A Static Analysis Platform for Investigating Security Trends in Repositories

Tim Sonnekalb, Christopher-Tobias Knaust,
Bernd Gruner, Clemens-Alexander Brust
Institute of Data Science
German Aerospace Center (DLR)
Jena, Germany
{firstname.lastname}@dlr.de

Lynn von Kurnatowski, Andreas Schreiber
Institute for Software Technology
German Aerospace Center (DLR)
Cologne, Germany
{firstname.lastname}@dlr.de

Thomas S. Heinze
Cooperative University Gera-Eisenach
Gera, Germany
thomas.heinze@dhge.de

Patrick Mäder
Data-intensive Systems and Visualization Group
Technical University Ilmenau
Ilmenau, Germany
patrick.maeder@tu-ilmenau.de

Abstract—Static analysis tools come in many forms and configurations, allowing them to handle various tasks in a (secure) development process: code style linting, bug/vulnerability detection, verification, etc., and adapt to the specific requirements of a software project, thus reducing the number of false positives.

The wide range of configuration options poses a hurdle in their use for software developers, as the tools cannot be deployed out-of-the-box. However, static analysis tools only develop their full benefit if they are integrated into the software development workflow and used on regular. Vulnerability management should be integrated via version history to identify hotspots, for example.

We present an analysis platform that integrates several static analysis tools that enable Git-based repositories to continuously monitor warnings across their version history. The framework is easily extensible with other tools and programming languages. We provide a visualization component in the form of a dashboard to display security trends and hotspots. Our tool can also be used to create a database of security alerts at a scale well-suited for machine learning applications such as bug or vulnerability detection.

Index Terms—static analysis, software security, automatic analysis, SAST tool database, security dashboard

I. INTRODUCTION

Static Analysis Security Testing (SAST) tools provide powerful means for finding bugs and code flaws in written code. These tools have a wide range of features and range from code style linting to formal verification of the underlying code. They allow for very detailed configuration settings and custom rules to account for the characteristics of a software project and maximize the detection rate.

In contrast, few open source projects incorporate SAST tools into their development workflows, as found by a large-scale study [1]. The study also showed that when SAST tools were used, the default configuration was usually not adjusted. When custom rules were written, they correspond to only a small percentage of the total rule value. In addition, this study reveals that most configuration files are never adjusted again over the entire development period. In this way, static analysis

cannot work effectively because the configuration does not match the requirements of the software.

Johnson et al. [2] found in interviews that developers find using these tools too burdensome and do not consider using bug tracking tools due to time constraints. Some would use such tools if they were already available in the development process and if the tool output is well-presented.

To dsvr developers most of the effort associated with using these code review tools, we develop an online static analysis platform that can be easily integrated into Git-based processes, run at regular intervals, and present the result via a dashboard. In addition, our analysis platform can be used to mine SAST warnings at scale and leverage a comprehensive database for further data science applications such as empirical vulnerability detection.

We design a SAST analysis platform that satisfies the following aspects:

Continuous Online Monitoring Our analysis checks for new commits for a given repository URL at regular intervals and analyzes them to support DevOps.

High Coverage Since we apply multiple SAST tools with different modes of operation, the probability of detecting a bug is higher, minimizing the number of false negatives [13].

Virtualization We deploy the analysis platform as Docker image to run it in an OS-independent manner.

Modular Design Our implementation allows easy extension with various SAST tools and programming languages.

Git-based We use Git as a widely used version control system and can apply our analysis to all Git-based repositories.

Mining/Visualization The analysis results are written to a database and visualized in a dashboard.

Schreiber et al. [3] tested this platform using the open source repositories of the German Covid contract tracing apps as case study. They also provided a database [4] which can be used

with our dashboard. Schreiber et al. [5] discussed elements for visualizing software components, including SAST warnings, which we used as the basis for our dashboard. Our contribution in this work is the implementation and publication of the code for further work. We provide the source code of our analysis platform, the dashboard and the Dockerfile via Gitlab [1].

II. SAST Platform Design

The components of our analysis platform are mapped in Figure 1. Starting from a Git URL, the script downloads the entire repository and extracts the repository state at a given commit. When the analysis is first run, the commits are downloaded and analyzed in chronological order. On all subsequent runs of the analysis, any new commits added since the last run are added to the analysis queue. The results are parsed and written to a database. In addition, the results can be displayed in our dashboard or retrieved from the database with other visualization tools such as Grafana [2].

A. SAST Tools

Our platform currently supports five SAST tools. Table I shows the implemented tools, each with their category, the analyzed programming languages, and the predefined config-uration. The categories range from coding rules to formal verification. As for the programming languages, in this work we mainly focus on the analysis of Android and Java. We selected these tools based on their suitability for analyzing the source code of the app.

We adapted the configuration of the tools (see Table I) to is-sue the security-related alerts in order to detect vulnerabilities. The user of this tool can adapt it to project-specific conditions. The generation of the best working configuration is still done manually by the developer, but with our platform, it is possible to test it across the entire development with a few simple steps.

The analysis itself runs in a Docker container to be platform-independent. To start the analysis, the script only needs the Git URL and the programming languages to be analyzed. We provide instructions to run the script as a systemd service to keep the database updated regularly. Our scripts are written in Python, so they can be easily adapted for further data science applications. We can include multiple repositories in the analysis queue. The execution of the tools does not affect the output of the results in the dashboard, as both parts of the platform run independently. The analysis service ideally runs on a server, while the output takes place in the browser of a user client.

The tools we use have different requirements for the ana-lyzed software. Either they use pure source code, compiled object code, or the analysis is directly integrated into the compilation process. Our platform currently supports Gradle and Maven builds by checking for the presence of the appro-priate files in the main folder or sub-folders and executing the associated build command. After each commit, we perform

a clean build to ensure all changes are applied correctly. Additional SAST tools can be easily added by including the installation of the tool in the Dockerfile, defining the initialization of the analysis with the configuration, and writing a parsing method. We provide instructions for this in our repository.

We pay special attention to fault tolerance. If a SAST tool run fails or is aborted, the analysis is repeated on the next iteration. If the analysis repeatedly ends with an error code, the commit with that tool is skipped.

B. SAST Database

We use a relational database for storing the SAST warnings, repository, and tool information. The warnings from multiple repositories, including all tool information and their configu-rations, are stored in six related tables. This allows us to query and analyze the data through the backend of our dashboard. We use the commit hash or snapshot ID as the code state. The relational model can also be used as an interface to other data mining and analysis tools, as the authors have shown for the combined analysis of vulnerability and provenance data [3].
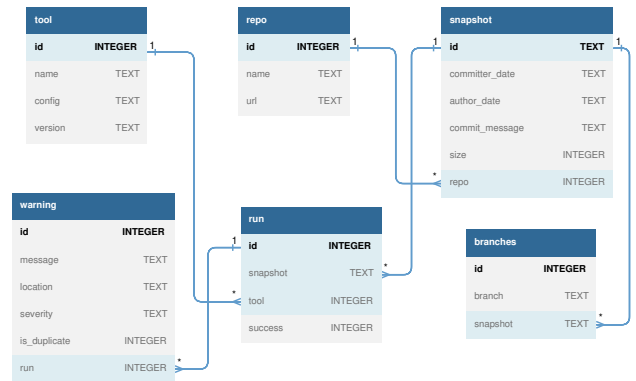

Fig. 2. Relational schema of our SAST database.

Figure 2 shows the database schema. It includes the follow-ing tables:

Repo contains the commited repository name and its Git URL.

Snapshot or commit contains the commit hash, commit date, author name, commit message, and code size mea-sured by lines of code.

Branches assigns a branch name to a specific *snapshot*.

Tool corresponds to a SAST tool, with a name, configuration and version number.

Run corresponds to an execution of a *tool* on a given *snapshot* and confirms a *success* flag which confirms a successful tool run or indicate errors [3].

Warning is a collection of all SAST warnings encountered with a message, location as file path, severity (if avail-able) and a duplicate check field.

---

[3]The user can read detailed error messages in the log files
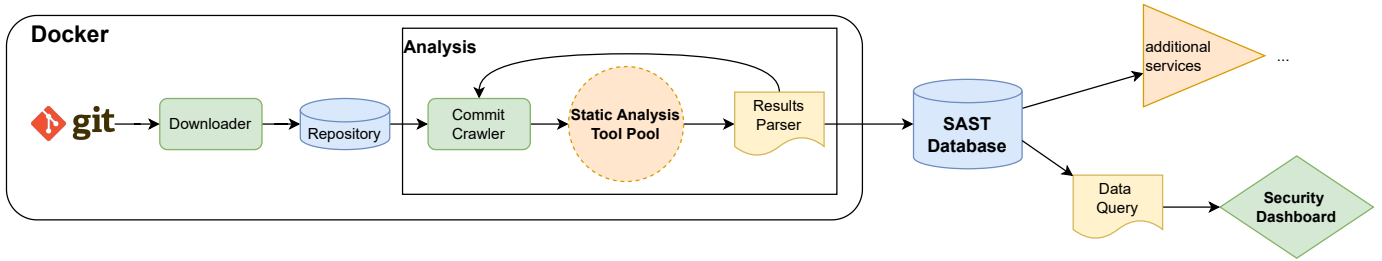
Fig. 1. Platform overview.

TABLE I
INCLUDED STATIC ANALYSIS TOOLS.

SAST tool	Category	Languages	Configuration
Flowdroid	taint analysis	Android	*java -jar soot.jar -s ./flowdroid/SourcesAndSinks.txt -o report.xml*
Infer	formal verification	Java, Android, C, C++, iOS	*infer run --results-dir {resultDir} --no-fail-on-issue -- {compileConfig}*
MobSF	various	Android, iOS, Windows	*mobsf --apikey {Key} upload {Apk} > mobsf_report.json*
PMD	coding rules	Java, JavaScript	*pmd -R rulesets/java/sunsecure.xml -failOnViolation false -f json*
Xanitizer	taint analysis	Java, Scala, JavaScript, TypeScript	*xanitizer generateDetailsInFindingsListReport=True overwriteConfigFile=True*

The data types of the table fields are either text or integer. We use 1-to-n relationships between the tables to be able to model the requirements.

After a tool run is complete, the results are returned depending on the tool, for example as a json file, and parsed appropriately so that they can be inserted with a SQL insert statement. The output of the tools, such as the warning message, location, and severity, varies by tool. It would be possible to normalize SAST warnings using, for example, the Static Analysis Results Interchange Format (SARIF) [6]. We do not normalize the values before writing them to the database, as we leave it open for future use.

The data model design we chose supports the visualization use cases we have for our dashboard. In the *warning* table, we can easily filter the warnings, for example, by location or severity. The core table is *run*, which has relationships with *warning*, *repo*, and *snapshot*. An insight into the history of warnings can be obtained with a join of *warning* and *run.id* and a select of the *snapshot.author_date*. In addition, it is possible to use the SAST data and correlate it with software metrics via the version history for deeper analysis, i.e., code churn correlated with the number of vulnerabilities [7].

III. DASHBOARD

We create a dashboard to visualize the security findings from the SAST database. Figure 3 shows an example. The frontend should help the developers filter warnings more quickly or identify particularly buggy software modules. It is also intended to show a security trend, indicating whether SAST tools have been used in the past and whether the overall code quality has improved over the version history. The latter plays an important role, especially for legacy code. Among others, the dashboard currently provides three basic visualizations:

Version History Line Chart This graph shows the number of SAST warnings in the version history to provide an overall security trend. This diagram is shown at the top left in Figure 3.

Alarm Type Tree Chart This chart displays the most common alert types, either based on their warning message or a vulnerability type such as CWE (if present). This diagram is shown at the top right in Figure 3.

Hotspot Module Map This mapping gives the viewer the ability to identify the hotspots, i.e., modules with a particularly high number of alerts.

Schreiber et al. provide a discussion of the visualization components in [3] and [5]. We visualize the results per tool because we cannot aggregate the values without normalizing the reports. Simple aggregation, such as counting warnings per code location, would be possible, but we cannot rule out the possibility that the report contains different bugs without parsing them. We also list the contents of the database to read the entire message at the bottom of Figure 3. We plan to integrate a direct link to the code location in an IDE.

IV. LIMITATIONS AND FUTURE WORK

Our analysis tool is ready-to-use for a huge number of Git-based projects. However, the platform currently focuses on Java projects and therefore supports only a subset of the respective SAST tools. In the future, we plan to extend this analysis platform to include more tools and other programming languages.

At the moment, we focus on collecting various warnings about the version history of a repository. We do not perform a qualitative evaluation or normalization of SAST reports. Depending on which tools are used, there may be an increase in false positives but also true positives with multiple SAST tools.
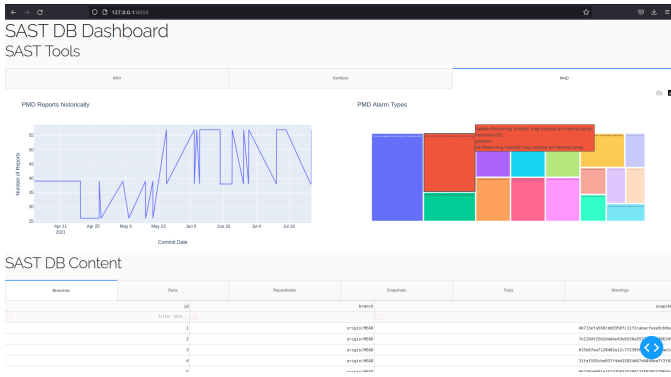
Fig. 3. Dashboard showcase.

Park et al. [8] developed a method to reduce the number of false positives by including snapshots of execution environments. Such a method could also be included in our platform by extending it with dynamic analysis tools. In addition, a transformer network could help to prioritize the SAST warnings and thus provide filter functions for the developer [9]. Another next step is to plan and conduct a user study to help evaluate which elements in our dashboard are helpful to developers or can be improved. We would like to conduct this with focus groups to see which representations are most effective within a developer team. This method is very effective for improvement for UX design of dashboards [10]. This also allows project-specific visualizations to be realized.

Static Analysis is used as a ground truth for machine learning applications [11]. By running our analysis platform on a large scale, it is possible to create a comprehensive dataset of SAST warnings. It can also be used to benchmark SAST tools. In the future, we will use the SAST database to aggregate messages and additionally provide security and quality assessment by calculating code and repository metrics.

V. RELATED WORK

Code review tools can be divided into SAST, DAST, IAST, and RASP tools. SAST tools are a form of white-box testing without running the code in a production environment. They are well suited for code checking during development or vulnerability scanning for quality assurance. In contrast, DAST (Dynamic Analysis Security Testing) tools correspond to black-box testing and can detect run time bugs by executing the analyzed software. IAST (Interactive Application Security Testing) tools combine SAST and DAST and perform code scans during runtime. RASP (Runtime Application Self-Protection) is a technique with a runtime instrumentation technique that can block cyberattacks. Our platform is solely based on SAST tools, but an extension with other tools is possible.

ReviewBot is a tool by Balachandran [12], that combines the results of multiple static analysis tools and automatically generates a code review. They collect user feedback for the reviews by conducting interviews and find that the majority of

the automatically generated comments are useful. The tools are either linting or pattern-detection tools. We could not find an implementation to replicate this tool.

Nunes et al. [13] combine SAST tools for the application of web security, specifically SQLi and XSS vulnerabilities in WordPress plugins. They used a database of historic vulnerabilities for evaluation of the tool's performance. Their pipeline uses metrics to rank the tools and combine the outputs. Our platform works with the current state of Git repositories, not historical data. Also, we are not limited to specific vulnerability types; we let the tools detect all of them. However, we also could not find their code for reproducibility.

Batyuk et al. [14] build an analysis service for Android binaries that generate detailed reports. The binaries are downloaded from the Android Market, decompiled, and analyzed through data mining operations. Our analysis service starts with the source code, not the released version of an app, and therefore uses different SAST tools.

Scrub [15], short for Source Code Review User Browser, is a tool that combines classic peer code review with machine-generated analysis. It provides a user interface similar to our dashboard where developers can compare the SAST results with their manually reviewed code. We did not plan to include manually reviewed code in our overview, as it is always biased by developer experience.

Meanwhile, Gitlab [16] and GitHub [17] provide security scanning with SAST tools built into CI/CD. The advantage of such an analysis as ours is its high customizability and a database as output that can be used for more detailed and individual evaluation. It is also a novelty to be able to test different tool configurations with our platform and thus visualize security trends over time or over different software submodules.

Arai et al. [18] use a gamification approach to motivate developers to use SAST tools and remove the warnings from the code. FindBugs' bug reports are classified into bug patterns, and a score is calculated per developer and per team. The score improves after corrections are made in the next iteration of the commit. This could be an interesting extension to our dashboard.

VI. CONCLUSION

We provide an open-source static analysis platform as a Docker container that is easy to use and extensible with minimal effort. We also provide a dashboard that shows trends in version history. We published our code for reproducibility.

The novelty of our method is that it is based on the entire development history of the project, which makes it possible to derive security trends and identify hotspots. Tool configurations can be tested more easily by looking at how they have performed so far on the repository history. Our analysis platform can help software developers in their daily work for the code review process and evaluate the version history of software projects for security and vulnerability analysis.

REFERENCES

[1] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 470–481, IEEE, 2016.

[2] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, IEEE, 2013.

[3] A. Schreiber, T. Sonnekalb, T. S. Heinze, L. von Kurnatowski, J. M. Gonzalez-Barahona, and H. Packer, "Provenance-based security audits and its application to covid-19 contact tracing apps," in *Provenance and Annotation of Data and Processes* (B. Glavic, V. Braganholo, and D. Koop, eds.), (Cham), pp. 88–105, Springer International Publishing, 2021.

[4] T. Sonnekalb, "SAST database of repository Luca App Android." https://doi.org/10.5281/zenodo.5036046, June 2021.

[5] A. Schreiber, T. Sonnekalb, and L. v. Kurnatowski, "Towards visual analytics dashboards for provenance-driven static application security testing," in *2021 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 42–46, 2021.

[6] G. Paul Anderson, "Static Analysis Results: A Format and a Protocol: SARIF & SASP," 2018.

[7] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[8] J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of javascript web applications in the wild," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 61–70, IEEE, 2016.

[9] T. Sonnekalb, T. S. Heinze, and P. Mäder, "Deep security analysis of program code," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–39, 2022.

[10] L. D. Roberts, J. A. Howell, and K. Seaman, "Give me a customizable dashboard: Personalized learning analytics dashboards in higher education," *Technology, Knowledge and Learning*, vol. 22, pp. 317–333, 2017.

[11] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "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, 2018.

[12] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 931–940, IEEE, 2013.

[13] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "On combining diverse static analysis tools for web security: An empirical study," in *2017 13th European Dependable Computing Conference (EDCC)*, pp. 121–128, 2017.

[14] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications," in *2011 6th International Conference on Malicious and Unwanted Software*, pp. 66–72, 2011.

[15] G. J. Holzmann, "Scrub: A tool for code reviews," *Innov. Syst. Softw. Eng.*, vol. 6, p. 311–318, dec 2010.

[16] "Static Application Security Testing (SAST) - Gitlab." https://docs.gitlab.com/ee/user/application_security/sast/. Accessed: 2022-12-13.

[17] "Features - Security - Github." https://github.com/features/security. Accessed: 2022-12-13.

[18] S. Arai, K. Sakamoto, H. Washizaki, and Y. Fukazawa, "A gamified tool for motivating developers to remove warnings of bug pattern tools," in *2014 6th international workshop on empirical software engineering in practice*, pp. 37–42, IEEE, 2014.