
24. Semgrep CI overview. (n.d.). Semgrep. Retrieved June 14, 2022, from https://semgrep.dev/docs/semgrep-ci/overview/
25. Semgrep. (n.d.). Retrieved May 30, 2022, from https://semgrep.dev/
26. Sigrid®. (2019, August 1). SIG | Getting Software Right for a Healthier Digital World. https://www.softwareimprovementgroup.com/solutions/sigrid-software-assurance-platform/
27. Snyk. (2020, October 21). Snyk. https://snyk.io/
28. Software Composition Analysis. (n.d.). Black Duck Software. Retrieved May 30, 2022, from https://www.blackducksoftware.com/
29. Software-Improvement-Group. FAQ (n.d.). Retrieved May 31, 2022, from https://github.com/Software-Improvement-Group/sigridci/blob/main/docs/faq.md#where-can-i-find-more-information-about-your-metrics
30. SonarCloud. (n.d.). SonarCloud. Retrieved May 31, 2022, from sonarcloud.io
31. SonarQube Documentation. (n.d.). SonarQube Docs. Retrieved July 10, 2022, from https://docs.sonarqube.org/latest/
32. squizz617. (n.d.). Java language support · Issue #12 · squizz617/vuddy. GitHub. Retrieved June 13, 2022, from https://github.com/squizz617/vuddy/issues/12
33. Stack Overflow Developer Survey (2020). Stack Overflow. Retrieved May 5, 2022, from https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-professional-developers
34. Supported languages. (n.d.). Semgrep. Retrieved May 31, 2022, from https://semgrep.dev/docs/language-support/
35. Synopsys Software Integrity Community. (n.d.). Retrieved May 31, 2022, from https://community.synopsys.com/s/
36. Synopsys. (n.d.). Black Duck Software Composition Analysis.
37. Thesaurus.com - The world's favorite online thesaurus! (n.d.). Thesaurus.Com. Retrieved March 21, 2022, from https://www.thesaurus.com/
38. Tortellini. (n.d.). Research Software Directory. Retrieved May 16, 2022, from https://www.research-software.nl/software/tortellini-github-action
39. What is a software bill of materials (SBOM)? (2022, March 16). Synopsys. https://www.synopsys.com/blogs/software-security/software-bill-of-materials-bom/