Empirical Software Engineering (2022) 27: 104
https://doi.org/10.1007/s10664-022-10165-y

Fluently specifying taint-flow queries with *fluent*TQL

Goran Piskachev[1] · Johannes Späth[2] · Ingo Budde[1] · Eric Bodden[1,3]

Accepted: 5 April 2022 / Published online: 30 May 2022

Abstract

Previous work has shown that taint analyses are only useful if correctly customized to the context in which they are used. Existing domain-specific languages (DSLs) allow such customization through the definition of deny-listing data-flow rules that describe potentially vulnerable or malicious taint-flows. These languages, however, are designed primarily for security experts who are expected to be knowledgeable in taint analysis. Software developers, however, consider these languages to be complex. This paper thus presents *fluent*TQL, a query specification language particularly for taint-flows. *fluent*TQL is internal Java DSL and uses a fluent-interface design. *fluent*TQL queries can express various taint-style vulnerability types, e.g. injections, cross-site scripting or path traversal. This paper describes *fluent*TQL's abstract and concrete syntax and defines its runtime semantics. The semantics are independent of any underlying analysis and allows evaluation of *fluent*TQL queries by a variety of taint analyses. Instantiations of *fluent*TQL, on top of two taint analysis solvers, Boomerang and FlowDroid, show and validate *fluent*TQL expressiveness. Based on existing examples from the literature, we have used *fluent*TQL to implement queries for 11 popular security vulnerability types in Java. Using our SQL injection specification, the Boomerang-based taint analysis found all 17 known taint-flows in the OWASP WebGoat application, whereas with FlowDroid 13 taint-flows were found. Similarly, in a vulnerable version of the Java Spring PetClinic application, the Boomerang-based taint analysis found all seven expected taint-flows. In seven real-world Android apps with 25 expected malicious taint-flows, 18 taint-flows were detected. In a user study with 26 software developers, *fluent*TQL reached a high usability score. In comparison to CodeQL, the state-of-the-art DSL by Semmle/GitHub, participants found *fluent*TQL more usable and with it they were able to specify taint analysis queries in shorter time.

Keywords Taint analysis · Program analysis · Domain-specific language · User study · Usability

Communicated by: Bara Buhnova

✉ Goran Piskachev
goran.piskachev@iem.fraunhofer.de

[1] Fraunhofer IEM, Paderborn, Germany

[2] CodeShield GmbH, Paderborn, Germany

[3] Department of Computer Science, Paderborn University, Paderborn, Germany

1 Introduction

Over the past decade, static and dynamic taint analyses have gained significant traction both in industry and academia (Späth et al. 2019; Bodden 2018; Grech et al. 2018; Arzt et al. 2014). This is due to the fact that—in principle—most types of security vulnerabilities on the code level, e.g. 17 of the 25 vulnerabilities types of SANS-25 (Mitre 2020a), can be detected via taint analysis (Piskachev et al. 2019). Similarly, the OWASP top 10 list (OWASP 2020b) comprises 6 taint-style vulnerability types.

Taint analysis tracks sensitive data from *sources*, which are typically method calls to application programming interfaces (APIs), to program statements performing security-relevant actions, known as *sinks*. To soundly and precisely detect security vulnerabilities in a given software development project, any taint analysis, whether static or dynamic, requires a configuration. Particularly, the sources and sinks must be configured regarding the libraries and frameworks the project uses (Arzt et al. 2013). Additionally, due to a lack of scalability, static analyses frequently are unable to analyze *all* the software's code and must instead be configured to cut corners.

Some existing static analysis tools from academia (Krüger et al. 2019; Martin et al. 2005; Johnson et al. 2015) as well as from industry (Checkmarx 2020; Microfocus 2020; Grammatech 2020; Github 2020) provide a DSL to configure their analyses. However, all of the existing DSLs are designed to be used by static analysis experts and not by software developers—despite the fact that developers are usually the ones who best know how the project under analysis is structured. This was also confirmed in a recent research project with several industry partners (SecuCheck 2021), in which the authors conducted interviews with software developers that have used various commercial and non-commercial static analysis tools. Eight of the nine interviewees find the configuration options of the taint analysis tools to be too complex. In another recent study among developers, the authors discovered that for 47.1% of the participants, there is a dedicated team to configure the used static analysis tools, 36.8% configure their analysis tools themselves, and 16.2% run on default settings (Nguyen Quang Do et al. 2020). Moreover, the existing DSLs require expertise in static code analysis which many developers do not have. We are not aware of any excising study that evaluates the usability of the DSLs used for configuring the tools.

While much effort has been spent on automatically proposing relevant sources, sanitizers, and sinks (Piskachev et al. 2019; Arzt et al. 2013; Sas et al. 2018) or inference of taint-flows (source-sanitizer-sink paths) (Livshits et al. 2009; Chibotaru et al. 2019; Song et al. 2019), in practice taint analyses still require substantial manual specification effort.

To address this shortcoming, this paper presents a new domain-specific language called *fluent*TQL. *fluent*TQL is designed for software developers—not static or dynamic analysis experts—and allows the specification of taint-flow queries. Compared to existing DSLs, the abstraction level of *fluent*TQL is specific to taint analysis and contains only concepts that allow software developers to easily create or modify taint-flow queries. In result, *fluent*TQL queries can be evaluated by virtually any existing taint analysis. This sets the language apart from previous more generic code-query language such as CodeQL, the state-of-the-art DSL used within the commercial tool LGTM by Semmle/GitHub. At the same time, *fluent*TQL is sufficiently expressive, though, to support the specification of multiple taint-flows which allow the detection of complex security vulnerabilities.

Since Java is still amongst the most widely used languages, we designed *fluent*TQL as an internal Java DSL with a fluent-API design.[1] This paper presents the syntax and semantics

[1] Fluent Interfaces: https://www.martinfowler.com/bliki/FluentInterface.html

of *fluent*TQL, which is independent of any concrete (static or dynamic) taint analysis. Our example implementation instantiates *fluent*TQL with two static taint analyses, one based on Boomerang (Späth et al. 2019) and one based on FlowDroid (Arzt et al. 2014). We explain how these implementations statically approximate the *fluent*TQL semantics. The implementation is built on top of MagpieBridge (Luo et al. 2019) and the Language Server Protocol (Microsoft 2020). In result, it can be used in a multitude of editors and integrated development environments (IDEs), including Vim, Eclipse, VSCode, IntelliJ, SublimeText, Emacs, Thea and Gitpod.

We evaluate the usability of *fluent*TQL through a user study with 26 participants (professional software developers, students, and researchers). We compare *fluent*TQL to the more generic CODEQL. The results show that software developers perceive *fluent*TQL as easier to use. *fluent*TQL has an excellent System Usability Score (SUS) (Brooke 2013) of 80,77 (out of 100), whereas (for taint analysis) CODEQL has a score of only 38,56.[2] The Net Promoter Score (NPS) (Reichheld 2003) shows that—for the task of specifying taint-flow queries—participants would recommend to others *fluent*TQL over CODEQL. Moreover, we evaluate the applicability of *fluent*TQL by providing specifications of 11 popular vulnerabilities with catalog of small programs. Additionally, we select two vulnerable Java applications (OWASP WebGoat application[3] and PetClinic[4]) and seven real-world Android applications from TaintBench (Luo et al. 2021) known with malicious behavior. For all applications, we specified corresponding *fluent*TQL queries and were able to detect most of the expected taint-flows.

To summarize, this paper makes the following contributions:

- *fluent*TQL, a new DSL for specifying taint-flow queries, designed to be well usable for software developers.
- A formal definition of the syntax and semantics of *fluent*TQL, the latter independent of any concrete taint-analysis tool.
- An implementation and an empirical evaluation of the usability of *fluent*TQL in comparison to a state-of-the-art DSL for static code analysis.

Our artifact includes the *fluent*TQL tooling, the catalog of queries for popular taint-style security vulnerabilities and the dataset of our user study. It is available anonymously online at https://fluenttql.github.io/

We next explain relevant concepts on taint analysis and elicit the requirements for a developer-centric DSL. In Section 3 we present *fluent*TQL with its syntax and semantics, and explain how our static instantiations statically approximate this semantics. In Section 4 we discuss the user study. We discuss related work in Section 5 and, finally, we conclude in Section 6.

2 Requirements for a Taint Analysis DSL

We next explain the concept of taint analysis with an example and define requirements for a developer-centric DSL for taint analysis.

[2] Interpreting SUS: 0–50 is bad, 51–67 is poor, 68 is an average usability, 69–80,3 is good, > 80,4 is excellent, and 100 is imaginary perfect.

[3] https://github.com/WebGoat/WebGoat

[4] https://github.com/contrast-community/spring-petclinic

Listing 1 shows an excerpt of Java code of an HTTP handler. The method *doGet* is called upon a GET-request from a web browser when a user changes the password by providing the username, the old password, and the new password. The method calls a helper method *changePassword* shown in Listing 2 which verifies the user and changes the database. The code in *doGet* contains a potential cross-site scripting vulnerability (XSS) (Mitre 2020d). The username value from the request in the variable *uName* is added to the created HTML page for the response object to inform the user if the password was changed successfully (line 5). There is no sanitization check if the value contains any malicious behavior before it is added to the generated HTML page.

The code in the helper method *changePassword* contains a potential NoSQL injection vulnerability (NoSQLi) (Mitre 2020e). A single atomic action performs the user authentication and a change of a password in line 20 in which two database documents (*filter* and *set*), one with *$where* clause and one with *$set* clause are executed. To report the taint-flow precisely, both values should be marked as tainted. We explain both XSS and NoSQLi vulnerabilities throughout this section.

2.1 Selection of Sensitive Methods

To detect such vulnerabilities using a taint analysis, one must configure the analysis with any security-relevant methods (SM), such as *sources*, *sinks* and *sanitizers*.

Consider the example of the XSS vulnerability in Listing 1. Here, untrusted data flows from the parameter *uName* of the method *doGet* to the sink in line 5 where method *append()* is called with a string value of a request. Figure 1(a) shows the data-flow graph extracted from the code. To fix this vulnerability, a software developer should apply a *sanitizer* such as *encodeHTML()* to clear potential malicious inputs from the variable *uName* before appending the contents to the HTML string. This leads to our first requirement:

> **R1**: *The DSL must allow one to express the following security-relevant methods (*SM*): source, sanitizer, and sink.*

2.2 Selection of In- and Out-Values

Apart from the selection of the call sites, the actual values flowing in or out of the methods (return values, parameters, and receiver) must be selected. For the source of the XSS vulnerability in Listing 1, the developer must select the argument value of the first parameter of

```
1  protected void doGet(@RequestParam("user")String uName,
       HttpServletRequest request, HttpServletResponse response)
       {
2      String oldPass = request.getParameter('oldKey');
3      String newPass = request.getParameter('newKey');
4      if (changePassword(uName, oldPass, newPass))
5      response.getWriter().append('<html>... Password changed
           for user' + uName + '...</html>'));
6      else
7      response.getWriter().append('<html>...
8      Wrong credentials....</html>');
9  }
```

Listing 1 Java code with potential XSS vulnerability (from line 1 to line 5)

```
10  protected boolean changePassword(String uName, String
        oldPass, String newPass) {
11      MongoClient myMongoClient = new MongoClient("localhost",
            8990);
12      MongoDatabase credDB =
            myMongoClient.getDatabase('CREDDB');
13      MongoCollection<Document> credCollection =
            credDB.getCollection('CRED', Document.class);
14      BasicDBObject filter = new BasicDBObject();
15      filter.put('$where', '(username == \"' + uName + '\") \&
            (password == \"' + oldPass + '\")');
16      BasicDBObject newPassDoc = new BasicDBObject();
17      newPassDoc.put('password', newPass);
18      BasicDBObject set = new BasicDBObject();
19      set.put('$set', newPassDoc);
20      UpdateResult res = credCollection.updateOne(filter, set);
21      return (res.getMatchedCount() == 1);
22  }
```

Listing 2: Potential NoSQLi vulnerability (lines 1-3 to line 20)

Listing 2 Java code with potential XSS vulnerability (from line 1 to line 5)

the method *doGet*. At the call to the sink of the vulnerability, the developer needs to provide the possibility to select a parameter of a called method.

R2: *The DSL must allow one to express the data-flow propagation of each* SM *to a granularity of single argument, a return value, and a receiver.*

2.3 Composition of Taint-Flows

The presented XSS vulnerability is detected by what we call a "single-step taint analysis". It is relatively easy to detect, even manually. But many real-world taint-analysis problems comprise a sequence of multiple events. For example, consider the NoSQL injection vulnerability in Listing 2 and its data-flow graph in Fig. 1(b).

The NoSQLi vulnerability occurs in line 20 when the method *updateOne* is called under the condition that the Mongo database has a record with the username and the old password that matches the values coming from the request object (*uName* in line 1 and *oldPass* in line 3). The value of *filter* contains the document that checks the existing password for the given username by calling the method *put* in line 15 with a *$where*-clause. The value of *set* contains the document that sets the new password by calling the method *put* in line 18 with

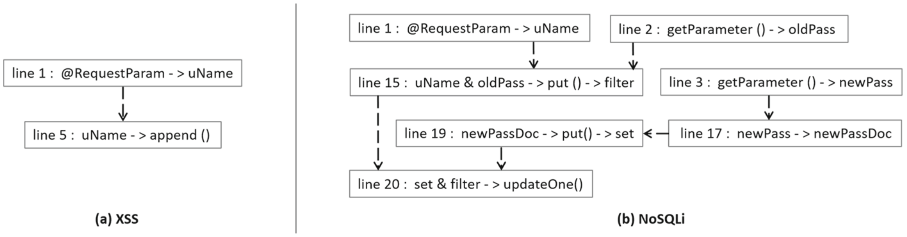
line 1 : @RequestParam - > uName
line 5 : uName - > append ()
(a) XSS
line 1 : @RequestParam - > uName
line 2 : getParameter () - > oldPass
line 15 : uName & oldPass - > put () - > filter
line 3 : getParameter () - > newPass
line 19 : newPassDoc - > put() - > set
line 17 : newPass - > newPassDoc
line 20 : set & filter - > updateOne()
(b) NoSQLi

Fig. 1 Data-flow graphs for (a) XSS and (b) NoSQLi vulnerabilities from Listing 1 and Listing 2

a *$set*-clause. When the method *put* is called in line 15 and line 17, the *uName* and *oldPass* taint the *filter* whereas the *newPass* taints the *set*. For the taint-flow to be complete, *both* calls to the method *put* must occur before the *set* and *filter* flow to the sink *updateOne()* in line 20. Thus, we desired a feature to compose complex queries consisting of multiple single-step taint analyses.

R3: *The DSL must allow one to express complex multi-step taint-flow queries.*

2.4 Detailed Error Message

When findings are reported, the analysis tool usually provides a description to the user to help understanding the vulnerability. The study of Christakis et al. (Christakis and Bird 2016) showed that software developers have difficulties in understanding those descriptions. For different vulnerabilities and types of data-flow the DSL shall present the results of the taint analysis with fine-grained error messages that help developers to quickly identify and fix the vulnerability. The user that specifies the taint-flow should be able to define a custom error message that can be reported at different locations.

R4: *The DSL must allow one to specify error messages for each type of finding.*

2.5 Integration into Developer's Workflow

Empirical studies show that software developers need static analysis tools integrated in their workflow (Christakis and Bird 2016; Johnson et al. 2013). Most software developers use integrated development environments (IDEs) and prefer static analyses to be directly integrated in the IDE. The results of the analysis should be shown within the IDE, preferably visible near the editor for the code. Therefore, a DSL designed for software developers should be integrated in this workflow with appropriate tooling and usability.

R5: *The DSL must integrate well with the software developers' workflow.*

2.6 Independence of Concrete Taint Analysis

Software developers desire reusing taint-flow specifications for both static and dynamic taint analyses. Moreover, some analysis tools are only part of the continuous integration whereas others can be integrated in different workflows, e.g. the IDE. To enable reusability of the specifications among different tools, the DSL semantics must therefore be independent of any concrete static or dynamic analysis. Thus, any limitations due to the approximations of the underlying solver are transferred to the results reported by *fluent*TQL.

R6: *The specified taint-flow queries can be reused among existing taint analysis tools, i.e., the DSL is independent of the underlying taint analysis.*

The NoSQLi vulnerability from the example in this section, can not be detected by default with the existing tools due to its specific structure. Such complex taint-flows require the user to specify a custom query. *fluent*TQL introduced in the next section aims at providing usable and easy approach for mainly software developers specifying custom queries. The existing DSLs are design for experts who have understanding in data-flow analysis, which most developers do not have. Moreover, based on our evaluation of the existing DSLs in Section 5, indicates that none of them completely fulfills all requirements.

3 *fluent*TQL

We next define the domain-specific language *fluent*TQL through its abstract and concrete syntax (Stahl et al. 2006). We also define the runtime semantics of *fluent*TQL as independent of a concrete taint analysis. Dynamic taint analyses could faithfully implement the semantics, whereas static taint analyses would seek to soundly approximate it. Finally, we discuss relevant implementation details.

3.1 Concrete Syntax

As a concrete syntax for *fluent*TQL, we decided to use a Java fluent-interface syntax. Since Java is one of the most popular programming languages, this allows software developers to learn the DSL with little effort. Moreover, in interviews with nine software developers (SecuCheck 2021), the authors asked what concrete syntax they would prefer if given the choice of (1) a fluent interface, (2) a graphical syntax, or (3) a textual syntax for taint-flow queries, six participants chose the fluent interface, and only two chose the graphical and one the textual syntax.

In the following, we explain the concrete syntax by specifying the *fluent*TQL queries for the detection of the XSS and NoSQLi code in Listing 1 and Listing 2. The specification is presented in Listing 3, where lines 23–31 contain the SM declaration and lines 32–38 contain the taint-flow queries.

In the code there are two potential sources. One source is the return value of the *getParameter()* method which in Listing 3 is specified in line 23. The first argument to the constructor of `Method()` takes a method signature as a String argument. Next, using the fluent interface of *fluent*TQL, we append `out()` indicating that the method generates a sensitive data-flow. Eventually, by appending `return()`, we select the return value as the out-value that is generated. The other source is the first parameter of the *doGet()* method (line 24) indicated by `out()` and `param(0)`.

The fluent interface of *fluent*TQL allows calling `out()` or `in()` on a Method object. After `out()` there has to be at least one more call to `return()`, `thisObject()` and/or one or more calls to `param(int)` with the integer referring to the parameter index of the out-value. After `in()` there must be a call to `thisObject()` and/or one or more calls to `param(int)`.

Both sources in line 23 and line 24 are potential sources for SQLi and XSS, i.e., they are not specific to the vulnerability type. Thus, they are grouped into a *MethodSet* object (line 25). Afterwards, the method *encodeHTML* is specified as sanitizer which is relevant to the XSS vulnerability only. The method *put()* is a propagator (i.e. only propagates the taint) but a required one, because it has to be called between the source and the sink for this specific vulnerability. It can be called with two different parameter types. Hence, it is specified twice (lines 27 and 28). They are grouped in the method set *reqPropagatorsPut*. Finally, the sinks are specified (lines 30 and 31). They are specific to each vulnerability type.

The taint-flow query for XSS is specified in line 32 where the class *TaintFlowQuery* is instantiated after which `from(...)`, `to(...)`, and `report(...)` are called.

```
23  Method source1 = new Method("String
        getParameter(String)").out().return();
24  Method source2 = new Method("void doGet(String,
        HttpServletRequest,
        HttpServletResponse)").out().param(0);
25  MethodSet sources = new
        MethodSet().add(source1).add(source2);
26  Method sanitizer = new Method("String encodeHTML
        (String)").in().param(0).out().return();
27  Method reqPropagator1 = new Method("BasicDBObject put(String,
        String)").in().param(1).out().thisObject();
28  Method reqPropagator2 = new Method("DBObject put(String,
        DBObject)").in().param(1).out().thisObject();
29  MethodSet reqPropagatorsPut = new MethodSet().
        add(reqPropagator1).add(reqPropagator2);
30  Method sinkXss = new Method("PrintWriter
        append(CharSequence)").in().param(0);
31  Method sinkNoSql = new Method("FindIterable
        updateOne(BasicDBObject,
        BasicDBObject)").in().param(0).param(1);
32  TaintFlowQuery xss = new
        TaintFlowQuery().from(source1).notThrough(sanitizer)
        .to(sinkXss).report("Reflective XSS
        vulnerability.").at(Location.SOURCE);
33  TaintFlowQuery noSQLi1 = new
        TaintFlowQuery().from(source1).through(
        reqPropagatorsPut).to(sinkNoSql).report(
        "No-SQL-Injection.").at(Location.SINK);
34  TaintFlowQuery noSQLi2 = new TaintFlowQuery();
35  noSQLi2.from(source1).through(reqPropagator1).to(
        sinkNoSql).and().from(source2).through(
36  reqPropagator1).to(sinkNoSql).and().from(source1).
37  through(reqPropagator2).to(sinkNoSql).report(
38  "No-SQL-Injection vulnerability with multiple
        taint-flows").at(Location.SOURCEANDSINK);
39  }
```

Listing 3 *fluent*TQL specification for XSS and NoSQLi in Listings 1 and 2

For the XSS taint-flow query, the sanitizer is also specified by calling the method `notThrough(...)`. Each of these methods expects an object of type *Method* or *MethodSet*.

At the end there is a call to `at(Location.SOURCE)` which is optional and expresses where in the code the report message should be shown. *Location* is an enumeration with values *SOURCE*, *SINK*, and *SOURCEANDSINK*. The taint-flow query can be read as follows: *If there is a taint-flow from the source* `source1` *not propagating through the* `sanitizer` *and reaching any of the* `sinkXss`*, then report a finding with “‘Reflective XSS vulnerability”’ at the source location.*

For the NoSQLi vulnerability there are two taint-flow queries in Listing 3, in lines 33–38. The object *noSQLi1* will report a finding with a message “‘No-SQL-injection vulnerability”’ for the source *getParameter*, defined with *source1*, propagating through any required propagator from the set *reqPropagatorsPut* reaching the *sinkNoSql*. If applied to the code example from Listing 1 and Listing 2, there will be two traces found which will be reported as separate findings. The taint-flow from the first parameter of *doGet* carrying the username will be missed. To detect this taint-flow as well, one can use the method set *sources* instead of the single method *source1*. On the other hand, a taint analysis specified as defined though *noSQLi2* will report a single finding only: For this specification, the three single taint-flows are joined by a call to `and()`, which means all separate taint-flows need to occur individually.

3.2 Abstract Syntax

We discuss the abstract syntax through the meta-model shown in Fig. 2. The DSL has a root node (class *RootNode*) containing all objects. An object of this class represents single instance of the DSL that can contain multiple top level elements. The abstract class *TopLevelElement* is a superclass of the main concepts in *fluent*TQL, i.e., the class *Method* and the class *TaintFlowQuery*.

3.2.1 Methods

The class *Method* represents a reference to a method from the analyzed code. It contains information about the method signature and the data-flow propagation when that method is called in a given context (conforming to **R1** and **R2**). This is expressed through the references to *InputDeclaration* and *OutputDeclaration*. A *Method* object has to have one or

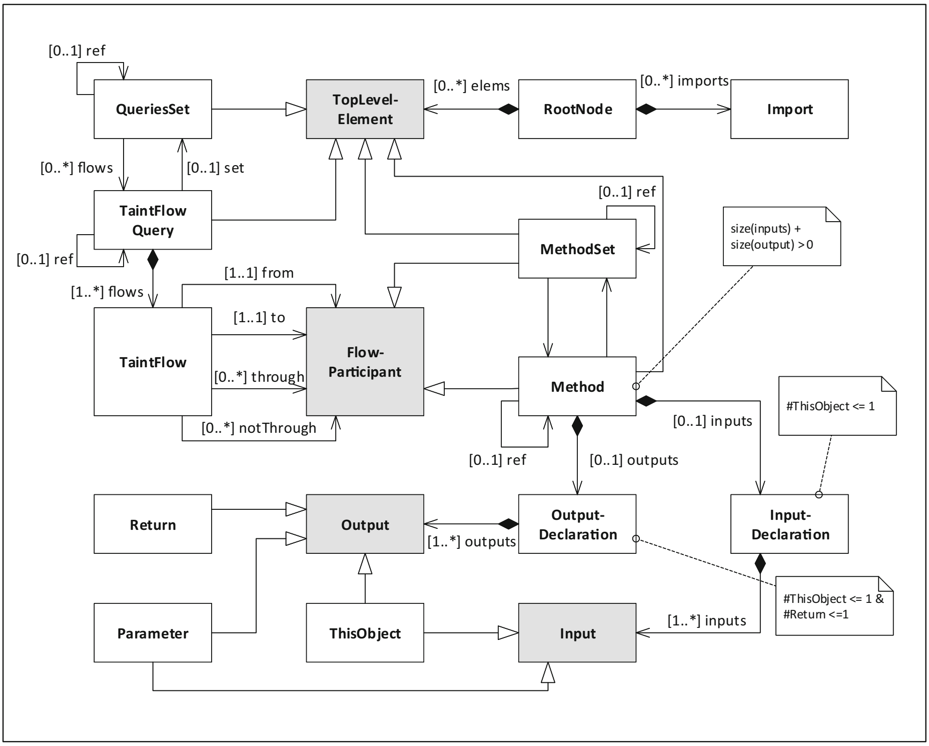

Fig. 2 *fluent*TQL meta-model (UML class diagram, gray-filled classes are abstract). The constraints of the cardinalities of the classes are shown as messages, since the semantics of UML class diagram can not express all of them.

both *InputDeclaration* or *OutputDeclaration* references. An *InputDeclaration* contains an in-value (abstract class *Input*), whereas *OutputDeclaration* contains an out-value (abstract class *Output*). In-values can be a parameter of a method call (class *Parameter*) or a receiver of the method (class *ThisObject*). Out-values can be a parameter, a receiver, or a return value (class *Return*). In-values flow into the method call and out-values flow out of the method call.

The class *Method* in combination with the classes *InputDeclaration* and *OutputDeclaration* can model sources, sinks, and sanitizers (**R1**). Source is a combination of *Method* and *OutputDeclaration* specifying which values become tainted through a method call. Sink is an instance of *Method* and *InputDeclaration* specifying which values must be tainted for the sink to be considered “reached”. Sanitizer is a combination of a *Method* and *InputDeclaration*, specifying which tainted value flowing in the method call will get untainted.

3.2.2 Required Propagators

Required propagators are method calls that have to be on the path between a source and a sink in order for a given vulnerability to be present. For instance the method *put()* in the running example from Section 2 has to be on the taint-flow trace from the source to the sink. It only propagates the taint from the in-value to the out-value. In *fluent*TQL, a required propagator is modeled as a combination of *Method*, *InputDeclaration*, and *OutputDeclaration*. This model allows propagating out-values once an in-value reaches a method. The analyses that are aware of these methods know how to propagate the data-flow without analyzing them, for example for improving scalability or handling calls for which the source code is not available.

3.2.3 Taint-Flow Queries

The class *TaintFlowQuery* represents a taint-flow query. It contains all the information one needs to trigger a taint analysis. It contains one or more *TaintFlow* objects and a user defined message (**R4**). The class *TaintFlow* has four references to the class *FlowParticipant*. The *from* reference defines the set of sources, the *through* reference defines the required propagators, the *notThrough* reference defines the sanitizers, and the *to* reference defines the sinks. For any valid *TaintFlow* there should be at least one source and one sink. A *FlowParticipant* is either a *Method* or a *MethodSet*, i.e., a collection of methods. Similarly, the *QueriesSet* is a collection of taint-flow queries.

3.2.4 Imports and Reuse

The root node can contain imports from other models defined in other locations. This is modeled via the class *Import*. This allows references of methods and taint-flow queries from different files. The classes *Import*, *MethodSet*, and *QueriesSet* are provided for maintenance, reusability, and structure of *fluent*TQL specifications, enabling software developers to define categories of methods and taint-flow queries and share them (**R5**). As Java internal DSL, the users of *fluent*TQL get all advantages of Java compared to any external DSL or XML/JSON-based DSL, often used in the existing tools. From Java, users can reuse existing abstractions such as packaging, modules, and object-oriented design to improve the maintenance, the readability, and the accessibility of the rules.

3.3 Semantics

A taint-flow query, an instance of the class *TaintFlowQuery*, is a *fluent*TQL specification that describes which traces of the program should be returned as findings to the user when a given taint analysis is triggered with that taint-flow query. In the following we define the relevant terms and how *fluent*TQL refers to them.

We denote M to be the set of all method signatures where a signature includes the fully qualified method name, parameter types, and a return type. A sensitive value is a type definition with information about the direction of propagation (in- or out-), and location (return, receiver, or parameter index). Hence, in- and out-values are sensitive values with in- and out-propagation, respectively.

Definition 1 A sensitive method is a tuple *(m,SV)*, where $m \in M$ and *SV* is a set of sensitive values. *SV* contains subset SV_{in} for in-values and subset SV_{out} for out-values.

Definition 2 A taint analysis specification TAS consists of the tuple (*Sources*, *Sanitizers*, *RequiredPropagators*, *Sinks*), where

1. *Sources* is a set of sensitive methods (m, SV_{out}) for which SV contains at least one out-value, $(SV_{out} \neq \emptyset)$,
2. *Sanitizers* and *Sinks* are sets of sensitive methods (m, SV_{in}) for which SV contains at least one in-value, $(SV_{in} \neq \emptyset)$, and
3. *RequiredPropagators* is a set of sensitive methods (m, SV_{in}, SV_{out}) containing at least one in-value and one out-value $(SV_{in} \neq \emptyset, SV_{out} \neq \emptyset)$.

Given a taint analysis specification TAS, some black-box taint analysis T and a program P, we assume the execution of T returns a set of traces for the data-flow, i.e., $T(P, \text{TAS}) = \{t_1 \ldots, t_n\}$ where each t_i is a data-flow trace. A trace is a sequence of program statements, i.e., $t_i = s_i^1 s_i^2 \ldots s_i^n$. For each individual trace t_i it holds that

- the first statement is a source statement, $s_i^1 \in$ *Sources*,
- the last statement is a sink, $s_i^n \in$ *Sinks*
- none of the statement s_i^j is a sanitizer, $s_i^j \notin$ *Sanitizers*, and
- if *RequiredPropagators* is non empty, there exists exactly one element from *RequiredPropagators* that appears at statement s_i^j, where $j \in \{1, \ldots, n\}$.

Note that in the case the analysis T is a dynamic taint analysis, the set of traces is a singleton set while static analyses, which simulate all possible executions, may generate multiple traces.

Example: A TAS can detect rudimentary data-flows modeled with the class *TaintFlow* from Fig. 2 such as the XSS vulnerability in Listing 1. The *TaintFlowQuery xss* in line 32 specifies a *TaintFlow* with

sources -	{*(getParameter(String),*$return^{\text{OUT}}$*), (doGet(String, ...),* 0^{OUT})}
sanitizers -	{*(encodeHTML(String),*0^{IN}*)*}
r. propagators -	{}
sinks -	{*(append(CharSequence),*0^{IN}*)*}

*fluent*TQL allows one to specify these sets with respective syntax elements `from(...)`, `notThrough(...)`, and `to(...)`. Running a taint analysis with the *fluent*TQL specification for *xss* on the code in Listing 1 returns the single trace consisting of the two statements[5] $t_1 = 1\ 5$.

Additionally, the syntax element `through(...)` allows to specify the set of *RequiredPropagators*.

For instance, the taint-flow query *noSQLi1* in line 33 specifies a non-empty set *RequiredPropagators*.

sources - $\{(\mathit{getParameter(String)}, return^{\mathrm{OUT}})\}$
sanitizers - $\{\}$
r. propagators - $\{(\mathit{put(String,\ String)}, 1^{\mathrm{IN}}, return^{\mathrm{OUT}}),\ (\mathit{put(String,\ BasicDBObject)}, 1^{\mathrm{IN}}, return^{\mathrm{OUT}})\}$
sinks - $\{(\mathit{updateOne(BasicDBObject,\ BasicDBObject)}, 0^{\mathrm{IN}}, 1^{\mathrm{IN}}), return^{\mathrm{OUT}}\}$

The result of the taint-flow query *noSQLi1* is

$$Trace_{noSQLi1} = 2\ 4\ 10\ 15\ 20,\ \ 3\ 4\ 10\ 17\ 19\ 20$$

and consists of two traces. Each of these traces is reported as a separate finding to the user. A "simple" finding is a single trace with a single message. For instance, the findings of *noSQLi1* are $Findings_{noSQLi1} = \{\ \mathrm{F}^1_{noSQLi1}, \mathrm{F}^2_{noSQLi1}\}$, where

$F^1_{noSQLi1} = (\{2\ \ 4\ \ 10\ \ 15\ \ 20\}$, *"No-SQL-Injection vulnerability."*)
$F^2_{noSQLi1} = (\{3\ \ 4\ \ 10\ \ 17\ \ 19\ \ 20\}$, *"No-SQL-Injection vulnerability."*)

Yet, for more complex queries one can use the *and()* operator, which combines findings over individual traces to a single finding over multiple traces.

Combining taint-flow queries: The *and()* operator allows one to merge multiple TAS as a single query. This is through an object of type *TaintFlowQuery* (from Fig. 2) that contains multiple objects of type *TaintFlow*. Formally, the operator computes the cross product of the traces of the individual TAS. For example, the taint-flow query *noSQLi2* in line 34 defines three TAS specifications:

sources - $\{(\mathit{getParameter(String)}, return^{\mathrm{OUT}})\}$
sanitizers - $\{\}$
r. propagators - $\{(\mathit{put(String,\ String)}, 1^{\mathrm{IN}}), return^{\mathrm{OUT}}\}$
sinks - $\{(\mathit{updateOne(BasicDBObject,\ BasicDBObject)}, 0^{\mathrm{IN}}, 1^{\mathrm{IN}})\}$

sources - $\{(\mathit{put(String,\ String)}, return^{\mathrm{OUT}})\}$
sanitizers - $\{\}$
r. propagators - $\{\}$
sinks - $\{(\mathit{updateOne(BasicDBObject,\ BasicDBObject)}, 0^{\mathrm{IN}}, 1^{\mathrm{IN}})\}$

sources - $\{(\mathit{getParameter(String)}, return^{\mathrm{OUT}})\}$
sanitizers - $\{\}$
r. propagators - $\{(\mathit{put(String,\ BasicDBObject)}, 1^{\mathrm{IN}}, return^{\mathrm{OUT}}))\}$
sinks - $\{(\mathit{updateOne(BasicDBObject,\ BasicDBObject)}, 0^{\mathrm{IN}}, 1^{\mathrm{IN}})\}]$

The first one returns the trace "2 4 10 15 20", the second one returns the trace "1 4 10 15 20", and the last one returns the trace "3 4 10 17 19 20". Yet, the result of the query *noSQLi2* will be a single finding, $Findings_{noSQLi2} = \{F^1_{noSQLi2}\}$, where

[5] We use line numbers from Listing 1 and Listing 2 to represent traces.

$F^1_{noSQLi2} = (\{2\ 4\ 10\ 15\ 20,\ \ 1\ 4\ 10\ 15\ 20,\ \ 3\ 4\ 10\ 17\ 19\ 20\}$, *“No-SQL-Injection vulnerability with multiple taint-flows.”*).

Calculating Traces By its definition, *fluent*TQL has a precise runtime semantics. However, when applied in static context, the traces need to be approximated by the underlying data-flow engine. Thus, reported traces of different tool implementations can differ.

To explain the precise runtime semantics for traces construction, we define a taint analysis core language, in similar fashion to previous works (Schwartz et al. 2010; Livshits 2012). Though simple, the core language covers relevant statements that can be mapped one-to-one with Java statements. The statements are listed in Table 1. A program of the language contains a sequence of statements with line number. For simplicity, we decided to exclude method calls from the language. These can be compiled to the language by storing the memory address of the return statement and transferring the control flow. This rule is not applied to the four statements in Table 1 which are special method calls.

We denote variables with x and y, a field of an object with f, and an i-th index of an array with $a[i]$. We model all memory locations through a shadow heap: The shadow-heap values for a memory location v is *true* if the value is tainted and *false* otherwise. The execution context Σ has the parameters listed in Table 2. $\Sigma.\Delta[x]$ stores the current taint value of variable x. We write $\Sigma \vdash x \Downarrow v$ to extract that value into v. Similarly, notations like $src(x) \vdash (m, SV)$ extract the method m with its sensitive values SV, when a method call src is matched. Additionally, Σ stores all traces $t \in T$ that will be created during the execution. t is a sequence of statements (which we here denote by line numbers).

Figure 3 shows *fluent*TQL’s semantics through inference rules. We use a syntax akin to the one used by Schwartz et al. (Schwartz et al. 2010). The semantics essentially define a regular dynamic taint analysis which, as side-effect collects un-sanitized traces from sources to sinks. For instance, given the statement $x = y$, the ASSIGN rule’s computation comprises four parts. First, $\Sigma \vdash y \Downarrow v$ evaluates and extracts the taint value v for variable y. Due to the assignment, the rule updates the taint value of x with v. The rule then also extracts each trace in $t \in \theta$ and adds to it the current statement, identified by its line number. The rules for load/store and array accesses are equivalent. The rule SOURCE creates a new trace and taints the out-value, the rule SINK gracefully terminates a trace by untainting the sensitive value. The rule SANITIZER also discontinues the tracing. The rule PROPAGATOR taints the out-value if the in-value is tainted. SKIP advances to the next statement whereas SEQ

Table 1 Statements of the core language for constructing *fluent*TQL traces

Statement	Description
src(x)	Call to a sensitive method *(m,SV_{out})* ∈*Sources* with sensitive parameter *x*
snk(x)	Call to a sensitive method *(m,SV_{in})* ∈*Sinks* with leaked parameter *x*
san(x)	Call to a sensitive method *(m,SV_{in})* ∈*Sanitizers* sanitizing parameter *x*
rpr(x)	Call to a sensitive method *(m,SV_{in}, SV_{out})* ∈*RequiredPropagators*
$x = y$	Assignment
$x = y.f$	Field load
$x.f = y$	Field store
$x = a[i]$	Read from array at index i
$a[i] = x$	Write to array at index i
$skip$	Skip and continue

Table 2 Statements of the core language for constructing *fluent*TQL traces

Parameter	Description
Δ	Match a variable, a field, an array element or a sensitive value to its taint value
λ	Match a given statement to its line number
θ	Returns the set of all traces created

enables the progression of the semantics covering the recursive case. The semantics must additionally enforce one aspect that we found hard to capture with inference rules: for such *fluent*TQL specifications that define required propagators, the taint analysis must ensure to report only such traces that actually contain all required propagators. Finally, the notion of user-defined message is skipped in the formal semantics due to simplicity, but we explain it in the following through our example.

$$\frac{\Sigma \vdash y \Downarrow v \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \qquad t \in \Sigma.\theta \qquad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x = y]\rangle]}{(\Sigma, x = y) \rightsquigarrow (\Sigma', skip)} \quad \text{(ASSIGN)}$$

$$\frac{\Sigma \vdash y.f \Downarrow v \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \qquad t \in \Sigma.\theta \qquad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x = y.f]\rangle]}{(\Sigma, x = y.f) \rightsquigarrow (\Sigma', skip)} \quad \text{(LOAD)}$$

$$\frac{\Sigma \vdash y \Downarrow v \qquad \Sigma.\Delta' = \Sigma.\Delta[x.f \leftarrow v] \qquad t \in \Sigma.\theta \qquad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x.f = y]\rangle]}{(\Sigma, x.f = y) \rightsquigarrow (\Sigma', skip)} \quad \text{(STORE)}$$

$$\frac{\Sigma \vdash a[i] \Downarrow v \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \qquad t \in \Sigma.\theta \qquad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[x = a[i]]\rangle])}{(\Sigma, x = a[i]) \rightsquigarrow (\Sigma', skip)} \quad \text{(READ-ARRAY)}$$

$$\frac{\Sigma \vdash x \Downarrow v \qquad \Sigma.\Delta' = \Sigma.\Delta[a[i] \leftarrow v] \qquad t \in \Sigma.\theta \qquad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[a[i] = x]\rangle]}{(\Sigma, a[i] = x) \rightsquigarrow (\Sigma', skip)} \quad \text{(WRITE-ARRAY)}$$

$$\frac{\begin{array}{c} src(x) \vdash (m, SV_{out}) \\ sv \in SV_{out} \qquad \Sigma.\Delta' = \Sigma.\Delta[sv = true] \qquad \Sigma.\theta' = \Sigma.\theta \cup \{\lambda[src(x)]\} \end{array}}{(\Sigma, src(x)) \rightsquigarrow (\Sigma', skip)} \quad \text{(SOURCE)}$$

$$\frac{\begin{array}{c} snk(x) \vdash (m, SV_{in}) \\ sv \in SV_{in} \qquad \Sigma \vdash sv \Downarrow true \qquad t \in \Sigma.\theta \qquad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[snk(x)]\rangle] \end{array}}{(\Sigma, snk(x)) \rightsquigarrow (\Sigma', skip)} \quad \text{(SINK)}$$

$$\frac{san(x) \vdash (m, SV_{in}) \qquad sv \in SV_{in} \qquad \Sigma.\Delta' = \Sigma.\Delta[sv = false]}{(\Sigma, san(x)) \rightsquigarrow (\Sigma', skip)} \quad \text{(SANITIZER)}$$

$$\frac{\begin{array}{c} rpr(x) \vdash (m, SV_{in}, SV_{out}) \qquad sv_1 \in SV_{in} \qquad sv_2 \in SV_{out} \\ \Sigma \vdash sv_1 \Downarrow true \qquad \Sigma.\Delta' = \Sigma.\Delta[sv_2 = true] \qquad \Sigma.\theta' = \Sigma.\theta[t \leftarrow \langle t, \lambda[rpr(x)]\rangle] \end{array}}{(\Sigma, rpr(x)) \rightsquigarrow (\Sigma', skip)} \quad \text{(PROPAGATOR)}$$

$$\frac{}{(\Sigma, skip; S) \rightsquigarrow (\Sigma, S)} \quad \text{(SKIP)} \qquad \qquad \frac{(\Sigma, S_1) \rightsquigarrow (\Sigma', S_1')}{(\Sigma, S_1; S_2) \rightsquigarrow (\Sigma', S_1'; S_2)} \quad \text{(SEQ)}$$

$$\frac{}{\Sigma \vdash v \Downarrow \, false} \quad \text{(FALSE)}$$

Fig. 3 Inference rules of the operational semantics of the traces construction in *fluent*TQL

Report Message As seen in the previous examples, the queries specified in *fluent*TQL contain a user-defined message which is added to each finding (**R4**). In the concrete syntax, the mandatory syntax element `report(...)` takes the string message as an argument. Optionally, the user may specify the location for the reporting message by using the syntax element `at(...)`. As an argument, the enumeration *Location* can be used, which contains three elements *SOURCE, SINK* and *SOURCEANDSINK*. *SOURCE* and *SINK* define that the reporting message should be shown at the source and the sink location respectively. For *SOURCEANDSINK* the message should be shown at both source and sink location in the code. If the finding has multiple traces then the reporting message is shown for each trace individually. For example, for *noSQLi2* the error message will be shown at each source and sink location, i.e., lines 1, 2, 3, and 20, because *SOURCEANDSINK* is used in the query specification (Listing 3, line 38). This information can be used by tools for visualization purposes. E.g. an IDE plug-in may display error markers in the editor at the source location, the sink location, or both.

Usability Versus Expressiveness *fluent*TQL is a DSL for users without deep expertise in static analysis as most software developers. Its purpose is to enable users specify a custom taint analysis for their codebase and detect many popular security vulnerabilities. Hence, the usability and simplicity of the language is the primary aim. A trade-off to this design decision is the lower expressiveness when compared to some existing DSLs such as CodeQL. *fluent*TQL does not provide the users a fine-grained manipulation of the abstract syntax tree (AST). Such expressive DSLs are used by program analysis experts. This fine-grained AST manipulation, can be useful for writing more compact code. Nonetheless, as our evaluation in Section 4 shows, most popular security vulnerabilities can be expressed in *fluent*TQL. This is due to the fact that the relevant data being tracked by the analysis is impacted only by specific method calls within the program, which is the case for most security vulnerabilities. Compared to CodeQL, in *fluent*TQL, other language constructs than method calls currently can't be modeled. However, as discussed in Section 4.4, extending *fluent*TQL with new language constructs is possible without significant semantic changes. On the side of expressiveness, *fluent*TQL has a support of **R3** which is only partially supported by other DSL as can be seen later in Section 5. Complex multi-step taint-flow queries are in particular relevant for stored versions of SQLi and XSS vulnerabilities. Finally, *fluent*TQL only support taint analysis, whereas other DSLs like CodeQL support additional types of analyses, such as value analysis.

3.4 Implementation

We implemented *fluent*TQL as an internal Java DSL which can be easily used in any Java project by implementing the interface *FluentTQLSpecification*. Hence, any Java editor can be used to write and edit *fluent*TQL queries.

Additionally, we implemented a server using the MagpieBridge framework (Luo et al. 2019) that can trigger, execute the analysis, and return the results to the IDE. *fluent*TQL is implemented as a standard Java library using the builder pattern to allow method chaining as user interface. All queries need to be implemented within a class that implements the interface *FluentTQLSpecification*. Using the Java classloader the classes are located and the queries correctly loaded and provided as input to the analysis.

As we rely on MagpieBridge, we support IDEs that support the Language Server Protocol (Microsoft 2020) such as Vim, Eclipse, VSCode, IntelliJ, and many more. The MagpieBridge server uses the Language Server Protocol to notify the IDE for available results. Figure 4 shows a component diagram of our implementation. The core analysis uses Soot as an underlying static analysis framework responsible for providing the main data structures, such as control-flow graph and call graph. Solvers such as Boomerang and FlowDroid provide interface for starting a taint analysis. The core analysis utilizes this to execute the semantics of the *fluent*TQL queries as described previously in *fluent*TQL's semantics. The core analysis matches the solver's APIs with the *fluent*TQL queries that are executed. For complex queries it breaks them into simple taint-flows which are independently solved by the underlying solver and their results merged afterwards. The queries are loaded through the *fluent*TQL-classloader into the MagpieBridge-Server.

Our implementation uses the standard IDE features: errors view, editor markups, and notifications to display the results from the analysis directly in the IDE. Additionally, it provides a configuration page where the user can filter the queries and the entry points used for the call graph used by the analysis.

To instantiate *fluent*TQL with concrete analyses, we first implemented a taint analysis built on top of the Boomerang solver (Späth et al. 2019), an efficient and precise context-, flow-, and field-sensitive data-flow engine with demand-driven pointer analysis. Boomerang provides an API to query all traces from given seeds. The API of the seed is expressible to cover the *fluent*TQL semantics of the sensitive methods. However, the basic API of Boomerang does not support sanitizers, nor required propagators. To support the sanitizers we transformed the bodies of the sanitizers to empty, which is a terminal case of the Boomerang data propagation solver. To support required propagators, we break the TAS specification to multiple TAS specifications containing only sources and sinks. A TAS with required propagator is broken to two TAS where the first one has the original source and the required propagator as sink, whereas the second one has the required propagator as source and the original sink as sink. Boomerang returns the traces of the individual TAS, and our implementation merges them. There is no explicit well-formed check in our implementation. However, we implemented the taint analysis with Boomerang on ourselves and we, therefore, trust its correctness with respect to the semantics of the constructed traces. However, future implementation with other solvers, should also include a well-formed check.

Moreover, we instantiated *fluent*TQL with the existing taint analysis of FlowDroid (Arzt et al. 2014). This, however, was not possible without limitations. Specifically, the default component for defining sources and sinks in FlowDroid is limited and supports only return as out-value of sources and parameter index as in-value of sinks. This can be extended by

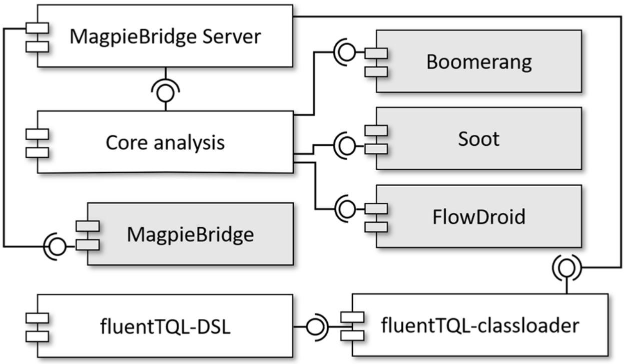

Fig. 4 Component diagram of the *fluent*TQL imeplementation as MagpieBridge server (gray components are external, white components are internal)

adding new implementation of the *SourceSinkManager*, which we left as future work. Sanitizers by default are not supported, but we applied the same solution as in our Boomerang implementation, whereas required propagators are not supported and requires either extension of the taint analysis or post-processing of the findings which we also consider as future work.

Finally, both instances of *fluent*TQL have some limitations in the way the traces are constructed and reported. Since *fluent*TQL has precise runtime semantics, it is expected that static analysis engines like Boomerang and FlowDroid will approximate. In particular, both engines will unsoundly underapproximate the constructed traces. For example, both apply different strategies for merging conditional paths of the program. Thus, these limitations are part of our implementation, too.

4 Evaluation

We evaluated the usability of *fluent*TQL by conducting a comparative user study between *fluent*TQL and CodeQL. We chose CodeQL because it is part of LGTM, a state-of-the-art security tool, which has, in our perspective, very good tool support and the query specifications are open-source. There is also an Eclipse plugin, a web console for queries, and integration with GitHub, a popular versioning system among developers. Additionally, we evaluated the applicability of *fluent*TQL by specifying queries for different set of applications: a catalog of eleven Java programs, each demonstrating different security vulnerability, the deliberately insecure application OWASP WebGoat aiming to teach developers about relevant security vulnerabilities, an insecure version of the Spring Demo application PetClinic, and randomly selected five real-world Android apps with known malicious taint-flows part of TaintBench (Luo et al. 2021). All selected applications have known expected taint-flows that can be used to evaluate how does the analysis perform in finding real vulnerabilities. We answer the following research questions:

- **RQ1** How usable is *fluent*TQL for software developers?
- **RQ2** How does *fluent*TQL compare to CodeQL for specifying taint-flow queries for taint-style security vulnerabilities?
- **RQ3** Are *fluent*TQL syntax elements sufficient to express queries for popular taint-style security vulnerabilities?
- **RQ4** Can *fluent*TQL express and detect known security vulnerabilities in Java/Android applications?

To answer the research questions, we use corresponding metrics. For **RQ1**, we use the System Usability Scale and Net Promoter Score. The same metrics are also used in **RQ2** to compare both DSLs. Additionally, we measure the time needed for the participants to complete the given tasks. We count only the solutions which are complete queries. The partial solutions are not counted due to the nature of the task. In similar realistic scenario, incomplete queries will not return results from the tools. For **RQ3**, we evaluate how each *fluent*TQL construct contributes in specifying the most popular Java security vulnerabilities. Moreover, we identify security vulnerabilities for which *fluent*TQL can not express the required constructs. Finally, for **RQ4**, we count how many of the expected taint-flows in the selected applications are found when *fluent*TQL runs with adequate queries.

The following subsection explains our methodology for the user study used to answer **RQ1** and **RQ2**. The next subsections discuss the results of each research question individually. Finally, we discuss threats to validity.

4.1 Methodology

Setup The user study was conducted over a set of teleconferences where each participant shared the screen. Each study took on average 80 minutes. The session was recorded for post-processing purposes. We invited 35 software developers to take part in the study, from which 26 accepted the invitation, referred to as P01-P26. We invited professional developers via our contacts from the industry as well as researchers and master level students. Additionally, we asked three students to participate in a test session, which helped us to estimate the time and adjust the difficulty of the tasks.

Due to the limited number of participants, we chose a within-subjects design. Hence, each participant worked in Eclipse with available tool support for both DSLs. The *fluent*TQL implementation used the more versatile instantiation based on Boomerang. To avoid any bias, we referred to the DSLs by DSL-1 and DSL-2. Initially, the participants received a project with all files needed for the practical part. The moderator gave an introduction to taint analysis and showed a Java code example with an SQL injection vulnerability (Mitre 2020h) to make sure that the participant understands the required concepts such as source, sanitizer, required propagator, and sink. Then, the exercises for DSL-1 and DSL-2 followed.

Each exercise consisted of a tutorial and a task. The tutorial for each DSL was based on the SQL injection vulnerability. Then, the participants had ten minutes to write a specification in the same DSL for a new vulnerability explained by the moderator. We chose the vulnerability types open redirect (Mitre 2020l) for *fluent*TQL and cross-site scripting (Mitre 2020d) for CODEQL. For either type, we selected an example with the same pattern in form of source-sanitizer-sink. This ensures that writing a specification for each vulnerability is equally hard, i.e., the effort is the same regardless of the vulnerability.

For each vulnerability type, we provided a Java code example as a reference. The participant was allowed to use any of the files provided that included the Java classes and the files with example specifications of *fluent*TQL and CODEQL. For each task, we additionally provided a file with a skeleton code in which the participant wrote the solution. During the tasks, the participants were allowed to ask questions for clarification.

After the tasks, we let the participants fill a web form. The moderator guided the participant in the discussion and collected the data for the questionnaire.

Questionnaire In total the questionnaire asked 28 questions, of which two are of open type and optional (Q26 and Q27). The complete list of questions is part of our artifact. Each of the questions asks for feedback for each DSL by the participant. From the 26 mandatory questions of closed type, 4 are informational, 20 are related to the System-Usability-Scale (SUS) (Brooke 2013), and two are related to the Net Promoter Score (NPS) (Reichheld 2003). The SUS value is a usability metric that can be calculated with ten simple questions in a predefined format. The SUS-related questions (Q4–Q23) are the same ten questions per DSL with answering options on agreement scale from one to five. SUS expresses usability of a single DSL. Hence, for comparison we use the same questions for each DSL. The NPS metric expresses how likely the participant would recommend something to a colleague. To calculate a value, NPS identifies so-called promoters and detractors among the participants. The NPS-related questions (Q24–Q25), ask for the likelihood of DSL1 being recommended over DSL2 for the task of specifying taint-flow queries and vice versa. The informational questions ask about participant coding experience (Q1), security expertise (Q2), willingness to learn a new DSLs (Q3), and preferred way of learning new languages (Q28).

Participants The study population with 26 participants is larger than the size of related studies that have been performed earlier, e.g. 10 in (Smith et al. 2019), 12 in (Smith et al. 2020), and 22 in (Nguyen Quang Do and Bodden 2020). We chose participants with a diverse background. Ten of them are professional developers, six are computer science students on the master level, and ten are researchers in computer science. The participants have different experiences in programming. Twelve of the participants have 10+, nine have 6–10, four have 3–5, and one has 1–2 years of programming experience. They rated their experience with security vulnerabilities. Three consider themselves as beginners, 16 have basic knowledge, five regularly inform themselves about the topic, and two consider themselves as experts.

Statistical Tests Along with the reported data and metrics, we perform relevant statistical tests. As a within-subject design our collected data is paired, i.e, for each participant we have one set of collected data. Exception are the SUS and NPS metrics which are aggregated among all participants. The limitation of this design is the possibility of carryover effects, such as learning effects. The main treatment variable is the *technique*, stating which DSL was used to solve each task (nominal data). In addition, we have an independent crossover treatment variable, the choice of DSL for the first task (binomial data). The background variable are: years of coding experience (ordinal data), position (nominal data), and security experience (nominal data). Finally, we have two effort variables, one for the outcome of each task (binomial data) and the time (ratio data). As most of the data is nominal and ordinal, we used only non-parametric statistical tests. Bellow we report individually each selected test and the results. We used the significance level $\alpha = 0.05$ for all the tests.

4.2 RQ1 Usability of *fluent*TQL

*fluent*TQL was positively received by the participants of our user study. It received an excellent System Usability Score of 80,77 on a scale from 0 to 100 where 68 is considered to be an average usability and 100 is imaginary perfect.

Using the null hypothesis "*fluent*TQL *is usable (SUS is bigger then the hypothetical value of 68)*", we select the Wilcoxon test (the data is ratio but without normal distribution). The test accepts the hypothesis with statistical significance and large effect size (>0.5). For the given task, 20 out of 26 participants have finished with a correct solution in 10 minutes (on average 472 s, with $\sigma = 99, 05$). Table 3 shows the exact time in seconds for each participant. In the open questions (Q26–Q27), many of the participants gave additional feedback what they like and what they would improve in *fluent*TQL. Most of the participants said that they can learn the language very easily, one of them said "'*with simple tutorial, I can learn it (fluent*TQL*) even without an expert. (...) it was very intuitive*'" and other said "'*I didn't have to learn a lot*'". Few participants mentioned that they like that the queries are compact and have the right level of abstraction.

We noted a few points that many participants disliked. Most dislike that the method signatures are specified as a string value. One participant said "'*method calls are prone to typos or cumbersome to create*'". For this, we already added a check in the editor to inform the users if their string is an invalid method signature. We support Java and Soot signatures. We even plan to add suggestions for existing methods from the workspace to the code completion feature of the editor. Some participants gave suggestions for improving the names of some keywords. For example the class *ThisObject*, which in *fluent*TQL is called with *thisObject()*, was earlier called *This* and confused many participants with the *this* keyword in Java.

Table 3 List of participants: coding experience and position, time in solving each task and DSL used in the first task (X means the participant did not solve the task in 10 minutes)

	Coding (years)	Position	Security experience	*fluent*TQL (seconds)	CodeQL (seconds)	1st DSL
P01	3–5	Developer	Basic	554	X	*fluent*TQL
P02	>10	Developer	Basic	499	X	*fluent*TQL
P03	6–10	Student	Expert	482	588	*fluent*TQL
P04	>10	Researcher	Basic	560	590	CodeQL
P05	>10	Researcher	Basic	X	591	*fluent*TQL
P06	>10	Researcher	Advanced	544	562	*fluent*TQL
P07	3–5	Researcher	Basic	X	595	*fluent*TQL
P08	>10	Student	Advanced	449	495	CodeQL
P09	6–10	Student	Basic	X	587	CodeQL
P10	>10	Researcher	Basic	545	567	CodeQL
P11	1–2	Researcher	Beginner	558	585	CodeQL
P12	6–10	Researcher	Basic	X	X	*fluent*TQL
P13	3–5	Researcher	Beginner	473	541	CodeQL
P14	6–10	Researcher	Basic	305	434	CodeQL
P15	6–10	Researcher	Basic	571	X	*fluent*TQL
P16	6–10	Student	Beginner	412	558	CodeQL
P17	>10	Developer	Basic	X	X	*fluent*TQL
P18	>10	Developer	Basic	328	600	CodeQL
P19	6–10	Developer	Basic	594	X	*fluent*TQL
P20	6–10	Developer	Expert	375	492	CodeQL
P21	6–10	Student	Basic	455	467	CodeQL
P22	>10	Developer	Advanced	X	X	CodeQL
P23	>10	Developer	Advanced	507	600	*fluent*TQL
P24	>10	Developer	Advanced	206	425	CodeQL
P25	3–5	Student	Basic	531	X	*fluent*TQL
P26	>10	Developer	Basic	492	X	*fluent*TQL

> *fluent*TQL as a new DSL is found to be very usable. The participants of our user study gave a score of 80,77 on the System Usability Score system.

4.3 RQ2 Comparison of *fluent*TQL and CodeQL

In terms of usability, with a SUS value of 38,56 CodeQL is perceived with bad usability. Using the null hypothesis "CodeQL *is not usable* (SUS is smaller than the hypothetical value of 68)", we select the Wilcoxon test. The test accepts the hypothesis with statistical significance and large effect size (>0.5). On the questions how likely will the participant recommend one DSL over the other for the task they were given (Q24-Q25), *fluent*TQL over CodeQL has a Net Promoter Score value of 30,77, whereas CodeQL over *fluent*TQL has a value of −86,96, where on the scale from −100 to 100, positive values are considered good. It follows that for specifying taint-flow queries, participants would more likely recommend *fluent*TQL over CodeQL.

To compare both languages, let us consider the CodeQL example for XSS in Listing 4. This is a solution for the task given to the participants. The query (lines 47–49) consists of three sections, *from*, *where*, and *select*. In the *from* section, the user defines objects from predefined or self-defined classes. In the *where* section, constraints are defined that may also contain calls to predicates. In the *select* section, the results of the query are defined. For taint analysis, CodeQL provides a module. The class *XSSConfig* extends from the configuration class for taint analysis where the sources, sanitizers, and sinks are defined. Additionally, the classes *RemoteFlowSource* and *XssSink* are provided and can be used to detect sources and sinks for XSS. The stub code with relevant imports given to each participant contained information that these classes exist and can be used. A user who needs other SM that the provided classes cannot detect, will need to write a new implementation. Note that the provided classes *RemoteFlowSource* and *XssSink* will match more sources and sinks than the *fluent*TQL query solution. To have an equivalent query as the one in *fluent*TQL, the participants would have to write additional code for the *isSource* (Line 42) and *isSink* (Line 43) methods instead of using the provided classes.

Few participants mentioned the amount of code they would need to write in CodeQL is large. One participant said, *"'...way too much code to get to the actual thing that needs to be written."'*.

Furthermore, we observed how each participant performed in solving the tasks. The task with CodeQL was solved by 17 participants, compared to 20 with *fluent*TQL. Fourteen participants solved both tasks. However, on this data, the Fisher's test (selected due to binomial small sample) did not indicate a statistical significance. We measured the time each participant needed for each task, which is given in Table 3. On average participants solved the task with CodeQL in 546 s ($\sigma = 57,89$), which is by 13,4% slower than with *fluent*TQL. Using the Wilcoxon test we found a statistical significance for the null hypothesis with a small effect.

We performed few additional Wilcoxon tests for the impact of the background variables, i.e. *Coding*, *Position*, and *Security experience*. None of these tests showed a statistical significance of the null hypothesis which tested whether the variable impacts the time of solving the tasks. Finally, we look into the outcome of each task. The null hypothesis is "*The order of the tasks impact the output*". We used the two-way ANOVA (Girden 1992) test, which

```
40      class XSSConfig extends TaintTracking2::Configuration {
41          XSSConfig() { this = "XSSConfig" }
42          override predicate isSource(DataFlow::Node source) {
                source instanceof RemoteFlowSource }
43          override predicate isSink(DataFlow::Node sink) { sink
                instanceof XssSink }
44          override predicate isSanitizer(DataFlow::Node node) {
45              node.getType() instanceof NumericType or
                    node.getType() instanceof BooleanType}
46      }
47      from DataFlow2::PathNode source, DataFlow2::PathNode
            sink, XSSConfig conf
48      where conf.hasFlowPath(source, sink)
49      select sink.getNode(), source, sink, "Cross-site
            scripting due to $@.", source.getNode(),
            "user-provided value"
50  }
```

Listing 4 CodeQL specification for XSS

did not show a statistical significance. Hence, we reject the null hypothesis and accept the alternative one stating that the order of the tasks does not impact the outcome.

While CodeQL is more expressive DSL for multiple types of static analyses, *fluent*TQL is more preferred among software developers due to its user-friendliness. CodeQL scored a bad SUS value of 38,56. On the NPS system, *fluent*TQL is preferred over CodeQL with a score of 30,77, whereas CodeQL is preferred over *fluent*TQL with a negative value of 86,96. When specifying a taint-flow for given known vulnerability, the participants in our user study were 13,4% faster when using *fluent*TQL compared to CodeQL.

4.4 RQ3 Expressiveness

To evaluate whether *fluent*TQL syntax elements are sufficient to express popular Java taint-style vulnerabilities, we created a catalog with Java code examples accompanied by *fluent*TQL specifications. The catalog contains eleven types of security vulnerabilities (Table 4). Each Java code example has a variant with and without sanitization. The catalog demonstrates different language syntax elements of *fluent*TQL and how they can be used for specifying vulnerabilities. The Java examples and the SM are manually collected from several sources including the Mitre (2020b) and OWASP (2020b) databases, OWASP benchmark project (OWASP 2020a), and other publicly available SM lists (Arzt et al. 2013; Brooke 2013; Piskachev et al. 2019; Thomé et al. 2017).

Many of the taint-style vulnerabilities from the Mitre and OWASP databases can be modeled with single taint flow queries. Yet, we found some examples such as the *noSQLi2* query in Listing 3 where the and() operator is needed.

Taint flows that require multiple intermediate source-sinks steps were necessary for the specification of many taint flows, i.e., the feature of multi-step taint analysis is ubiquitous.

Table 4 List of vulnerability types implemented in the *fluent*TQL catalog (so - sources, sa - sanitizers, rp - required propagators, si - sinks)

Vulnerability type	flows	so	sa	rp	si	SM
SQL injection (Mitre 2020h)	3	13	3	6	10	32
XPath (Mitre 2020c)	1	12	1	0	12	25
Command injection (Mitre 2020f)	1	12	1	1	1	15
XML injection (Mitre 2020m)	1	12	1	0	4	17
LDAP injection (Mitre 2020g)	1	12	1	0	8	21
Cross-site scripting (Mitre 2020d)	2	13	1	1	3	18
Open redirect (Mitre 2020l)	2	13	1	0	2	16
NoSQL injection (Mitre 2020e)	2	5	2	3	2	12
Trust boundary violation (Mitre 2020k)	1	12	1	0	1	15
Path traversal (Mitre 2020j)	2	12	1	1	2	16
Log injection (Mitre 2020i)	2	12	1	1	4	18
Total (unique):	18	46	14	13	49	122

For example, the OWASP Benchmark test 00001[6] contains Path Traversal vulnerability (Mitre 2020j). A *File* object is constructed using a *String* parameter as location to the file. If the *String* is user-controllable, i.e., tainted, and the *File* object is passed to a *FileInputStream* constructor, a path traversal vulnerability occurs. The file constructor in this case is a required propagator that ensures the order of SM calls.

When it comes to the SM specifications, we observed that most of the sources have a *Return* object as an out-value. For sinks, most of the in-values are *Parameter* objects.

Additionally, we inspected the vulnerability types (known as Common Weakness Enumerations - CWEs) in the SANS-25 list (Mitre 2020a). 17 of 25 vulnerability types can be expressed as taint-style. 13 of those can be modeled in *fluent*TQL.

The remaining four CWEs are: CWE-119, CWE-787, CWE-476, and CWE-798. Both, CWE-119 and CWE-787 are related to buffer overflows, which do not apply to Java. The CWE-476 cannot be expressed because the potential sources are `new`-expressions, which cannot currently be modeled. Also, constant values cannot currently be modeled as potential sources which is needed for CWE-798 where these values should be detected as hard-coded credentials. Extending *fluent*TQL to support `new`-expressions and constant values is possible in the abstract syntax by modeling them with a new class that extends the class *FlowParticipant*. The semantics needs to be extended to define how these values will be detected and define appropriate concrete syntax.

Even though our implementation of *fluent*TQL if bound to Java only, *fluent*TQL can express taint-style vulnerabilities in other languages too. To specify a query for other languages, the only requirement is that the sources, sanitizers, and sinks are defined as method calls. Similar to Java, *fluent*TQL can be adapted to work for C/C++, C#, other JVM-hosted languages and cover a wide range of taint-style vulnerabilities. In languages such as JavaScript, the coverage of vulnerabilities is smaller since the sources and sinks are often not method calls.

> *fluent*TQL is able to express all taint-style vulnerabilities in which the key constructs of the taint-flows are method calls. Our implementation shows that at least 11 types of security vulnerabilities can be specified with *fluent*TQL. These are the most popular security vulnerabilities for Java. Theoretically, one can express many more.

4.5 RQ4 Analyzing Java/Android Applications

To answer **RQ4** we ran *fluent*TQL queries on two Java applications, OWASP WebGoat and PetClinic, and seven Android applications from TaintBench (Luo et al. 2021).

The OWASP WebGoat is a deliberately insecure application aiming to teach developers about relevant security vulnerabilities. As a Java Spring application,[7] it is popular in the community and has been used for evaluating static analyses (Antoniadis et al. 2020). We used this application to evaluate the applicability of *fluent*TQL on real-world scenario, including specifying taint-flow queries and running our Boomerang-based and FlowDroid-based taint analysis.

We chose to work with the SQL injection as example since it has the most taint-flows in WebGoat. We documented all 17 SQL injection taint-flows in OWASP WebGoat and use

[6] https://github.com/OWASP/Benchmark/blob/master/src/main/java/org/owasp/benchmark/testcode/BenchmarkTest00001.java

[7] https://spring.io/

Table 5 Overview of the evaluated Java projects. Flows/B/F is number of expected taint-flows (vulnerability instances) and those found by Boomerang and FlowDroid, CWE is number of common weakness enumerations (vulnerability types), Runtime is average over ten runs

Project	#Classes	#Flows/B/F	#CWE	#Runtime(s) B/F
Catalog	36	27/27/25	11	52.8/43.7
PetClinic	42	4/4/4	1	10.9/14.4
WebGoat	35	17/17/13	1	30.3/36.7

them as ground truth. This was manually done by following the directions of the lessons present in WebGoat and inspecting the source code.

Next, we specified the sensitive methods which includes 17 sources, 1 sanitizer, 2 required propagators, and 2 sinks. We only needed to create two taint-flow queries to be able to cover all types of taint-flows. The Boomerang-based implementation was able to detect all 17 taint-flows. The official FlowDroid implementation (we used version 2.8) was not able to find any taint-flow in WebGoat. We investigated and found out that FlowDroid defines only the return values of the sources as taints. For all taint-flows in WebGoat, the taints are the parameters of the sources. Hence, we adapted FlowDroid to support this and after doing so, the FlowDroid implementation detected 13 taint-flows. Those that were missed are the types that contain a required propagator which is currently not supported by FlowDroid.

For the second Java application, PetClinic, we followed the same steps as for OWASP WebGoat. We identified and documented five taint-flows of type hibernate injection and two taint-flows of type cross-site request forgery. In this application, all taint-flows were detected by our implementation with Boomerang and our updated version of FlowDroid. Table 5 shows summary of the Java applications.

TaintBench is a collection of real-world Android apps that contain malicious behavior in form of taint-flows. These apps have well documented information about the expected taint-flows and should help analyses writers evaluate their tools in a rigorous and fair way. Table 6 summarizes the findings of running our *fluent*TQL implementation with Boomerang as well as with FlowDroid. Out of 25 expected taint-flows among all apps the Boomerang-based implementation found 18 whereas FlowDroid-based implementation found 13. We manually inspected those that were not found and identified two causes which are due to the existing solvers and not the inability of *fluent*TQL to express them. The first cause is the inability to analyze taint-flows through different threads in the code. Due to implicit data-flow behavior of the threads, the existing call graph algorithms have limitation in modeling this correctly. This second cause is that the existing data-flow analyses do not analyze the expressions within path constraints. In the case of our experiments, we found that the call of the source method is within the condition of an IF statement, which is not analyzed by Boomerang nor by FlowDroid.

The runtime values reported in Tables 5 and 6 are the average values over ten runs on a system with Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 16 GB RAM with Win-10 OS.

Detecting Taint-Flows Through Files The code in Listing 5 shows the *loadClass* method from the app *dsencrypt*[8] which contains the malicious taint-flow. It reads an encrypted zip file from asset folder (source in Line 54), decrypts it and extracts class.dex which contains

[8] https://github.com/TaintBench/dsencrypt_samp/blob/master/src/main/java/com/kbstar/kb/android/star/ProxyApp.java

Table 6 Overview of the evaluated Android apps from TaintBench. Flows/B/F is number of expected taint-flows (vulnerability instances) and those found by Boomerang and FlowDroid, Runtime is average over ten runs

App	#Classes	#Flows/B/F	#Runtime(s) B/F
blackfish	338	13/11/11	18.6/29.8
beita_com_beita_contact	379	3/1/1	11.2/25.5
phospy	236	2/2/0	8.6/11.5
repane	5	1/1/0	3/4.8
dsencrypt	4	1/1/1	10.2/4.9
fakeappstore	402	3/2/0	23/16.5
fakemart	868	2/0/0	27.3/34.9

malicious code (intermediate statements in the trace are lines 57, 58, and 59). The malicious code is called via reflection (sink in Line 67). As reported in the work by Luo et al. (2021), these kind of taint-flows going through files, databases, etc., can not be detected by the existing Android taint analysis tools. With *fluent*TQL, we are now able to model and detect these taint-flows using the and() operator.

> Our Boomerang-based implementation of *fluent*TQL is able to detect all expected taint-flows in the Java Spring applications: OWASP WebGoat and the PetClinic. Among seven real-world Android apps with malicious taint-flows, *fluent*TQL can detect 18 out of 25 expected taint-flows. Those that can not be detected are complex modeling of threads and not considering path conditions.

```
51  private void loadClass(Context context) {
52      ...
53      try {
54          InputStream is = getAssets().open("ds");
55          int len = is.available();
56          byte[] encrypeData = new byte[len];
57          is.read(encrypeData, 0, len);
58          byte[] rawdata = new DesUtils(
                DesUtils.STRING_KEY).decrypt(encrypeData);
59          FileOutputStream fos = new
                FileOutputStream(sourcePathName);
60          fos.write(rawdata);
61          fos.close();
62      } catch (Exception e) {
63          e.printStackTrace();
64      }
65      try {
66          Object[] argsObj = new Object[]{sourcePathName,
                outputPathName, Integer.valueOf(0)};
67          DexFile dx = (DexFile) Class.forName(
                "dalvik.system.DexFile").getMethod("loadDex", new
                Class[]{String.class, String.class,
                Integer.TYPE}).invoke(null, argsObj);
68          ...
69      } catch (Exception e2) {
70      }
71  }
```

Listing 5 Malisious taint-flow through a file in the *dsencrypt* app from TaintBench

4.6 Threats to Validity

We next discuss the most relevant threats to the validity of our study design and evaluation based on the threat types by Cook and Campbell (1979).

External Validity The participation in the study was voluntary. We asked our contacts in industry to invite their software developers. The invitation mentioned that the study would try to compare two domain-specific languages for static analysis. Having this information, it is more likely that the participants have some interest in the design of programming languages and/or static analysis. Hence, there is a threat of having a subject not representative for the entire population of software developers.

Internal Validity Apart from professional software developers, we invited researchers and master-level students from the university. Previous work has shown that graduate students are valid proxies for software developers in such studies (Naiakshina et al. 2017, 2018, 2019, 2020). Also, our results confirm that there is no significant correlation between the position of the participant and the performance in the task, thus also confirming that—for the purpose of such studies—researchers and master-level students have coding knowledge comparable to professional developers.

Moreover, the format of within-subjects study design has its own limitations. As both tasks were the same, but for a different DSL and context (vulnerability example), when solving the second task, participants may have be influenced by the first task, known as carryover effects. To deal with this we applied randomization of the order of the tasks.

Construct Validity Another threat to validity is the fairness of the tasks. Both DSLs are not equally expressible. This means one may need more or less time to learn a new DSL. To address this, we took into consideration the following points. First, we used vulnerabilities that have the same taint-flow pattern. The Java code shown as an example for each task had the same complexity. Second, for each task, we provided a stub code for the solution. In the case of CodeQL, which is more expressible DSL than *fluent*TQL, and has support not only for taint analysis but other analyses too, we asked the participants to focus only on the taint analysis module. Finally, we had three test sessions to adjust the tasks and define what exact information the participants will need to be able to solve each task in under ten minutes. Similarly, a possible threat to validity comes from the design of our study to use the open redirect vulnerability with *fluent*TQL and XSS for CodeQL for all participants without switching among half of the participants. To mitigate the threat, we have selected the code examples used in the tasks to have the same structure, i.e. the taint was in both cases created by a call to an HTTP request object and only the sink method differs for each vulnerability. Additionally, while explaining each task, we also explained the vulnerability. While the participants were performing the task, we encouraged them to verbally share their thoughts. After processing the recorded material, we find none of the participants to struggle with understanding the vulnerability itself.

5 Related Work

With few exceptions, such as DroidSafe (Gordon et al. 2015), which has hard-coded SM, the SM of the existing static analyses can be customized by the user, to some extend. In this section, we discuss the related approaches summarized in Table 7 that shows design principals of each approach and level of fulfillment to the requirements from Section 2.

5.1 Graph-Based Approaches

We group DSLs in this category that allow users to explicitly manipulate graph structures to specify code patterns.

CxQL, the DSL of the tool Checkmarx (2020), is general-purpose-like and object-oriented language. It supports a wide range of SM (**R1**). The flow propagation is done implicitly via the graph patterns (**R2**). As a commercial tool it provides well integrated workflow for the users (**R4** and **R5**). Since CxQL is capable of expressing a broad range of graph properties, it is hard to integrate it to a generic taint analysis (**R6**) as it is bound to the tool's core analysis.

CodeQL has good integration in the developers' workflow (**R5**) with plugins for popular IDEs as well as web-based interface. It is a declarative language with support for predicates and object-oriented design. The DSL can express a wide range of properties similar to CxQL.

PIDGIN (Johnson et al. 2015) follows the object-oriented design. It is not designed for taint analysis, therefore it is hard to integrate it in a generic way (**R6**). PIDGIN does not provide tool integrations for software developers.

IncA (Szabó et al. 2016) is a DSL for specification of the rules for incremental execution of static analyses. Compared to the other DSL in this category it is the least expressive and targets very specific domain. The SM and flow propagation can be expressed through the graph patterns (**R1**, **R2**). User-defined messages are not supported (**R4**).

5.2 Typestate Approaches

This category consists of two DSLs, i.e. CrySL and PQL, which are designed for typestate analysis, that detects the incorrect API usage.

Table 7 List of related approaches, their design characteristic, and their support of the requirements in Section 2

	Approach	declarative	general-like	object-oriented	constraints	pattern-based	annotations	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**
Graph-based	CxQL (Checkmarx, 2020)		X	X				●	◓	◓	●	●	○
	CodeQL (Github, 2020)	X	X	X				●	◓	◓	●	●	○
	PIDGIN (Johnson et al., 2015)			X	X			●	●	◓	○	○	○
	IncA (Szabó et al., 2016)				X	X		◓	◓	○	○	◓	○
Typestate	CrySL (Krüger et al., 2019)	X				X		●	●	◓	○	◓	○
	PQL (Martin et al., 2005)	X			X			●	●	◓	○	○	○
Other	CheckersF. (Dietl et al., 2011)						X	◓	●	○	●	○	○
	Apposcopy (Feng et al., 2014)	X					X	◓	●	◓	○	○	◓
	Athena (Le and Soffa, 2011)	X			X	X		●	●	○	○	○	○
	AQL (Pauck et al., 2018)				X	X		◓	●	○	○	●	○
	WAFL (Sridharan et al., 2011)		X	X				◓	◓	◓	○	○	○
	Saluki (Gotovchits et al., 2018)	X			X	X		◓	●	◓	○	○	○
	*fluent*TQL					X		●	●	●	●	●	●

●—fullfilled, ◓—partially fullfilled, ○—not fullfilled

CrySL (Krüger et al. 2019) enables cryptography experts to specify the correct usage of the crypto API making it not suitable for generic taint analysis (**R6**). The DSL is declarative with mechanism based on predicates and constraints. It has a full support for SM and flow propagation (**R1**, **R2**). The tool support is maintained.

PQL (Martin et al. 2005) is declarative DSL with specifications comparable to CrySL. Compared to *fluent*TQL, the PQL's syntax significantly differs from regular Java syntax making it difficult to use for non-experts. PQL does not ship with available tooling support for the users (**R6**).

5.3 Other Approaches

The approach used in CheckersFramework (Dietl et al. 2011) is based on the annotations *@tainted* and *@untainted* which developers can use to annotate their code to mark custom SM. The annotation specification requires additional manual work, which - in the case of legacy code - is even infeasible (**R5**). The CheckersFramework allows to configure the messages that are reported (**R4**).

Apposcopy (Feng et al. 2014) is a Android-specific taint analysis (**R6**) with a Datalog-based DSL for data-flow and control-flow predicates. The sources, sinks, and propagators are specified in form of annotations (**R1**, **R2**). We were not able to find tooling support (**R5**) for the language.

Athena (Le and Soffa 2011) is a declarative DSL based on patterns and constraints with explicit support for SM and flow propagations (**R1**, **R2**). Athena does not support user-defined messages and is tightly coupled to a generator of analysis configurations.

AQL (Pauck et al. 2018) is an Android-specific (**R6**) querying language for taint flow results from different taint analysis tools. It supports specifications of sources and sinks (**R1**) and flow propagations (**R2**) and provides workflow with tool support for developers to query taint results from multiple tools (**R5**).

WAFL is the DSL of the F4F approach (Sridharan et al. 2011) and is a general-purpose-like language with object-oriented design that allows the specification of reflective behavior in frameworks so that static analyses can propagate the flow. WAFL has partial support for SM and flow propagation (**R1**, **R2**). Its main purpose is modeling of frameworks.

Finally, Saluki (Gotovchits et al. 2018) is a declarative DSL where method patterns can be specified. SM and flow propagation are supported (**R1**, **R2**), but complex flows not (**R3**).

6 Conclusion

Static and dynamic taint analyses can detect many popular vulnerability types. Using them during development time can reduce the costs. To use taint analysis tools efficiently, software developers need to configure them to their contexts.

We proposed *fluent*TQL, a new domain-specific language for taint analysis designed to be used by software developers. *fluent*TQL is able to express many taint-style vulnerability types. It supports single-step as well as multi-step taint-flow queries. Moreover, taint-flow queries can be combined with *and* operator to express parallel taint-flows. *fluent*TQL uses Java fluent interface as a concrete syntax. Its semantics is independent of a concrete implementation of taint analysis, making it easy for integration into existing tools.

In a comparative, within-subjects user study, *fluent*TQL showed to be more usable for software developers than CodeQL, a state-of-the-art DSL of the commercial tool LGTM.

Participants were faster in solving the task of specifying a taint-flow query for given vulnerability type in *fluent*TQL than in CodeQL.

In future, we plan to make *fluent*TQL more expressive. With *fluent*TQL, only method calls can be modeled as sources and sinks. We will add support for modeling specific variables, such as constant values for detecting hardcoded credentials and nullable variables for detecting null pointer dereferences. Moreover, we will add support for regular expressions to ease the specification of methods in case of method overloading in Java. Additionally, we plan on investigating the usefulness of incorporating the concept of entry point as part of *fluent*TQL. Finally, we are working on further evaluations of the applicability of *fluent*TQL in other real-world large-scale Java applications and creating new queries for detecting more security vulnerabilities.

Acknowledgements We gratefully acknowlede the funding by the project "AppSecure.nrw - Security-by-Design of Java-based Applications" of the European Regional Development Fund (ERDF-0801379). We thank Ranjith Krishnamurthy and Abdul Rehman Tareen for their contribution to the implementation of the MagpieBridge server.

Author Contributions The first author is the main contributor to this research. The second and fourth author contributed with conceptual ideas and feedback. The third author contributed to the implementation and with ideas for the concrete syntax of fluentTQL.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability https://fluenttql.github.io/

Code Availability https://github.com/secure-software-engineering/secucheck

Declarations

Ethics Approval The user study design has been approved for ethical correctness by one of the companies participated in the study as well as by the corresponding head of department at Fraunhofer IEM.

Consent to Participate and Publication For the user study described in section 4, all 26 participants signed a written consent form in which they agreed to participate voluntarily in the study. They also agreed that the collected data can be used for research publication. The written consent form was obtained from all participants before the study.

Conflicts of Interest/Competing Interests Not applicable

References

Antoniadis A, Filippakis N, Krishnan P, Ramesh R, Allen N, Smaragdakis Y (2020) Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation, PLDI 2020. ACM, New York, pp 794–807. https://doi.org/10.1145/3385412.3386026

Arzt S, Rasthofer S, Bodden E (2013) Susi: a tool for the fully automated classification and categorization of android sources and sinks. In: Network and distributed system security symposium 2013, NDSS'13

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon YL, Octeau D, McDaniel P (2014) Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation, PLDI '14. ACM, New York, pp 259–269

Bodden E (2018) The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). In: ACM SIGPLAN International workshop on the state of the art in java program analysis (SOAP 2018), ISSTA '18. ACM, New York, pp 85–93

Brooke J (2013) Sus: a retrospective. J Usability Stud 8(2):29–40

Checkmarx (2020) Checkmarx. https://www.checkmarx.com/, online; accessed January 2021

Chibotaru V, Bichsel B, Raychev V, Vechev M (2019) Scalable taint specification inference with big code. In: Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation, PLDI 2019. ACM, New York, pp 760–774

Christakis M, Bird C (2016) What developers want and need from program analysis: an empirical study. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016. ACM, New York, pp 332–343

Cook TD, Campbell DT (1979) Quasi-experimentation: design and analysis issues for field settings. Houghton Mifflin, Boston

Dietl W, Dietzel S, Ernst MD, Muşlu K, Schiller TW (2011) Building and using pluggable type-checkers. In: Proceedings of the 33rd international conference on software engineering, ICSE11. ACM, New York, pp 681–690

Feng Y, Anand S, Dillig I, Aiken A (2014) Apposcopy: semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, FSE 2014. ACM, New York, pp 576–587

Girden ER (1992) ANOVA: repeated measures. 84, Sage

Github S (2020) Lgtm. http://lgtm.com/, online; accessed January 2021

Gordon MI, Kim D, Perkins JH, Gilham L, Nguyen N, Rinard MC (2015) Information flow analysis of android applications in droidsafe. In: 22nd Annual network and distributed system security symposium, NDSS 2015, San Diego, California, USA, February 8–11, 2015. The Internet Society

Gotovchits I, van Tonder R, Brumley D (2018) Saluki: finding taint-style vulnerabilities with static property checking. In: Proceedings of the NDSS Workshop on Binary Analysis Research

Grammatech (2020) Codesonar. https://www.grammatech.com/products/codesonar, online; accessed January 2021

Grech N, Fourtounis G, Francalanza A, Smaragdakis Y (2018) Shooting from the heap: ultra-scalable static analysis with heap snapshots. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2018. ACM, New York, pp 198–208

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs? In: Proceedings of the international conference on software engineering, ICSE '13. IEEE Press, Piscataway, pp 672–681

Johnson A, Waye L, Moore S, Chong S (2015) Exploring and enforcing security guarantees via program dependence graphs. SIGPLAN Not 50(6):291–302

Krüger S, Späth J, Ali K, Bodden E, Mezini M (2019) Crysl: an extensible approach to validating the correct usage of cryptographic apis. IEEE Trans Softw Eng

Le W, Soffa ML (2011) Generating analyses for detecting faults in path segments. In: Proceedings of the 2011 international symposium on software testing and analysis, ISSTA11. ACM, New York, pp 320–330

Livshits B (2012) Dynamic taint tracking in managed runtimes. Tech. rep., Microsoft Research

Livshits B, Nori AV, Rajamani SK, Banerjee A (2009) Merlin: specification inference for explicit information flow problems. SIGPLAN Not 44(6):75–86

Luo L, Dolby J, Bodden E (2019) Magpiebridge: a general approach to integrating static analyses into IDEs and editors (tool insights paper). In: Donaldson AF (ed) 33rd European conference on object-oriented programming (ECOOP 2019), Schloss Dagstuhl–Leibniz-Zentrum fuer informatik, Dagstuhl, Germany, vol 134, pp 21:1–21:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.21

Luo L, Pauck F, Piskachev G, Benz M, Pashchenko I, Mory M, Bodden E, Hermann B, Massacci F (2021) Taintbench: automatic real-world malware benchmarking of android taint analyses. Empir Softw Eng

Martin M, Livshits B, Lam MS (2005) Finding application errors and security flaws using pql: a program query language. SIGPLAN Not 40(10):365–383

Microfocus (2020) Fortify. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview, online; accessed January 2021

Microsoft (2020) Language server protocol. https://microsoft.github.io/language-server-protocol/, online; accessed January 2021

Mitre CWE (2020a) 2011 cwe/sans top 25 most dangerous software errors. http://cwe.mitre.org/top25/, online; accessed January 2021

Mitre CWE (2020b) Cwe home page. http://cwe.mitre.org/, online; accessed January 2021

Mitre CWE (2020c) Improper neutralization of data within xpath expressions. https://cwe.mitre.org/data/definitions/643.html, online; accessed January 2021

Mitre CWE (2020d) Improper neutralization of input during web page generation. https://cwe.mitre.org/data/definitions/79.html, online; accessed January 2021

Mitre CWE (2020e) Improper neutralization of special elements in data query logic. https://cwe.mitre.org/data/definitions/943.html, online; accessed January 2021

Mitre CWE (2020f) Improper neutralization of special elements used in a command. https://cwe.mitre.org/data/definitions/77.html, online; accessed January 2021

Mitre CWE (2020g) Improper neutralization of special elements used in an ldap query. https://cwe.mitre.org/data/definitions/90.html, online; accessed January 2021

Mitre CWE (2020h) Improper neutralization of special elements used in an sql command. https://cwe.mitre.org/data/definitions/89.html, online; accessed January 2021

Mitre CWE (2020i) Improper output neutralization for logs. https://cwe.mitre.org/data/definitions/117.html, online; accessed January 2021

Mitre CWE (2020j) Relative path traversal. https://cwe.mitre.org/data/definitions/23.html, online; accessed January 2021

Mitre CWE (2020k) Trust boundary violation. https://cwe.mitre.org/data/definitions/501.html, online; accessed January 2021

Mitre CWE (2020l) Url redirection to untrusted site (open redirect). https://cwe.mitre.org/data/definitions/601.html, online; accessed January 2021

Mitre CWE (2020m) Xml injection. https://cwe.mitre.org/data/definitions/91.html, online; accessed January 2021

Naiakshina A, Danilova A, Tiefenau C, Herzog M, Dechand S, Smith M (2017) Why do developers get password storage wrong? A qualitative usability study. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, CCS 17. ACM, New York, pp 311–328. https://doi.org/10.1145/3133956.3134082

Naiakshina A, Danilova A, Tiefenau C, Smith M (2018) Deception task design in developer password studies: exploring a student sample. In: Proceedings of the fourteenth USENIX conference on usable privacy and security, USENIX Association, USA, SOUPS 18, pp 297–313

Naiakshina A, Danilova A, Gerlitz E, von Zezschwitz E, Smith M (2019) If you want, i can store the encrypted password: a password-storage field study with freelance developers. In: Proceedings of the conference on human factors in computing systems, CHI 19. ACM, New York, pp 1–12

Naiakshina A, Danilova A, Gerlitz E, Smith M (2020) On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers. In: Proceedings of the 2020 CHI conference on human factors in computing systems, CHI 20. ACM, New York, pp 1–13

Nguyen Quang Do L, Bodden E (2020) Explaining static analysis with rule graphs. IEEE Trans Softw Eng 1–1. https://doi.org/10.1109/TSE.2020.3004525

Nguyen Quang Do L, Wright JR, Ali K (2020) Why do software developers use static analysis tools? A user-centered study of developer needs and motivations. In: Proceedings of the sixteenth symposium on usable privacy and security. https://doi.org/10.1109/TSE.2020.3004525

OWASP (2020a) Owasp benchmark. https://owasp.org/www-project-benchmark/, online; accessed January 2021

OWASP OWASP (2020b) Owasp top 10 most critical web application security risks. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, online; accessed January 2021

Pauck F, Bodden E, Wehrheim H (2018) Do android taint analysis tools keep their promises? In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE 2018. ACM, New York, pp 331–341

Piskachev G, Do LNQ, Bodden E (2019) Codebase-adaptive detection of security-relevant methods. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2019. ACM, New York, pp 181–191

Reichheld FF (2003) The one number you need to grow. Harv Bus Rev 81(12):46–55

Sas D, Bessi M, Fontana FA (2018) Automatic detection of sources and sinks in arbitrary java libraries. In: 2018 IEEE 18th International working conference on source code analysis and manipulation (SCAM), pp 103–112

Schwartz EJ, Avgerinos T, Brumley D (2010) All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on security and privacy, pp 317–331. https://doi.org/10.1109/SP.2010.26

SecuCheck RP (2021) Interviews with developers. https://secucheck.github.io/, online; accessed January 2021

Smith J, Johnson B, Murphy-Hill E, Chu B, Lipford HR (2019) How developers diagnose potential security vulnerabilities with a static analysis tool. IEEE Trans Softw Eng 45(9):877–897

Smith J, Nguyen Quang Do L, Murphy-Hill E (2020) Why can't Johnny fix vulnerabilities: a usability evaluation of static analysis tools for security. In: Proceedings of the sixteenth symposium on usable privacy and security, SOUPS 2020

Song T, Li X, Feng Z, Xu G (2019) Inferring patterns for taint-style vulnerabilities with security patches. IEEE Access 7:52339–52349

Späth J, Ali K, Bodden E (2019) Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. Proceedings of the ACM SIGPLAN symposium on principles of programming languages 3(POPL):48:1–48:29

Sridharan M, Artzi S, Pistoia M, Guarnieri S, Tripp O, Berg R (2011) F4f: taint analysis of framework-based web applications. SIGPLAN Not 46(10):1053–1068

Stahl T, Voelter M, Czarnecki K (2006) Model-driven software development: technology, engineering, management. Wiley, Hoboken

Szabó T, Erdweg S, Voelter M (2016) Inca: a dsl for the definition of incremental program analyses. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016. ACM, New York, pp 320–331

Thomé J, Shar LK, Bianculli D, Briand LC (2017) Joanaudit: a tool for auditing common injection vulnerabilities. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017. ACM, New York, pp 1004–1008

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Goran Piskachev (M.Sc. 2015) is a team manager for development tools for secure services and apps at Fraunhofer IEM in Paderborn. He received his master degree in computer science and is currently a PhD student at Paderborn University. Previously, he completed an engineering degree at the Ss. Cyril and Methodius University in Skopje. His research interests include static code analysis, security testing, domain specific languages, and machine learning for code analysis.

Johannes Späth (PhD 2019) has a research background in static program analysis and completed his PhD at the University of Paderborn in 2019. His research interests lie in pointer analysis and taint analysis with applications for automated detection of security vulnerabilities. During his PhD he worked on CogniCrypt, an eclipse-based IDE plugin that supports developers in using cryptographic APIs in Java. Since 2020, Johannes is a co-founder of CodeShield, a startup that focuses on cloud application security.

Ingo Budde (B.Sc 2018) is a Software Engineer at Fraunhofer IEM in Paderborn. He received his B.Sc degree in Computer Science from Paderborn University (2018) and is interested in static code analysis.

Eric Bodden is a full professor for Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. He is further the director for Software Engineering and IT Security at the Fraunhofer Institute for Engineering Mechatronic Systems Design. Prof. Bodden has been recognized several times for his research on program analysis and software security, most notably with the German IT-Security Price and the Heinz Maier-Leibnitz Price of the German Research Foundation, as well as with several distinguished paper and distinguished reviewer awards. He is an ACM Distinguished Member.

Terms and Conditions