VUSC: An Extensible Research Platform for Java-Based Static Analysis

Marc Miltenberger
Fraunhofer SIT — ATHENE
Darmstadt, Germany
marc.miltenberger@sit.fraunhofer.de

Steven Arzt
Fraunhofer SIT — ATHENE
Darmstadt, Germany
steven.arzt@sit.fraunhofer.de

***Abstract*—Detecting security vulnerabilities in backend Web applications as well as mobile apps is extremely important. Static analysis for vulnerability analysis has subsequently developed as an important field of research. Researchers need extensible frameworks to avoid starting from scratch with every new research project. Compared to commercially available scanners, open-source frameworks often only provide basic functionality. This limits the ability of researchers to evaluate novel algorithms. Lacking access to full code scanners, new building blocks are often tested in isolation. In this paper, we present VUSC, a fast, precise and extensible vulnerability scanner for Android and Java bytecode. It features a plugin architecture for commonly used static analyses such as call graph, taint and value analyses, allowing researchers to build upon our work and using VUSC as a reference platform. We show that VUSC achieves a precision of around 90% on benchmarks. Video: https://youtu.be/QpXs9hv5zGc, Dataset: https://github.com/Fraunhofer-SIT/ASE2025-StaticAnalysisInfrastructure/**

I. Introduction

The security of Java-based applications is of utmost importance because they form the basis for many everyday services. Java-based web services are the foundation for many critical applications such as online banking and eGovernment. If implemented insecurely, attackers can steal sensitive personal data or inflict financial or reputational damage on the users.

At the same time, Android phones are used to process sensitive information such as banking or health data. The research community has worked on security analysis tools for Java and Android applications. However, due to the many challenges the various platforms pose to analysis tools, many research projects tackle individual building blocks or particular types of vulnerabilities. Popular research topics include callgraph construction [1], value analysis [2] or data flow analysis [3], in addition to modeling specific Android features such as inter-component communication (ICC) [4]. This focus of the research community on individual challenges leaves the task of building a full vulnerability scanner mostly to commercial companies. As a consequence, the community only has limited access to the full scanners. As we show in our evaluation, the precision and recall of the few existing open-source scanners is severely lacking, making them unsuitable alternatives.

While this focus of research activity has its advantages, it forces authors to evaluate their algorithms in isolation [5]. A new callgraph algorithm, for example, can be tested by manually inspecting the discovered edges. Alternatively, researchers can integrate it into an open-source data flow analysis and check the effect of their new approach on the data flow results. Still, both testing strategies are not an end-to-end assessment of the new callgraph algorithm in the context of a real-world security analysis tool. Such an evaluation cannot answer the question whether new vulnerabilities can be found. We argue that the community needs a modular full-scale vulnerability scanner in which individual building blocks can be replaced, while keeping all other components untouched. Such a platform would allow for testing new ideas and approaches in a comprehensive environment.

While frameworks such as Soot [6] or Wala [7] exist and provide many low-level building blocks such as frontends for Android's Dalvik bytecode, this is not sufficient for a vulnerability analysis. An inter-process communication (IPC) analysis for Android, for example, requires ICC links, which requires values extracted from Intents. Authors who need a full callgraph as a basis for their work would need to stitch together a callgraph algorithm, a value analysis algorithm and an IPC resolver. Only after spending substantial engineering effort, which is unrelated to their core contribution, they have a platform on which they can start their own work.

In this paper, we present VUSC, a full vulnerability scanner with a modular architecture based on the Eclipse OSGi framework. VUSC is highly configurable and allows researchers to plug in their approaches via extension points with the possibility to replace existing algorithms. VUSC provides all the other building blocks required for vulnerability scanning as well as a library of roughly 400 analyses. After integrating their own analysis or building block, researchers can trigger vulnerability scans on individual programs or large datasets via the VUSC scheduling and job management engine.

As we show on the OWASP, Juliet and Ghera benchmarks, VUSC offers precision comparable to the most advanced commercial scanners on the market, by far surpassing what can be expected from open-source scanners. We also show that new plugins can be implemented with a few lines of code while relying on powerful features such as callgraph, data flow or value analysis.

We make VUSC available to non-profit academic research institutions via our academic licensing program. Note that Export regulations and restrictions may apply and are beyond our control. VUSC has officially been declared a dual use

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00354

good by the German export control authority BAFA, which requires additional checks and controls for each intended delivery outside of Germany. Moreover, our closed source license model enables us to finance the extensive development work through commercial customers, while making a contribution to the research world.

II. Architecture

Figure 1 shows the workflow of VUSC. In the following section, we present all steps involved in an analysis.

A. Analysis Projects

VUSC is built around the concept of a *project*. The project stores all data associated with a job such as the original input file, but also intermediate results such as the generated intermediate representation (IR) code, the callgraph, or the detected data flows.

Since VUSC is highly modular, the concept of a project must account different target platforms. For an Android app, the project must store Android components and their ICC links. For a Java-based web application, the project must provide an abstraction for servlets and REST endpoints. The different project types are contributed by an extensible set of plugins. For example, the implementation of the Android project data structure is part of the *android* platform module. The project type for Spring Boot-based web applications is instead part of the *jvm.web.spring* platform module. Based on this modular design, VUSC's project data structure allows for platform-specific extensions.

B. Project Import

VUSC takes a binary as input without any user-defined configuration. With an extensible set of *input detectors*, VUSC determines the type of the input file, e.g., an Android app. VUSC does not rely on file extensions, but matches the file contents against platform-specific patterns. As shown in Figure 1, the input detector then instantiates a project of the respective type as well as the analysis engine that can handle this project type. An engine translates the code of the target program into an IR and parses all auxiliary files such as the Android manifest or resource files.

The concept of an analysis engine is hierarchical. The Soot-based analysis engine is responsible for handling all Java and Android inputs and converts the bytecode into the Jimple IR. Specialized analysis engines augment platform-specific settings for Soot and FlowDroid. The Android engine extracts specific resource files such as the Android Manifest and the *strings.xml* files.

C. Analysis Scheduling

The imported VUSC project is registered with the job scheduler which keeps track of the computer's CPU and memory load. With the default configuration, it schedules an extensive list of about 400 vulnerability analyses. Each vulnerability analysis is a plugin into the VUSC framework. It can access the entire project including all IR code to perform arbitrary static analysis tasks. Furthermore, it can add vulnerability markers to the project to report its findings. These markers are the results that will be displayed to the user.

VUSC projects are conceptually similar to "blackboard" approaches that existing work uses for synchronizing analyses and exchanging data between them [8]. In these approaches, different analyses share a common data storage where analysis can publish (intermediate) results and can subscribe to other analyses' contributions. The data structure itself is agnostic to the individual data items. Android-specific information, for example, is only understood by Android-specific analyses.

An analysis defines the project type(s) for which it is applicable. Analyses can bind to all levels of the project hierarchy. They may, for example, check for misuses of the Java crypto API that this used in Android and Java alike. Other analyses only bind to the Android projects and, e.g., check whether the app exports critial components in its manifest.

Each analysis may depend on the results of *base analyses* such as the callgraph analysis. The scheduler ensures that dependencies are resolved and all analyses are scheduled in the correct order. A base analysis is only performed if there is at least one vulnerability analysis that requires it. Its result is shared across all vulnerability analyses. Note that base analyses are normal plugins and thus can be exchanged.

The data flow analysis is also a base analysis that queries each other analysis for sources and sinks. VUSC merges all individual source/sink lists and runs the static data-flow tracker FlowDroid [3] on the result. When the data flow analysis has completed, the flows are split such that each vulnerability analysis only receives the subset of flows relevant to its respective subset of sources and sinks. We find that this approach is more efficient than running FlowDroid separately for each vulnerability analysis, which is in line with existing work [9].

D. Platform Abstractions

VUSC provides domain-specific abstractions for each platform. For Android apps, VUSC identifies the components and their ICC links [10]. For web applications, VUSC identifies servlets and their corresponding REST endpoints. It then provides the resulting list of platform-specific entities to analyses as part of the project. Where possible, VUSC unifies concepts and provides the same abstractions. For example, REST endpoints are applicable to all web applications, regardless of a concrete web application framework. They always have a URL, a list of parameters, an HTTP method, etc. If a specific framework provides more information, the abstraction is augmented with this additional information that can be optionally used by vulnerability analyses.

Analyses can take advantage of platform abstractions. For example, the SQL injection analysis is implemented only once for all platforms and identifies data flows from *untrusted sources* to *database commands*. Instead of defining concrete sources and sinks, it uses placeholders that are called *source presets* and *sink presets*, respectively. These presets are filled

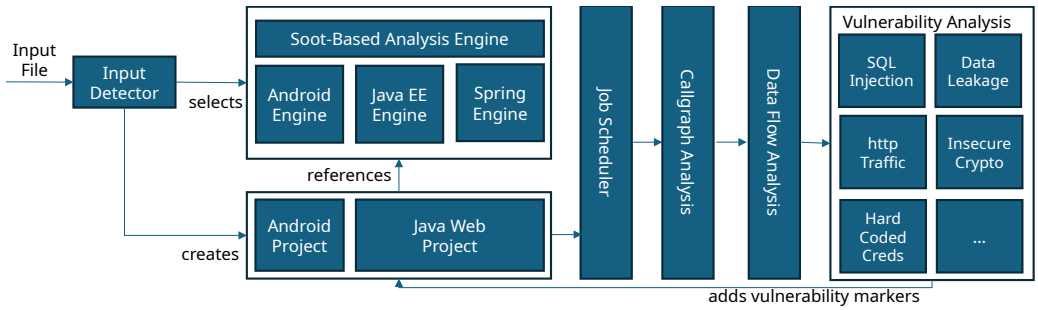

Fig. 1. Analysis Workflow: The input detector instantiates the analysis engine and the project before scheduling the analyses for execution.

by the platform module. Each platform has a notion of what an untrusted input is and what a database command is.

III. Features

In addition to the platform and analysis modules described in Section II-D, VUSC provides a *knowledge base*. Many analyses need to check programs against acceptable standards. For cryptography, NIST provides a list of algorithms deemed secure in the US. In Europe, the ENISA list is used. VUSC abstracts from such preferences using policies against which analyses can query cipher specifications in JCA (Java Cryptography Architecture) format.

Further, VUSC provides a value analysis [2]. Vulnerability analyses can query the values that a specific variable may take at a specific statement in the code. The value analysis extracts *pseudo constants*, i.e., values that need not be constant, but that are computed using values available in the code. This approach allows analyses to opaquely handle obfuscated values or values that are constructed by code, e.g., when building a URL from its parts (host, base URL, query, etc.).

We stress that all features can be replaced in VUSC. If a researcher aims to develop an analysis, they can keep the rest of the engine intact. If they wish to evaluate a new value analysis instead, they can replace the value analysis module and keep the existing vulnerability analyses for an end-to-end-evaluation. VUSC is a full code scanner that can serve as a testbed for new program analysis results by installing a new module and changing the configuration to select the new component as the default. Note that *additive* features are even simpler, e.g., adding a new platform. In that case, VUSC will automatically select the new module.

IV. Evaluation

We evaluate VUSC on three research questions:
RQ1: How effective and precise is VUSC on benchmarks?
RQ2: Does VUSC detect vulnerabilities in real-world apps?
RQ3: How time and memory efficient is VUSC?

A. Experiment Setup

For the evaluation on benchmarks, we ran VUSC on a Laptop with a i7-8850H CPU and 32 GiB of RAM. We configured VUSC to use a maximum of 16 GiB RAM.

For the evaluation on real-world applications, we ran VUSC on a server with 160 Xeon(R) Platinum 8380 CPU cores and 2 TiB of RAM. We configured VUSC to run with a maximum memory allocation of 500 GiB. Note that we did not observe VUSC allocating more than 84 GiB at runtime (see RQ3).

B. RQ1: Effectiveness and Precision on benchmarks

Benchmark suites consist of positive and negative test cases, which are usually grouped by CWE. A positive test case exhibits vulnerable behavior regarding that CWE, while a negative test case should not contain such a vulnerability.

We evaluate VUSC with three different benchmarks: Juliet, OWASP and Ghera. On all benchmarks, we compared VUSC, Insider, ShiftLeft, SonarCloud, FindSecBugs and Visual Code Grepper (VCG) in their respective newest version.

1) Juliet test cases: We used the Juliet [11] 1.3 test cases for Java with the corrections from literature [12]. This approach yielded 22,119 test cases across 26 different CWEs.

On average over all CWEs, VUSC shows a true positive rate (tpr) of over 90% and a false positive rate (fpr) of only 1.11%. VUSC has the highest tpr and lowest fpr of all tested tools. Of the other approaches, FindSecBugs features the highest tpr of 28.30%, but also reports many false positives with an fpr of 17.71%. ShiftLeft achieves a similar tpr of 27.34% and a slightly higher fpr of 23.51%. SonarCloud delivers a tpr of 10.76% with a fpr of 7.25%. Insider and VCG both only achieve a tpr of less than 3% with an fpr of around 3%.

2) OWASP test cases: OWASP Benchmark suite 1.2 features 2,740 test cases covering 11 CWEs. When excluding Trust Boundary, VUSC has a perfect tpr (100%) on all CWEs. On the complete suite, VUSC achieves a tpr of over 90% with a fpr under 8%. The second best approach FindSecBugs features a higher tpr of 97%. It supports Trust Boundary, but misses 31% of Weak Hash test cases and has a higher fpr of 53%. ShiftLeft achieves a tpr of 90.94% with a fpr of 47.54%. Sonarcloud delivers a tpr of 76.64% with a fpr of 26.27%. VCG/Insider have a tpr of less than 50% and a fpr of 23%.

3) Ghera Test Suite: Ghera [13] is a vulnerability benchmark suite for Android applications. We corrected several errors we noticed in the ground truth. The data package shows the corresponding test cases. On this test suite, VUSC achieves a tpr of over 94% with a fpr of under 2%. The second

highest tpr was achieved by Sonar with a tpr of 35.5% with a fpr of 7.69%. FindSecBugs achieves 24.04% tpr/5.77% fpr.

C. RQ2: Vulnerability Results on Real-World Apps

To measure the precision of VUSC on real-world apps, we have randomly selected 900 apps from the Google Play Store via the AndoZoo 2024 dataset [14]. VUSC analyzed 873 apps successfully. We were not able to process 27 apps using Soot. We cannot compare VUSC against the other approaches from RQ1, since these require source code. From these 873 apps, we randomly sampled 10 apps, on which VUSC reported 684 findings in total. Three different security experts manually checked these findings. The experts used the CodeInspect bytecode analysis tool that provides an interactive debugger in addition to typical IDE features for code navigation.

The experts could rate findings as false positive or true positive. In the case of disagreement, the experts discussed the matter. If no agreement could be reached, majority voting was used for the final verdict, which happened only once. We provide all verdicts in our supplemental data.

For these ten apps, VUSC shows a precision of 92.70 %. Note that we do not have any information about true positives or negatives for real-world applications, as we do not have any ground truth data. Therefore, we cannot calculate tpr/fpr.

D. RQ3: Time and Memory Efficiency

We measure VUSC's time and memory efficiency on the set of 873 apps from RQ2. While the fastest analysis took 47.6 seconds, the average app took 980.28 seconds. The slowest app took just under one hour. Two thirds of the applications were analyzed in under 20 minutes each. Recall that VUSC checks for 253 different vulnerabilities in Android apps. During the analysis, we measured the memory usage of VUSC every 30 seconds. The min/average/max usage per app is 274 MiB/10.2 GiB/86.2 GiB. 80% of the applications were analyzed using less than 16 GiB of memory.

E. Discussion

The efficiency and effectiveness of VUSC mainly stems from its building blocks. ValDroid [2] is used for value analysis and FlowDroid [3] for data flow analysis. The inter-component analysis for Android is based on ICCTA [10] with ValDroid replacing IC3 [4]. Additionally, VUSC contains technical components such as parsers and job schedulers as well as around 400 vulnerability analyses.

To ensure the stability of VUSC, we have run the tool on thousands of applications and checked for crashes or error reports. Since VUSC is a commercial product, we receive feedback from paying customers on private apps. If such an error report affects an open-source building block from the literature, we contribute our fixes back to the original project.

V. Conclusion

In this paper, we have shown VUSC, an extensible framework for finding vulnerabilities in Android apps and Java-based applications using advanced static code analysis. It features a call graph, data flow and value analysis. Its modular building blocks can be exchanged to provide research testbeds. VUSC can be extended to support more platform types.

VUSC outperforms other tools on the OWASP, Juliet and Ghera test suites. 80% of real-world applications were analyzed taking less than 30 minutes using less than 16 GiB of memory. We offer VUSC to non-profit research institutions under an academic license as a base for future projects.

Acknowledgment

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini, "Call graph construction for java libraries," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 474–486.

[2] M. Miltenberger and S. Arzt, "Precisely extracting complex variable values from android apps," *ACM Trans. Softw. Eng. Methodol.*, feb 2024. [Online]. Available: https://doi.org/10.1145/3649591

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[4] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 77–88.

[5] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 176–186.

[6] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, no. 35, 2011.

[7] I. Watson, "Watson libraries for analysis," *Main Page*. [Online]. Available: http://wala.sourceforge.net/wiki/index.php

[8] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, "Modular collaborative program analysis in opal," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, p. 184ff.

[9] S. Arzt, "Static data flow analysis for android applications: Statische datenflussanalyse für android-anwendungen," Ph.D. dissertation, Technische Universität Darmstadt, 2017.

[10] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[11] F. B. Jr. and P. Black, "The juliet 1.1 c/c++ and java test suite," no. 45, 2012-10-01 00:10:00 2012.

[12] M. Miltenberger, S. Arzt, P. Holzinger, and J. Näumann, "Benchmarking the benchmarks," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 387–400. [Online]. Available: https://doi.org/10.1145/3579856.3582830

[13] J. Mitra and V.-P. Ranganath, "Ghera: A repository of android app vulnerability benchmarks," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE. ACM, Nov. 2017.

[14] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Androzoo: Collecting millions of android apps for the research community," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 468–471.