SECUCHECK: Engineering configurable taint analysis for software developers

Goran Piskachev Ranjith Krishnamurthy Eric Bodden
Fraunhofer IEM Fraunhofer IEM Paderborn University and Fraunhofer IEM
goran.piskachev@iem.fraunhofer.de ranjith.krishnamurthy@iem.fraunhofer.de eric.bodden@uni-paderborn.de

Abstract—Due to its ability to detect many frequently occurring security vulnerabilities, taint analysis is one of the core static analyses used by many static application security testing (SAST) tools. Previous studies have identified issues that software developers face with SAST tools. This paper reports on our experience in building a configurable taint analysis tool, named SECUCHECK, that runs in multiple integrated development environments. SECUCHECK is built on top of multiple existing components and comes with a Java-internal domain-specific language fluentTQL for specifying taint-flows, designed for software developers. We evaluate the applicability of SECUCHECK in detecting eleven taint-style vulnerabilities in microbench programs and three real-world Java applications with known vulnerabilities. Empirically, we identify factors that impact the runtime of SECUCHECK.

Index Terms—static analysis, security, integrated development environment, taint analysis, domain-specific languages

I. INTRODUCTION

With the increased digitalization and processing of sensitive data, the early detection of security vulnerabilities during development becomes highly relevant for many companies. In that context, taint analysis has been successfully applied in detecting SQL injections, memory leaks, cross-site scripting, and other vulnerabilities [1]. It is one of the core techniques used in many static application security testing (SAST) tools (e.g., CheckMarx [2], LGTM [3]).

Recent usability studies on SAST tools [4]–[6] have identified software developers' requirements and issues with the existing tools. Based on these studies, we list the requirements for the future SAST tools that this paper addresses:

R1 Workflow integration: Software developers reported that the tools should be well integrated within their daily used development environments (IDEs) to develop new applications. They should appear as part of the IDEs and only provide the necessary findings reported from the analysis using the standard IDE features, such as error view, editor markers, and syntax highlighting.

R2 Configurable tools: One of the known weaknesses of static analysis, including taint analysis, is the reporting of false findings, which causes usability issues [6]. One approach to improve this is by configuring the rules of the analyses through domain-specific languages (DSLs). This allows the specification of custom rules for company-specific contexts. Even though many tools provide such DSLs, their stakeholders are static analysis experts. Software developers need developer-centric DSLs.

R3 Explainability: The messages of the findings shown to the users should be understandable. The tools should provide additional information about the findings when needed. Referring to R2, the DSL should also be understandable for software developers.

R4 Fast results: Taint analysis can run long on real-world applications measured in minutes and even hours, which is not practical in the IDEs. Hence, a taint analysis running in the IDE should provide means to analyze only parts of the code relevant to the user in the current context in terms of only a few minutes or seconds.

In this paper, we discuss our experience in developing a configurable taint analysis tool for Java, named SECUCHECK that addresses the previously stated requirements. The security rules for the analysis are written in fluentTQL, an existing Java internal DSL. SECUCHECK can run on two data-flow solvers, the first one based on Synchronized Pushdown Systems [7] and the second one based on Interprocedural, Finite, Distributive, Subset Problems [8]. We provide insights on our architectural decisions and highlight several technical details to help other practitioners in building similar tools. SECUCHECK is built as a MAGPIEBRIDGE server [9] and can run in multiple IDEs supporting all native features. Additionally, SECUCHECK uses the MAGPIEBRIDGE support of the HTTP protocol to display a graphical configuration page for selecting rules, entry points, and other options, enabling users to configure the tool to run on the relevant parts of the code and provide fast results.

Our evaluation shows the applicability of SECUCHECK on microbenchmark programs covering different security vulnerabilities and real-world applications with known security vulnerabilities. Moreover, based on our empirical data, we identify factors that impact the analysis runtime in practice. For example, for the set of applications we used, the entry points selection has a much smaller impact on the runtime than the security rules selection in each run of the analysis.

The list of contributions that this paper makes are:
• SECUCHECK: an open-source 1 taint analysis tool with configurable rules running in multiple IDEs,
• translation of fluentTQL into English sentences that improves the explainability aspect of SECUCHECK,

1https://github.com/secure-software-engineering/secucheck
II. REUSING EXISTING COMPONENTS

To avoid re-inventing the wheel, 

SECUCHECK builds on top of existing open-source components that are well-established in the software engineering community (gray components in Figure 1), of which in the following, we explain those important for the taint analysis.

A. Static analysis framework: Soot

SOOT is the core component for static analysis. It transforms the compiled Java bytecode to a simple 3-address intermediate representation on which data-flow analyses run. SOOT has simple built-in analyses, e.g., dead code elimination and constant propagation, on which more complex client analyses can be built on. SOOT provides two core data structures, i.e., a control-flow graph for intra-procedural analyses and a call graph for inter-procedural analyses.

B. Data-flow analysis solvers: BOOMERANG and FLOWDROID

SECUCHECK integrates two data-flow analysis solvers, BOOMERANG and FLOWDROID; both built on top of SOOT. BOOMERANG is a demand-driven, points-to analysis using two synchronized pushdown systems [7]. The client analyses can create queries for a given location (a variable and a statement) to compute a tree structure of paths reachable from that location. SECUCHECK uses this tree structure and processes it to detect taint-flows conforming to the fluentTQL specifications (Sub-section III-C) selected by the user. BOOMERANG is the default solver in SECUCHECK. Alternatively, the user may configure to run the analysis using the FLOWDROID solver based on the IFDS framework [10]. FLOWDROID is designed as a taint-tracking engine for Android apps that for a given list of sources and sinks reports the existing taint-flows between any source/sink pair. Further details on how SECUCHECK uses both solvers are provided in Sub-section III-B.

C. Multi-IDE support with MAGPIEBRIDGE

To support wide range of IDEs (R1), we built SECUCHECK as a MAGPIEBRIDGE server [9] which uses the LSP protocol2 to communicate with any LSP-aware IDE. MAGPIEBRIDGE runs the analysis triggered by the user through a configuration HTML page and returns the results in JSON format to the IDE. The LSP protocol supports many standard UI features, such as syntax highlighting, error markers, messages for the error view, and hover information.

III. SECUCHECK

A. Architecture

Figure 1 shows the internal components of SECUCHECK and their interaction with the external components. The components in orange are directly accessible to the users through provided interfaces. The SECUCHECK-core analysis runs the main analysis process. It uses SOOT to generate the Jimple format from the bytecode being analysed and calls the BOOMERANG or the FLOWDROID APIs to run one of the solvers. SECUCHECK-Magpie integrates the SECUCHECK-core into MAGPIEBRIDGE. An alternative way to run SECUCHECK is through the command line tool SECUCHECK-cmd. The fluentTQL-DSL is a DSL for specifying taint-flows queries for the analysis. fluentTQL-classloader uses the JCL-core to load the taint-flow specifications into the JVM. The maven-plugin-api provides APIs for running tools as Maven plugin. This is used by fluentTQL-maven-plugin to run a semantic check of the fluentTQL specifications. The fluentTQL2English transforms the fluentTQL specifications into English sentences to provide the user more detailed description of the queries (R3). The components SECUCHECK-Magpie and SECUCHECK-cmd use fluentTQL2English to display it in the error message (R1).

Figure 1: SECUCHECK architecture.

B. Taint analysis

a) Example: We use the example code in Figure 2c containing an SQL injection vulnerability to demonstrate the taint analysis in SECUCHECK. The code reads an untrusted data from the scanner (line 29), establishes a connection to an SQL database (lines 32-33), creates and executes a query statement (line 35), and returns the result. The code is vulnerable to SQL injection because an untrusted value from the scanner is appended to the SQL query without proper validation before executing it. SECUCHECK as a taint analysis will detect this taint-flow if the return value of the method nextLine (line 29) is modelled as a source and the parameter value of the method executeQuery (line 35) is modelled as a sink.

b) Querying BOOMERANG: SECUCHECK uses BOOMERANG’s demand-driven points-to analysis to calculate the data-flow information. It analyses each fluentTQL

---

2https://microsoft.github.io/language-server-protocol/
SECUCHECK-core extracts all sources from the fluentTQL specifications selected by the user. A new BOOMERANG forward query is created for each matched source in the analysed Jimple code and tainted variable specified in the fluentTQL. To handle methods from libraries for which the code is not available, SECUCHECK uses the API from the BOOMERANG’s DemandDrivenManager that takes a list of propagators, which are defined as a method signature and taint propagation rule. For example, the string concatenation in Java bytecode will be represented with the method java.lang.StringBuilder.append() (like in line 35 from Figure 2c), for which SECUCHECK is aware that this method taints the return value when the argument is tainted. SECUCHECK has a built-in list of such JDK methods, called general propagators. The users can use the fluentTQL DSL to specify new propagators and model the behavior of other regularly used libraries. In practice, this improves the runtime by not analysing the library calls multiple times. Moreover, when the library code is unavailable, it reduces the number of false negatives (findings that would be missed if the taint propagation through a given method is missing). SECUCHECK also supports the concept of required propagator, which is a method call that has to be on the taint-flow path between the source and a sink. This is useful for some vulnerabilities where the order of method calls in the program is relevant such as the incorrect use of cryptographic libraries. Finally, when each query is finished, BOOMERANG returns all taint-flow paths from the source variable. Then, SECUCHECK-core checks whether any path contains a sink statement conforming to the taint-flow specified in the fluentTQL specification for which the found taint-flow is reported as a potential vulnerability to the user.

c) Running FLOWDROID’s Infoflow: FLOWDROID consists of a general taint analysis component called Infoflow and a component that models the Android-specific behavior. SECUCHECK directly calls Infoflow by providing a list of method signatures for sources and sinks. The propagators are built-in and are read from a text file. For each required propagator, SECUCHECK divides the taint-flow into two, i.e., first, from source to the required propagator, and second, from the required propagator to the sink. For each of these taint flows, SECUCHECK calls Infoflow. Once the results are returned, they are reported.
by SECUCHECK. Currently, SECUCHECK has three technical limitations when using Infoflow. First, Infoflow only supports method calls as sources and sinks. Hence, when the source is also the analysis’ entry point such as the handler method of HTTP requests in Java EE / Spring applications, the taint is not created within the source definition, and taint-flows are not detected. Second, Infoflow does support sanitizers. And third, Infoflow taints always the return value of a source and the parameters of a sink. Other options are not configurable through the available API. Final limitation of FLOWDROID is that it does not provide the line numbers of the sources and sinks.

C. fluentTQL

fluentTQL is an existing DSL [11] that is integrated into SECUCHECK as a means to support developers in specifying new and project-specific taint-flows (R2). It is implemented as a Java-internal DSL using the builder pattern for its fluent syntax and the Java preprocessor for the annotations. Listing 1 shows an example, specification for detecting the SQL injection from Figure 2c. Any Java class that implements the interface FluentTQLUserInterface will be considered as a specification by SECUCHECK. Users can use the class Method to declare appropriate relevant methods from the codebase, such as sources, sinks, sanitizers, and propagators (lines 5-15). The constructor of MethodSelector takes a method signature as string to be matched by SECUCHECK-core. The tainted variables are modelled through annotations. For example, the annotation OutFlowReturnValue can be used to model that the return value of the method flowing out becomes tainted. It can be used for source, sanitizer, and propagator. Similarly, the annotation InFlowParam (parameterID=0) models the first parameter of the method flowing in as relevant for the analysis. It can be used for sink, sanitizer, and propagator. The query specifications are returned in the interface method getFluentTQLSpecification (line 17) in which TaintFlowQuery objects can be created and returned as a list. The query gets an ID as string (line 20), and builds the expected taint-flow via method chain: from for source, notThrough for sanitizer, to for sink, report for error message, at for location of the error message, and build for completing the query.

As internal DSL, fluentTQL provides the entire Java infrastructure to the user, making it easy to learn users familiar with Java. The specifications can be reused across multiple projects. The methods can be grouped into method sets for easier maintenance and better organization.

Finally, to ensure that the user provides valid specifications, SECUCHECK has a Maven plugin using the Maven-plugin-api to perform static semantic checks. These checks include the uniqueness of the vulnerability ID, correct use of the annotations, completeness of the method objects, etc.

D. UI Features

In the following, we discuss the three main user interface features that SECUCHECK provides through its MAGPIEBRIDGE server to the users of multiple IDEs clients.

Listing 1: fluentTQL specification for simple SQL Injection (the fully qualified names are omitted due to simplicity)

```java
public class SimpleSQLInjectionSpec implements FluentTQLUserInterface {
    @OutFlowReturnValue
    public Method source = new MethodSelector("String nextLine()") ;
    @InFlowParam (parameterID = 0)
    @OutFlowReturnValue
    public Method sanitizer = new MethodSelector("String sanitize(String)");
    @InFlowParam (parameterID = 0)
    public Method sink = new MethodSelector("ResultSet executeQuery(String)");

    public List<FluentTQLSpecification> getFluentTQLSpecification() {
        TaintFlowQuery myTF = new TaintFlowQueryBuilder("SQLi vulnerability")
            .from (source)
            .notThrough (sanitizer).
            .to (sink)
            .report("SQL Injection - CWE89")
            .at (LOCATION.SOURCEANDSINK).
            .build();

        List<FluentTQLSpecification> specs = new ArrayList<>();
        specs.add(myTF);
        return specs;
    }
}
```

a) Configuration page: For managing the analysis, SECUCHECK has two configuration pages (R2), created with the Bootstrap 3.3.5 framework. This is supported through the MAGPIEBRIDGE server using the HTTP protocol. When the project in the IDE opens, SECUCHECK will create the first configuration page as shown in Figure 2a. The project name is shown on the top (1). Two tabs (2) and (4) are available on this page. (1) is for setting the path of external jar with fluentTQL specification. (2) is for selecting the IDE, (4) is for customizing the view of the queries on the next page. When the first page is submitted, the second one will automatically appear (Figure 2b). This page shows six buttons for submitting a configuration (5), triggering the analysis (6), cancelling already started analysis (7), clearing the results from the previous analysis in the IDE (8), selecting all elements from the list (9), and deselecting all elements from the list (10). The page has three lists of elements, one in each tab. (11) shows a list of all taint-flow queries (R2) that are available. (12) lists all classes from the codebase that can be selected as entry points for the call graph construction (R2). With (13) the user can select the solver, BOOMERANG or FLOWDROID. These selections allow the user to run the analysis for specific context and get fast results (R4).

3https://bootstrapdocs.com/3.3.5/docs/getting-started/
b) **IDE standard UI features:** Figure 2c shows a screenshot of the Eclipse as example IDE that indicates the standard editor features (R1) that SECUCHECK uses to display the results of the analysis. The results are listed in the standard error view (3). Error markers are shown on the side of the editor (2). A hover over the error item or the marker shows more detailed description of the found taint-flow (14). This message is an English translation of the fluentTQL specification for the found taint-flow. We explain this translation in the following.

c) **Explainability of the findings:** To improve the explainability of the result messages in SECUCHECK, we implemented fluentTQL2Eng, a translator to English sentences. fluentTQL2Eng parses a taint-flow query object and visits each field. It maps each field to a predefined phrase recursively. Within the sentence, it adds information about the found taint-flow by the analysis, such as the source and sink locations. The final sentence is provided to SECUCHECK-Magpie which maps the message to the corresponding findings. An example is shown in the yellow message box (14) in Figure 2c.

E. **Command-line support**

While in some scenarios, SECUCHECK can deliver results in seconds for real-world applications, as our evaluation shows in the following section, it may run for minutes or few hours in other scenarios. In such cases, batch analysis in the build pipeline is preferred. Therefore, SECUCHECK provides a command-line tool. The configuration options provided via the configuration pages can be specified in YAML format given as an input along with the bytecode being analysed and the fluentTQL specifications.

IV. **Evaluation**

We answer the following research questions:

**RQ1** Can SECUCHECK analyse real-world Java applications?

**RQ2** What factors impact the runtime of SECUCHECK?

The results below are based on our experiments with SECUCHECK running the BOOMERANG solver. We omitted the FLOWDROID solver due to its limitations to handle the types of applications we selected.

a) **RQ1:** To evaluate the applicability of SECUCHECK, we run the analysis on four different projects:

- Catalog Microbenchmark\(^4\) - a set of small Java programs with 11 types of vulnerabilities and 27 taint-flows.
- Spring ToDo App\(^5\) - a small demo project for managing tasks with 9 types of vulnerabilities and 14 taint-flows.
- PetClinic\(^6\) - an insecure version of the official Spring PetClinic application with known vulnerabilities of which we documented four taint-flows of type hibernate injection.
- OWASP WebGoat\(^7\) - an insecure Spring application for learning security vulnerabilities, of which we documented 16 taint-flows of type SQL injection.

\(^4\)https://fluenttql.github.io/catalog/
\(^5\)https://github.com/secure-software-engineering/secucheck
\(^6\)https://github.com/contrast-community/spring-petclinic
\(^7\)https://owasp.org/www-project-webgoat/

We specified fluentTQL queries for the selected projects. SECUCHECK found all expected taint-flows. Table I provides an overview of the four projects. We used Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 16 GB RAM with Win-10 OS.

b) **RQ2:** To identify relevant factors for the analysis runtime, we focused on three aspects: (1) impact of the number of entry points selected for the call graph, (2) impact of the number of taint-flow queries, and (3) impact of the query complexity. For all three aspects, we took measurements of the analysis runtime over 10 runs of SECUCHECK with the BOOMERANG solver on the four selected projects.

Figure 3 shows the total analysis run for each project, when the number of entry points increases and all available taint-flow queries (specifications) are used. The graphs of all projects follow the same trend of slow linear growth. Contrary, when the number of selected taint-flow queries increases the growth is much significant. This case is shown in Figure 4 when only single entry point is selected in each run Figure 5 when all entry points are selected in each run. In both figures, we see the same trend, which compared to Figure 3 has much higher growth. Note: fluentTQL can express all queries of the same CWE with only few queries, e.g., the 4 taint-flows in PetClinic can be detected by single query as all taint-flows are of type SQL injection. For the purpose of having higher number of queries in **RQ2**, we expressed each taint-flow in a separate query.

With respect to query complexity, we selected five types of fluentTQL queries and ran them on the Catalog project:

1. query with single source and sink (simple case where SECUCHECK calls only one BOOMERANG query)
2. query with single source, sanitizer, and sink (simple case where SECUCHECK calls only one BOOMERANG query and passes the sanitizer to be processed by the DemandDrivenManager)
3. query with single source, 10 required propagators, and a single sink
4. query with set of 28 sources, single sanitizer, and 12 sinks (or-semantics, where for each combination of source and sink new BOOMERANG query is created)
5. query with three parallel taint-flows (use of and-operator to run multiple queries and report single results when each query returns a finding).

Table I: Overview of the evaluated projects. Flows is number of expected taint-flows (vulnerability instances), Queries is number of fluentTQL queries, CWE is number of common weakness enumerations (vulnerability types), Runtime is average over ten runs.

<table>
<thead>
<tr>
<th>Project</th>
<th>#Classes</th>
<th>#Flows</th>
<th>#Queries</th>
<th>#CWE</th>
<th>Runtime(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalog</td>
<td>36</td>
<td>27</td>
<td>19</td>
<td>11</td>
<td>52.79</td>
</tr>
<tr>
<td>ToDo App</td>
<td>26</td>
<td>14</td>
<td>14</td>
<td>8</td>
<td>34.45</td>
</tr>
<tr>
<td>PetClinic</td>
<td>42</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>10.94</td>
</tr>
<tr>
<td>WebGoat</td>
<td>35</td>
<td>16</td>
<td>16</td>
<td>1</td>
<td>30.29</td>
</tr>
</tbody>
</table>

The average runtime in seconds over 10 runs for each complexity type are, 7.16 for type 1, 7.07 for type 2, 7.11 for type 3, 10.08 for type 4, and 16.65 for type 5. The query
types 1, 2, and 3 show similar runtime, meaning that the use of sanitizers or required propagators does not have significant impact. Whereas, based on types 4 and 5 the use of method sets and the and-operator increase the runtime of the queries.

Figure 3: Analysis runtime by increasing the number of entry points

Figure 4: Analysis runtime by increasing the number of taint-flow specifications (single entry point)

V. CONCLUSION AND FUTURE WORK

In this paper, we presented SECUCHECK, a taint analysis tool for software developers to run in multiple IDEs for early detection of security vulnerabilities. SECUCHECK comes with fluentTQL, an internal-Java DSL with fluent syntax for specifying taint-flow queries to enable the users to customize the queries to the codebase or add queries for new security vulnerabilities. We discussed the architecture of SECUCHECK and the underlying components with their interfaces. To get fast results, SECUCHECK provides an HTML page in which the user can limit the scope of the analysis by selecting only relevant fluentTQL queries and entry points of the analysis. In the evaluation, we show the applicability of SECUCHECK on real-world Java applications with known vulnerabilities. Finally, based on empirical data, we found out that the number of selected taint-flow queries has much higher impact on the runtime than the number of call graph entry points. Moreover, the complex queries significantly increases the runtime.

SECUCHECK is available as an open-source tool. We plan on extending and improving the tooling with new features that improve the usability and explainability, such as recommendations for fixes, followed by a thorough evaluation on R1-3 via user study. From the analysis perspective, we plan on providing improved algorithms for call graph construction.

REFERENCES