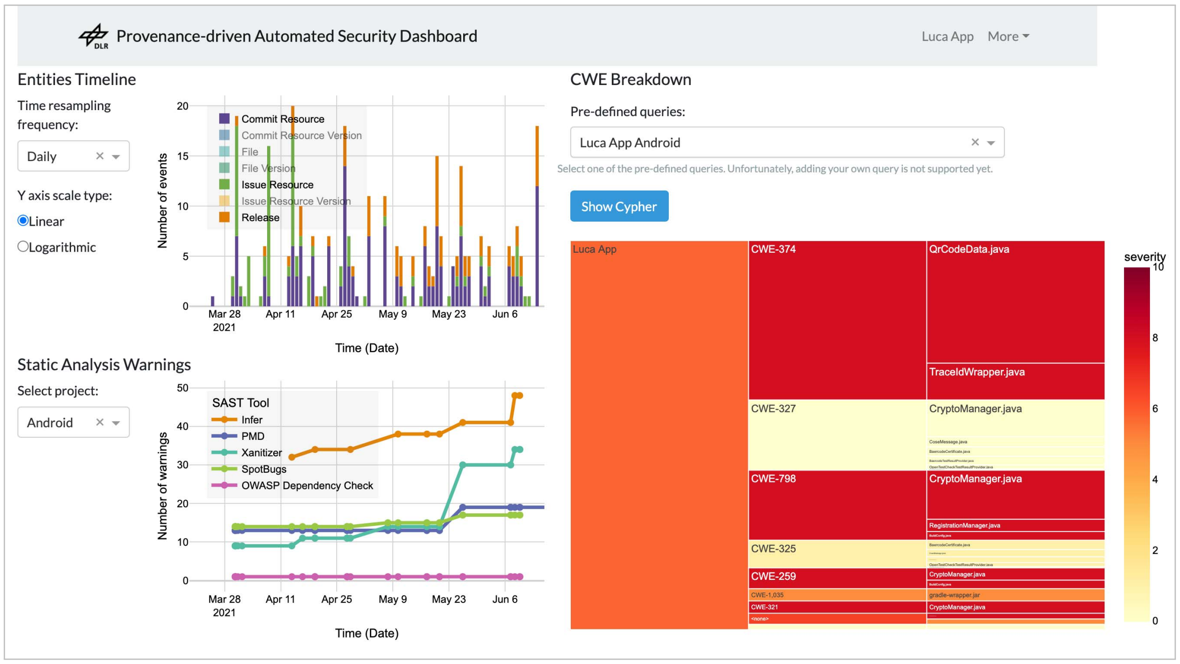
Towards Visual Analytics Dashboards for Provenance-driven Static Application Security Testing

Andreas Schreiber*   Tim Sonnekalb†   Lynn von Kurnatowski‡

German Aerospace Center (DLR)



Time resampling frequency: Daily

Y axis scale type: [✓ Linear] Logarithmic

Select project: Android

Pre-defined queries: Luca App Android

Select one of the pre-defined queries. Unfortunately, adding your own query is not supported yet.

Figure 1: Interactive web-based dashboard, which allows to analyze and overview the results of security analyses of the source code ("Static Analysis Warnings" and "CWE Breakdown") based on the provenance information of the software development process ("Entities Timeline").

Abstract

The use of static code analysis tools for security audits can be time consuming, as the many existing tools focus on different aspects and therefore development teams often use several of these tools to keep code quality high and prevent security issues. Displaying the results of multiple tools, such as code smells and security warnings, in a unified interface can help developers get a better overview and prioritize upcoming work. We present visualizations and a dashboard that interactively display results from static code analysis for "interesting" commits during development. With this, we aim to provide an effective visual analytics tool for code security analysis results.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Visual analytics; Security and privacy—Software and application security—Software security engineering

*e-mail: andreas.schreiber@dlr.de
†e-mail: tim.sonnekalb@dlr.de
‡e-mail: lynn.kurnatowski@dlr.de

1 Introduction

Static code analysis attempts to predict potential runtime behavior of software without necessarily executing the software; and without using any input data [2]. It can help to improve software by finding weaknesses early in the software development process.

Software development is a highly complex process, in which usually a team of software developers use numerous tools to develop a software as error-free as possible in a short time. To understand the software development process and to be able to make statements about quality, reliability and trustworthiness of the software product, one can record and analyze the *provenance* [14] of the development process or the produced software artifacts.

To better understand the relationships between the development process and code quality based on static code analysis, we combine the two data sources—provenance information and results of static code analysis tools—and evaluate them using analytical and visual methods. With the result of such analysis, we aim to assess the reliability of software systems with high criticality (e.g., space mission control systems) or the trustworthiness of software with high societal relevance (e.g., COVID-19 contact tracing apps).

The basic principle of our provenance-driven code security analysis is to find and select relevant or "interesting" activities in the development process by making queries on provenance information and then evaluating the results of static code analysis at the times of these activities [25, 27].

Our *main contribution is a first visual dashboard for provenance-driven security analysis*. The use of provenance from all relevant processes, which are stored in a *standardized provenance data model*, distinguishes our approach from other approaches to security analysis and auditing.

We describe our current status towards being able to efficiently and effectively perform provenance-driven security analysis using visual methods and interactive dashboards in the following structure: We briefly describe which methods and tools we use for static code analysis (Sect. 2). We give some basic information about provenance and specifically how we record and store provenance of software development processes (Sect. 3). We show our first visualizations and a dashboard we are developing to perform interactive visual analytics for code security analysis (Sect. 4). Finally, we summarize related work, especially on visualization of software artifacts (Sect. 5).

2 Static Application Security Testing (SAST)

Static code analysis—or *Static Application Security Testing* (SAST)—is a static software testing procedure performed at compilation time to detect certain types of errors in the source code. Continuous use of this method during the software development process can reveal early indicators of bugs, code smells, defects, or vulnerabilities [18]. There are various tools, ranging from *linters*, which check the occurrence of textual or syntactical information on pre-defined code rules, up to *full-fledged verification tools*, which formally prove specific criteria in the code. These criteria cover multiple aspects of source code, including cryptography checks, taint-related problems (e.g., data and resource leaks), use of hard-coded credentials or null-pointer errors. Typical obstacles for using static code analysis tools in practice are the false positive ratio, understandable and actionable analysis results or the integration in the development process, making their use by software developers work-intensive [10]. Usually, it is not enough to use only one tool [20]. With our approach, we want to address parts of these issues, making these tools more user-friendly.

For our analyses we use several tools for Java; both tools with an open source license and commercial tools (Table 1).

Table 1: Used static analysis tools.

Static analysis tool	**Tool category**
Flowdroid	taint analysis
Xanitizer	taint analysis
Infer	formal verification
Spotbugs/FindSecBugs	coding rules
PMD	linter, code smells
OWASP Dependency Check	dependencies

3 Provenance in Software Engineering

Provenance describes the people, institutions, entities, and activities involved in creating, processing, or providing data [16]. Provenance can be formally expressed in different ways. We use the W3C specification PROV [15], which among other definitions defines the provenance data model PROV-DM [16]. The core structure of PROV-DM relies on the definition of the model class elements *entities*, *activities*, and *agents* that are involved in producing a piece of data or artifact and on definitions of *relations* to relate these class elements.

For software development processes based on `git` repositories, we extract *retrospective provenance* [13] with data mining on the repositories; in the case of GitHub and GitLab, this also includes provenance information from the respective issue trackers and release systems. The resulting provenance data then contains all activities (e.g., commits, issues changes, releases), the generated or changed entities (e.g., source code files or issues), and involved agents (e.g., developers, testers, or users) along with their relations. Our tool GitLab2PROV [3, 24] uses the GitLab API to generate the provenance information in the form of the text representation PROV-JSON, which is converted to other formats for further analysis and visualization and imported as a property graph into the graph database Neo4j (Fig. 2).

4 Visualization and Interactive Dashboard

As a case study, we illustrate our approaches to interactive visualization with the *luca App*[1]. The luca App is a mobile app for providing data for contact tracing and risk contact notification during a pandemic. In Germany, there is a controversial discussion about the app, which currently makes it a socially relevant software application. Many IT security experts criticize the app for its security gaps and data protection shortcomings [17, 28]. Nevertheless, many German states have bought it and made it mandatory for checking in at restaurants and stores. For this reason, the app and its development process are interesting objects of investigation.

Most of the development of the *luca App* is done publicly on GitLab[2] in eight repositories. Development began in February 2021, with 13 people committing changes to git and 42 people contributing changes to issues as of early June[3] [23].

Similar to our approach for the CWA app [27], we conduct the following steps (Fig. 2):

Step 1: Query the provenance graph using Cypher[4] for a distinct list of commits.

Step 2: Clean—and optionally filter—the query result to get a clean list of commit hashes.

Step 3: Query the SAST database [26] for each of the commit hashes from Step 2.

Step 4: Analyze the results from Step 3. For example, by summarizing, classifying, or visualizing them.

In practice, we use Python to submit the Cypher query to Neo4j and store the result in a Pandas `DataFrame`, on which we do Step 2. Then, we submit an SQL query on the SAST database using Pandas' `read_sql_query` method for each of the commit hashes, which returns—for example—the number of warnings reported for changed files during the related commits.

4.1 SAST Results

The results of SAST tools are presented very differently. Simple tools simply output the results on the terminal; advanced tools provide a graphical user interface or are integrated as plug-ins into IDEs. The granularity and accuracy of the results also varies widely. For example, there is a difference to the users of the tools whether an entire function is marked as "insecure" or only a single line. The precision often depends on the type of verification method used; a formal verification is more accurate than coding rules by linters (Sect. 2).

4.1.1 Number of Warnings

The *number of warnings* issued by the individual tools is shown as a line chart (Fig. 3). We see that the trend of the individual traces either remains at about the same level or increases over the development time. The cumulative number of warnings increases, which can either mean that the development team does not or rarely use static analysis tools or that there are many false positives in the results [22]. Both of these factors suggest using visualization to

[1]`https://www.luca-app.de`
[2]`https://gitlab.com/lucaapp`
[3]`https://cauldron.io/project/4448`
[4]`https://neo4j.com/developer/cypher/`

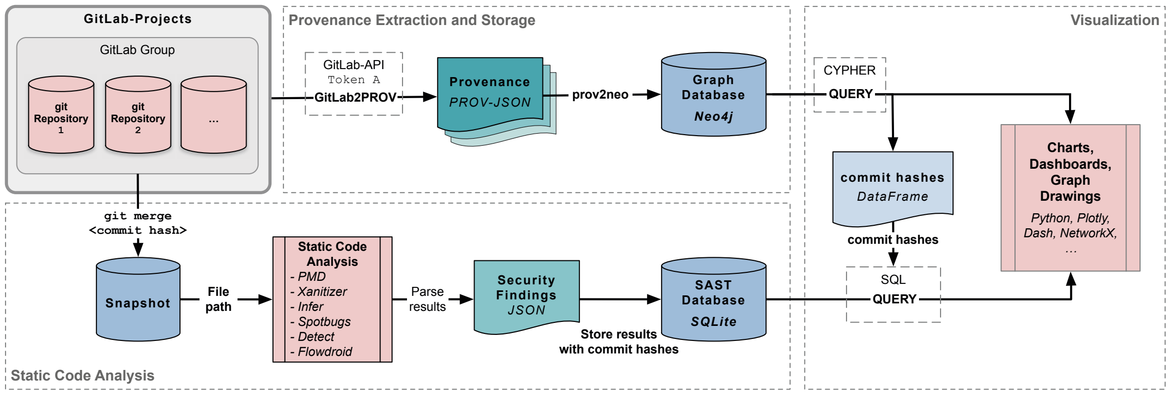


Figure 2: Workflow for the provenance-driven automated code analysis: provenance is extracted from git repositories on GitLab using our tool GitLab2PROV [24] to PROV-JSON files (one file for each repository) [23] and stored in a Neo4j database using our tool prov2neo [4]. The SAST database [26] is created by applying the various static code analysis tools (Table 1) to the code at specific commit. The data for the visual analytics are generated by matching database queries.

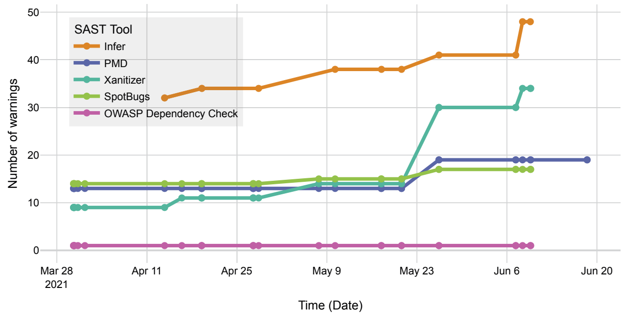


Figure 3: Security-related warnings in our SAST database queried for the *luca App for Android* (`https://gitlab.com/lucaapp/android`).

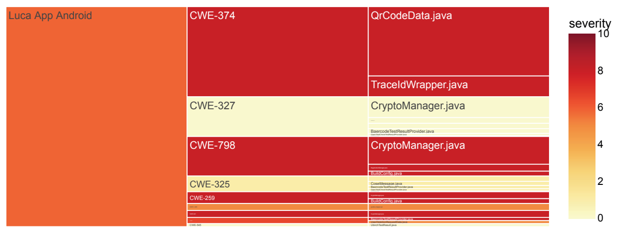


Figure 4: Visualization of the hierarchy of software projects or a selected set of snapshots, of the occurred warnings of static code analysis tools, and the affected files as an *Icicle chart*.

make the results more accessible and tangible to the developers, or to make false positives easier to find in the source code.

4.1.2 Hierarchy of Projects, Warnings, and Files

We visualize the hierarchy of selected projects and commits and the distribution of warnings and files as an *Icicle chart* [11] (Fig. 4). A user study has shown that an Icicle chart is preferable to a treemap because it has better performance for navigation and hierarchy understanding [30]. Visually, Icicle charts have a good balance between readability and compactness [12] and are well suited for identification of small values in multivariate data sets [31].

We categorized the SAST results according to the *Common Weakness Enumeration*[5] (CWE) category system for software weaknesses and vulnerabilities. Unfortunately, not all SAST tools expose the category or severity of the vulnerabilities, which is why we use the visualization as an Icicle chart only for a part of the warnings.

In our Icicle chart, we map the CWE classes to the second hierarchy level and the affected source code modules to the third hierarchy level. The size of the individual areas corresponds to the frequency of the warnings to easily identify warnings that occur often. We map the severity of the warnings to the color scale to easily see where serious warnings occur; we use a color scale ranging from light yellow for minor bugs to dark red for highly critical vulnerabilities[6].

Our way of presenting the warnings can be useful for development teams by either assigning the resolution of the warnings directly to developers of different experience levels according to the severity or—what we want to use in the future—assigning it based on the provenance (i.e., developers who have already worked on the code module in question).

4.2 Provenance and Events

To visually represent the provenance information we use several visualizations, such as graph visualizations, metrics visualizations (e.g., bar charts), time-oriented visualizations (e.g., Sankey charts), task-oriented and work process-oriented visualizations (e.g., Gantt charts), or hierarchy-oriented visualizations (e.g., treemap charts). To provide more interactivity and to be able to provide the visualizations in the form of a web application, we can compile the individual visualizations into a dashboard

We visualize the distribution of GitLab events (i.e., PROV Activities) as a timeline with bars—for example, summed (i.e., re-sampled) by date ranges such as 'Weekly' (Fig. 5). Events are further aggregated to events creating new commits, files, and issues (*Commit Resource*, *File*, *Issue Resource*) and events changing existing commits, files and issues (*Commit Resource Version*, *File Version*, *Issue Resource Version*). With only few events in February and March of 2021, the most activity takes place in the first three weeks of April. In the first week, the most amount of issues are generated. By far the most files are created, as well as changed in the third week of April. After these three weeks the activity decreases again, with mostly file version events occurring during the next weeks.

[5]`https://cwe.mitre.org/`

[6]We use the continuous color scale "ylorrd" of Plotly; `https://plotly.com/python/builtin-colorscales`

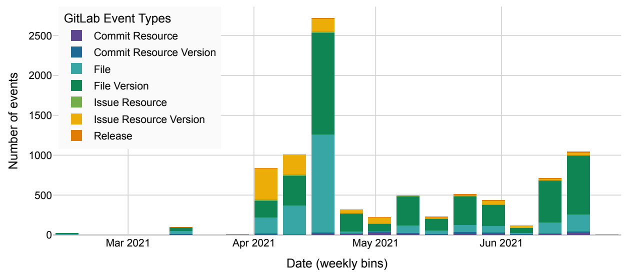


Figure 5: Distribution of events over time for all `lucaapp` repositories. In the diagram, the events are summarized by week.

4.3 Interactive Dashboard

To increase the usability and interaction possibilities of the described visualizations, the visualizations can be integrated into web-based user interfaces. There are two common approaches for this: *notebook interfaces* and *dashboards*. Notebook interfaces such as JUPYTER [8, 19] are very well suited for exploratory, flexible data analysis. Dashboards [6] provide a more rigid but interactive web-based interface that aims to provide an overview of information (similar to reports).

We developed a web-based interactive dashboard contains selected visualizations, which are provided with interactive control elements (Fig. 1).

Since we mainly want to have a fixed view with selected visualizations, we decided to develop a dashboard instead of a notebook interface. The reason for this is that the information shown is initially intended as a basis for discussions and decision making between developers and managers, and should therefore not change constantly in an exploratory manner. Especially for software systems with high social relevance (such as the *luca App* used here as an example), the dashboard should also be accessible to the interested public. The visualizations integrated into the dashboard were selected after discussions with software developers from the field of secure software development. It came out that first of all the temporal course of the development activities and the warnings are interesting (Fig. 1; left column), that it should be possible to select a time range here and that then the type of warnings for this time range should be explored further (Fig. 1; right column).

We create dashboards based on the Python framework DASH[7]. Visualizations created with the Python library PLOTLY can be integrated into interactive web-based applications; in particular, it is possible to add interaction elements such as menus, radio buttons, or input fields (Fig. 1). The individual charts benefit from Plotly's fundamental features, such as zoom, responsiveness, compare data on hover, selection and crossfiltering, and custom controls.

The "Entities Timeline" section refers to all projects, since it concerns here the development activities over the time; here one can select the resampling frequency, to represent for example daily, weekly, or quarterly activity progressions. The "Static Analysis Warnings" section shows the time history of the warnings, where you can select the project of interest (i.e., repository). The time axes of the entities and the warnings are linked; if you select a time range, the same range is also displayed in the other chart. The "CWE Breakdown" section contains the Icicle chart with an additional selection option for the 'root' (i.e., the first hierarchical level) of the displayed data. Currently, predefined CYPHER queries are stored here, which for example provide all commits of a project or only the commits to files where multiple developers have contributed—the possibility to add queries without changing the Python code is part of our future work.

[7]`https://dash.plotly.com/`

Currently, the layout and integrated visualizations are defined in Python code; likewise, the connected databases are fixed.

4.4 Use Case

The current design of our dashboard has two main distinct audiences: the software development team, including software engineering managers, and people from the public who have an interest in critically following and assessing the development. Both audiences have an understanding of how software development processes fundamentally work and are willing to dig into details of the development process. Both target groups also have an interest in ensuring that the software is secure and prohibits any attack vectors for misuse of the processed data.

An intended use case is to find out which potential security weaknesses are in the source code of the software and at which times and through which steps in the development process the number and severity of the weaknesses has changed.

The approach is that users of the dashboard first look at the "Entities Timeline" to see how the steps of software development have progressed over time by selecting the event types that interest them (e.g., releases or extensive discussions in the Issues). You can limit the timeline to specific time periods, and the "Static Analysis Warnings" will adjust accordingly to show the same time period. For the selected time period, the "CWE Breakdown" then displays the security weaknesses, which can still be explored visually down to the level of individual files. To further narrow down the displayed security weaknesses, they can be further narrowed down via CYPHER queries (Sect. 4.3).

5 RELATED WORK

The tool *Cesar* [1] improved the usability of the SAST tool FINDBUGS. The authors evaluated the provided user interface by conducting an user study and improved the visualization by categorizing the vulnerabilities. They used an interactive treemap showing the distribution of selected vulnerability categories. The *Nessus Vulnerability Visualization dashboard* [9] processed data of network vulnerabilities in particular. Their central element is also a treemap presenting the distribution of network scans.

The focus of the *Analizo visualization toolkit* [29] is on extensibility and multi-language support. The tool calculates various code metrics, dependency graphs and a software evolution matrix. The authors worked on the transparency of information but did not focus security-related aspects. Similar to that, *Source Meter* [5] is another dashboard for code metrics. The dashboard Seconda [21] divides global and local code metrics in their analysis and illustrations. The visual analysis of code security by Goodall et al. [7] presents the results of different SAST tools in form of a treemap.

6 CONCLUSIONS AND FUTURE WORK

We have described how we are developing a first variant of an interactive dashboard based on our work on provenance-driven security analysis. The dashboard contains visualizations of software development provenance and static code analysis results. The dashboard's interaction capabilities allow the visualizations to be linked, which helps with interactive visual analytics.

Future work on the dashboard and visualizations is mainly to improve the visual design and style by developing a consistent visual concept and evaluating it through user studies. In addition, we are planning technical enhancements such as a search function, more cross-filtering capabilities, and higher configurability.

ACKNOWLEDGMENTS

We thank DLR's student research assistants Claas de Boer (TU Dresden) for his tool `prov2neo` and Christopher-Tobias Knaust (Friedrich Schiller University Jena) for his work on SAST analysis.

References

[1] H. Assal, S. Chiasson, and R. Biddle. Cesar: Visual representation of source code vulnerabilities. In *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–8, 2016. doi: 10.1109/VIZSEC.2016.7739576

[2] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. doi: 10.1109/MS.2008.130

[3] C. de Boer and A. Schreiber. DLR-SC/gitlab2prov: GitLab2PROV 0.5, June 2021. doi: 10.5281/zenodo.5009043

[4] C. de Boer and A. Schreiber. DLR-SC/prov2neo: prov2neo 1.0, June 2021. doi: 10.5281/zenodo.5013869

[5] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, and T. Bakota. Source meter sonar qube plug-in. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 77–82, 2014. doi: 10.1109/SCAM.2014.31

[6] S. Few. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, Inc., 2006.

[7] J. R. Goodall, H. Radwan, and L. Halseth. Visual analysis of code security. In *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*, VizSec '10, pp. 46—-51. Association for Computing Machinery, New York, NY, USA, 2010. doi: 10.1145/1850795.1850800

[8] B. E. Granger and F. Perez. Jupyter: Thinking and storytelling with code and data. *Computing in Science & Engineering*, 23(02):7–14, mar 2021. doi: 10.1109/MCSE.2021.3059263

[9] L. Harrison, R. Spahn, M. Iannacone, E. Downing, and J. R. Goodall. Nv: Nessus vulnerability visualization for the web. In *Proceedings of the Ninth International Symposium on Visualization for Cyber Security*, VizSec '12, pp. 25—32. Association for Computing Machinery, New York, NY, USA, 2012. doi: 10.1145/2379690.2379694

[10] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In D. Notkin, B. H. C. Cheng, and K. Pohl, eds., *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pp. 672–681. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606613

[11] J. B. Kruskal and J. M. Landwehr. Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983. doi: 10.1080/00031305.1983.10482733

[12] M. J. McGuffin and J.-M. Robert. Quantifying the space-efficiency of 2d graphical representations of trees. *Information Visualization*, 9(2):115–140, 2010. doi: 10.1057/ivs.2009.4

[13] T. McPhillips, S. Bowers, K. Belhajjame, and B. Ludäscher. Retrospective provenance without a runtime provenance recorder. In *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*, TaPP'15. USENIX Association, USA, 2015.

[14] L. Moreau, P. Groth, S. Miles, J. Vazquez-Salceda, J. Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, and L. Varga. The provenance of electronic data. *Communications of the ACM*, 51(4):52–58, 2008. doi: 10.1145/1330311.1330323

[15] L. Moreau and P. T. Groth. *Provenance: An Introduction to PROV*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2013. doi: 10.2200/S00528ED1V01Y201308WBE007

[16] L. Moreau, P. Missier, K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker, S. Miles, J. Myers, S. Sahoo, and C. Tilmes. PROV-DM: The PROV data model, 2013.

[17] S. Munzert, M. Papoutsi, and H. Nowak. Nutzung von digitalen tools zur unterstützung von covid-19-kontaktverfolgung: Wie populär sind corona-warn-app und luca-app in der dritten pandemiewelle? Technical report, respondi, 2021. (german).

[18] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings. 27th International Conference on Software Engineering (ICSE 2005)*, pp. 580–586. ACM, 2005.

[19] J. P. Ono, J. Freire, and C. T. Silva. Interactive data visualization in jupyter notebooks. *Computing in Science & Engineering*, 23(02):99–106, mar 2021. doi: 10.1109/MCSE.2021.3052619

[20] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. S. Cruzes. Myths and facts about static application security testing tools: An action research at telenor digital. In *Proceedings of 19th International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2018)*, vol. 314 of *Lecture Notes in Business Information Processing*, pp. 86–103. Springer, Porto, Portugal, 2018. doi: 10.1007/978-3-319-91602-6_6

[21] J. Pérez, R. Deshayes, M. Goeminne, and T. Mens. Seconda: Software ecosystem analysis dashboard. In *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 527–530, 2012. doi: 10.1109/CSMR.2012.69

[22] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill. Identifying and documenting false positive patterns generated by static code analysis tools. In *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, pp. 55–61, 2017. doi: 10.1109/SER-IP.2017..20

[23] A. Schreiber. Provenance of luca app development., June 2021. doi: 10.5281/zenodo.5034813

[24] A. Schreiber, C. de Boer, and L. von Kurnatowski. GitLab2PROV—provenance of software projects hosted on GitLab. In *13th International Workshop on Theory and Practice of Provenance (TaPP 2021)*. USENIX Association, July 2021.

[25] 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*, vol. 12839 of *Lecture Notes in Computer Science*. Springer International Publishing, 2021. (in press). doi: 10.1007/978-3-030-80960-7_6

[26] T. Sonnekalb. Sast database of repository luca app android, June 2021. doi: 10.5281/zenodo.5036046

[27] T. Sonnekalb, T. S. Heinze, L. von Kurnatowski, A. Schreiber, J. M. Gonzalez-Barahona, and H. Packer. Towards automated, provenance-driven security audit for git-based repositories: Applied to Germany's Corona-Warn-App. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment (SEAD '20)*. ACM, New York, NY, USA, 2020. doi: 10.1145/3416507.3423190

[28] T. Stadler, W. Lueks, K. Kohls, and C. Troncoso. Preliminary analysis of potential harms in the luca tracing system. Mar. 2021.

[29] A. Terceiro, J. Costa, J. Miranda, P. Meirelles, L. R. Rios, L. Almeida, C. Chavez, and F. Kon. Analizo: an extensible multi-language source code analysis and visualization toolkit. In *Brazilian conference on software: theory and practice (Tools Session)*, 2010.

[30] L. Woodburn, Y. Yang, and K. Marriott. Interactive visualisation of hierarchical quantitative data: An evaluation. Aug. 2019. doi: 10.1109/VISUAL.2019.8933545

[31] B. Zheng and F. Sadlo. On the visualization of hierarchical multivariate data. In *2021 IEEE 14th Pacific Visualization Symposium (PacificVis)*, pp. 136–145, 2021. doi: 10.1109/PacificVis52677.2021.00026