UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Master's Thesis

Submitted to the Secure Software Engineering Research Group
in Partial Fulfilment of the Requirements for the Degree of

Master of Science

Extending FluentTQL

Specifying taint flows through a DSL

by
Enri Ozuni

Thesis Supervisors:
Prof. Dr. Eric Bodden
Dr. Matthias Meyer

Paderborn, October 21, 2021

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Abstract. Domain-specific languages (DSLs) have become helpful in describing malicious taint flows and serve as a starting point for taint analyses. FluentTQL is one of these languages and is utilized to specify taint queries related to vulnerabilities such as SQL Injection or Cross-site Scripting. It comes together with a taint analysis tool such as SecuCheck, which analyzes a program to find the specified malicious flows. Our work includes enhancements to both the syntax of the language and the taint analysis implementation in the tool. Two of them have to do with easily specifying overloaded versions of methods that are part of a taint query and a second alternative of writing such method specifications. Another improvement is the ability to specify other types of vulnerabilities in FluentTQL, such as NULL Pointer Dereference or Hard-coded Credentials. The last enhancement is shortening the overall runtime of the analysis by manually specifying entry points in the language for given taint queries. Results show a clear improvement in the usability of the DSL and better performance of the static taint analysis.

Contents

1 Introduction **1**

2 Background **7**

3 Extensions **11**
3.1 Alternative option of writing a method signature 13
3.1.1 Implementation 15
3.2 Increasing the expressivity when specifying signatures 16
3.2.1 Implementation 18
3.3 Utilization of entry points for taint flow queries 19
3.3.1 Implementation 20
3.4 Ability to detect HCC and NPD vulnerabilities 21
3.4.1 Implementation 23
3.5 Addition of a backward taint analysis 24
3.5.1 Implementation 25

4 Evaluation **27**
4.1 User Study 27
4.1.1 Participants 28
4.1.2 Tasks 29
4.1.3 Questionnaires 34
4.1.4 Setup 36
4.1.5 Results 37
4.2 Experimental Approach 41
4.2.1 Specifications 42
4.2.2 Setup 45
4.2.3 Results 45
4.3 Sporadic Approach 48

5 Related Work **51**

6 Conclusion **53**

Bibliography **55**

1 Introduction

Nowadays, we have all seen the relevance of hygiene. Everyone has experienced how fast and effortlessly a virus can spread; a handshake or being nearby a coughing person are different ways to contact a virus. It is an infectious agent that cannot be spotted with the naked eye but has the power of disrupting our everyday routine and affecting the health of so many of us. Malicious or tainted data can be similar to a virus. They should always be sanitized before use because they can taint more data and lead to severe problems in software [Cam].

Taint checking or taint analysis can be used to automatically detect malicious data that are not sanitized and also flow from sources to sinks [ARF+14]. A source can be a method that returns sensitive information or accepts user input. Moreover, a sink can be a function that performs delicate or security-related operations, like storing data in a database. Also, a sanitizer can be a specific method that cleans malicious spots in a given data. This kind of static analysis identifies the above-mentioned flows of harmful data without needing to run the code. To properly work, a set of sources, sinks, and sanitizers must be detected or set. There are some ways of handling this operation, like the utilization of models that are trained with machine learning [RAB14] or the usage of domain-specific languages (DSL) [Ber07], with the latter being the focus of this thesis.

FluentTQL is a DSL for specifying taint flows and their respective parts: sources, sinks, sanitizers, and taint flow queries [Pis]. It is an internal DSL and uses Java as a host language. As Java is one of the most used programming languages by developers, it makes FluentTQL easy to understand and use since many developers are familiar with the syntax of Java. The language is designed to be used even by developers who do not fully understand how taint analysis works under the trunk. Our DSL also comes with an analysis tool called SecuCheck, which uses taint queries specified in FluentTQL to detect taint-related vulnerabilities [Pis]. It is built by applying the Soot [VRCG+10] and Boomerang [SNQDAB16] frameworks for its taint analysis implementation. SecuCheck on itself is built on top of MagpieBrige [LDB19], which is another framework for integrating static analyses into different IDEs and editors. FluentTQL and SecuCheck are currently used to run taint analyses and detect vulnerabilities in Java programs.

To better understand how FluentTQL works, we will demonstrate it through an example. The code in Listing 1.1 (adapted from [Bae]) shows a database-related operation that contains an SQL Injection vulnerability; this is one of the most dangerous vulnerabilities in both CWE's top 25 list [CWEa] and OWASP's top 10 list [OWA]. The method `updateAccount` is located in the `Account` class and is used to update an entry in a database by utilizing the given parameters of `name` and `id`, as seen in line 11. The `name` parameter is given as input from some user that wants their account name updated, whereas the `id` is processed automatically from the

user who makes such request. The `updateAccount` method can be considered a source since it can introduce tainted information in the form of `name`. Moreover, the possible tainted data is propagated in the `query` variable in lines 14-15. The `query` is then given as a parameter to the `executeUpdate` method in line 17. Such a method can be considered a sink since it performs security-related operations. In this example, the `name` parameter is not sanitized, and a malicious user could easily inject SQL commands such as entirely deleting a database.

```
1  package example;
2
3  // Imports
4
5  public class Account {
6
7      // Fields
8
9      // Methods
10
11     public void updateAccount(String name, String id)
12         throws SQLException
13     {
14         String query = "UPDATE people SET name='"+name
15                        +"' WHERE id = '"+ id + "'";
16         Connection c = dataSource.getConnection();
17         int update = c.createStatement().executeUpdate(query);
18     }
19
20     // Some more methods
21
22 }
```

Listing 1.1: Java example containing an SQL Injection vulnerability

The code in Listing 1.2 shows a specification in FluentTQL, which can be used as a query for the SecuCheck tool to find the SQL Injection vulnerability in Listing 1.1. The first three parts define the source (lines 3-7), the sanitizer (lines 9-13), the sink (lines 15-19); the last part defines the taint flow query (lines 23-26), which is constructed from the three above mentioned parts.

```
1  // Configurations
2
3  String sourceSignature = "example.Account: "
4                           +"void updateAccount"
5                           +"(java.lang.String)";
6  Method source = new MethodConfigurator(sourceSignature)
7                  .out().param(0);
8
9  String sanitizerSignature = "example.Account: "
10                          +"java.lang.String sanitize"
11                          +"(java.lang.String)";
12 Method sanitizer = new MethodConfigurator(sanitizerSignature)
13                 .in().param(0).out().return();
14
15 String sinkSignature = "example.Account: "
16                         +"int executeUpdate"
17                         +"(java.lang.String)";
18 Method sink = new MethodConfigurator(sinkSignature)
19                .in().param(0);
20
21 // Some more configurations
22
```

```
23 TaintFlowQuery taintflow = new TaintFlowQueryBuilder("SQL-Injection");
24 taintflow.from(source).notThrough(sanitizer).to(sink)
25                 .report("SQL injection vulnerability.").at(Location.SINK)
26                 .build();
```

Listing 1.2: Specification for SQL Injection vulnerability in Listing 1.1

The first three parts show that sources, sinks, sanitizers (and even propagators, which are not showcased in this example) are defined as methods by specifying their full method signature. The latter contains information on where the method is located, its return type, name, and input parameters. Furthermore, methods are configured to track the tainted information using keywords such as `in` or `out` which display that the tainted data is flowing inside or outside the method, respectively. The specific data in the methods are tracked by using keywords such as `param` or `return`. The last part of the FluentTQL specification shows how a taint query is built: `from` contains possible sources, `notThrough` specifies sanitizers, `to` accommodates sinks and `through` would contain propagators. Moreover, `report` is used to report a custom-written message in case vulnerabilities queried with our taint flow are found, and `at` defines where this error message will appear.

The whole information specified in FluentTQL, from methods to taint flow queries, is easily readable and writable. This clarity is achieved by using the builder pattern [Nel] because it allows us to write understandable code to initialize objects with many fields, as is the case with `Method` and `TaintFlowQuery` in Listing 1.2. This pattern aids the implementation of FluentTQL as an internal DSL and utilizes Java as a host language. The DSL is meant to be straightforward to use. The sources, sinks, and sanitizers can be described at the granularity of variables since they are the ones that store the tainted data. Also, one can write custom error messages for given taint flow specifications and where such messages should be shown in the source code. Besides its many features, FluentTQL and its static analysis tool SecuCheck can have some shortcomings, which are the motivation for the work of this thesis.

Firstly, the flow participants, such as is the case with sources, sinks, sanitizers, and propagators, need always to be written in a `String` format. Users have to write the full signature of the method by specifying the class where the method is present, its return type, name, and parameters. Also, this information needs to be in that exact order, and users should not forget characters such as colon (`:`) or round brackets. The colon (`:`) must be used after finishing the specification of the class where the method is located, and round brackets have to be used for writing the method parameters. If any of these symbols are missing in the signature, the static taint analysis will not match the signature even if the method is located in the source code, and thus no taint flow will be reported. For this reason, we have extended the DSL through another element called `MethodSignature` that uses the same builder pattern for initializing the signature of a method. The `MethodSignature` element can better guide inexperienced users of our DSL when writing the building blocks of a signature and does not need from them the extra symbols, which evaluation results show that they are quickly forgotten during the specification process. Both signature objects can be used interchangeably in the language, and a user study was conducted regarding their usability. Results show that the new feature achieved a System Usability Scale (SUS) [B+96] score of 85,8 as opposed to that of 70,4 of the old feature. A score above 68 is considered above average, showing the high usability of both language elements. The new feature also achieved a better Net Promoter (NP) [Rei03] score than the older feature.

Moreover, let us consider the code in Listing 1.1. In line 17, we have the `executeUpdate` method which does an update operation in the database by utilizing the `query` in lines 14-15. As was demonstrated in Listing 1.2, this function is considered a sink in the FluentTQL specification. The `executeUpdate` method can be overloaded and used with different settings

of parameters as described in the Javadoc of `Statement` interface [Ora]. If many overloaded versions of the `executeUpdate` method were used in our program, then we would also have to specify all signatures of these methods in detail in FluentTQL. This makes it tedious and increases the risk of typos when writing each overloaded method's different parameters. For this reason, we have extended the language and made the specification process easier in such scenarios. If users would want to specify all overloaded versions of a method in our DSL, they can write the keyword `ANY` that represents any possible arguments that a method can take. The taint analysis will recognize the usage of this keyword and continue the operation like usual by finding all usages of the `executeUpdate` method, regardless of their parameters. Like the previous feature, a usability evaluation was conducted by comparing both the old and new alternatives. Results show that our extension not only made the specification process 72 percent faster but reached excellent usability metrics. The new feature achieved a SUS score of 94,6 instead of 61,3 of the old feature and a perfect NP score of 100 percent.

Furthermore, the code excerpt in Listing 1.1 is only used to demonstrate the vulnerability in it. Usual programs and applications are often made out of tens of thousands of lines of code. In order to find a taint-based vulnerability in a large program, the whole program needs to be queried with the taint flows specified in FluentTQL. The old analysis of SecuCheck used to list all public methods of a given source code and consider all of them as entry points for the taint analysis. It used to query the application code from one entry point to the next and displayed any found results from the analysis. Analyzing the whole source code in large applications is usually expensive and requires considerable time and computer resources. Also, vulnerabilities are usually found in parts of code that handle user input, and having the possibility only to analyze those parts would make the analysis more efficient. We have extended the language and analysis for the above reason by allowing the user to specify entry points related to given taint flow queries. In this manner, users can direct the taint analysis in only those parts of code that handle user input or where vulnerabilities might be present. Knowledgeable developers and security experts can use the entry point feature in taint analysis. An evaluation was performed by specifying entry points in eight different FluentTQL specifications related to famous vulnerabilities on a demo To-Do List application. Experimental results show that the analysis was, in general, 43 percent faster when entry points were used in the specifications. Usage of entry points was also done for the best scenario by only specifying the entry points, which would lead to faster detection of the vulnerabilities.

As the specification in Listing 1.2 demonstrates, FluentTQL helps define taint queries related to injection-style vulnerabilities like SQL Injection (SQL-I) or Cross-site Scripting (XSS). In such vulnerabilities, the dangerous data usually comes from outside the program, as it happens with user-supplied data. Then, we need to track these tainted data from a source method until they reach another method that performs sensitive operations. However, there are other vulnerabilities in *CWE's top 25 list* [CWEa] like the use of Hard-coded Credentials (HCC) or NULL Pointer Dereference (NPD), where the weakness starts from a variable inside the program due to implementation faults by the developers. The old version of FluentTQL only modeled sources and sinks to the granularity of methods and, therefore, could not specify a variable as being a source, as it happens with HCC and NPD vulnerabilities. Our language extension allows users to specify the sources related to such vulnerabilities as enumerations. These enumerations are used to hint at either null or hardcoded variables, a piece of information used by the SecuCheck tool to employ a custom-built specific taint analysis for finding the HCC and NPD vulnerabilities. The custom-built analysis works backward instead of the forward direction, which is the default direction in the taint analysis. It starts from the specified sink methods and up to the null or hardcoded variables that it can find.

A final addition to the SecuCheck tool is a complete and independent backward taint analysis that works for all types of specifications, not only for HCC- and NPD-related ones. Previously, the taint analysis was only implemented forward, where it would initially look for specified sources in the source code. When only finding any, it would then start a demand-driven analysis for finding the other specified parts of the taint flow query, such as propagators, sanitizers, and sinks. This strategy would lead to poor performance when there were many specified sources and only one sink; if the specified sink would not be present in the code, then the forward taint analysis would waste time and query the whole program in search of the sink. For scenarios such as above, a backward taint analysis would be more efficient because it would first search for the specified sinks, and when not finding any, it would simply stop and finish promptly. At present, the taint analysis in SecuCheck can be started in both directions, depending on the number of specified sources and sinks. The direction is chosen automatically in the SecuCheck tool and does not require assistance from the FluentTQL users.

The following chapters contain the background, extensions, evaluation, related work, and conclusion of this master thesis. Chapter 2 specifies terms and concepts that are utilized to understand the work and research of this thesis. Then, Chapter 3 explains in detail all extensions that are performed to FluentTQL and SecuCheck, such as their usage scenarios and implementation details. Next, Chapter 4 of evaluation contains details on how the extensions were evaluated, the obtained results, and the research questions that were answered from this work. Furthermore, Chapter 5 includes relevant research to our work and how they relate to our extensions. Lastly, Chapter 6 concludes this master thesis with the main points that can be taken away and ideas on future research.

2 Background

This chapter contains concepts, notions, and tools that are of relevance for the work of this thesis. One may have information on many terms, but they are explained in the context of the FluentTQL and SecuCheck tools.

Domain-Specific Language: A domain-specific language (DSL) [Fow10] is a computer language that is targeted to a particular kind of problem, in comparison to a general-purpose language, like Java, that is aimed at any software problem. FluentTQL is a DSL that is specifically used to specify taint flows for common software vulnerabilities. An important and valuable distinction to make is between external and internal DSLs. External DSLs have their custom syntax, and one has to write a complete parser to process them. For example, these types of DSLs can be built by using the XML language. On the other hand, Internal DSLs are particular ways of using a host language to give the host language the feel of another language. Internal DSLs are also referred to as embedded DSLs or Fluent Interfaces [Fow]. FluentTQL is an internal DSL that uses Java as a host language to initialize more readable objects via the builder pattern. Initialized objects are taint flow participants like sources and sinks or taint flow queries that are given as input to the taint analysis in SecuCheck.

Taint Analysis: A variable that contains sensitive information is called a tainted variable, otherwise untainted variable. *Taint analysis* is a static analysis that tracks the flow of tainted data from a source to a sink. A source may be a method that returns or contains sensitive information in the form of variables; methods like `getPassword` can be a source. A sink may be a method that performs sensitive operations with variables; examples of sink methods can be `executeQuery` or `sendEmail`. Other notable parts in this analysis are propagators and sanitizers. A propagator can be a method that propagates sensitive information from one variable to another variable; methods like `concatenate` `append` may be a propagator. Lastly, a sanitizer may be a method that removes sensitive information from a tainted variable; such methods can have names like `encrypt` or `sanitize`. SecuCheck is a tool that performs taint analysis by being initialized by sources, sinks, propagators, and sanitizers which we need to find as participants in a taint flow when statically analyzing a program.

Builder Pattern: The most common form of object construction is through the use of constructors and their parameters. One can even construct an object with a constructor without arguments and then use setter methods to change the object's state. In classes that contain

many fields, object creation can become a little messy because one may need to initialize all of its fields, which might affect the code's readability. The builder pattern [FRBS08] is a design pattern that provides a different solution when constructing complex objects in object-oriented programming. In line 24 of Listing 1.2, we use this pattern to perform the initialization of the `TaintFlowQuery` object. The construction is done by method calls such as `from`, `to`, `at` and so on, which are setter methods that return a object in order to continue the fluent operation via method chaining. Only the last method of `build` is used in this pattern to return the initialized object which in this case is `TaintFlowQuery`.

One of the benefits of this pattern is that it makes the initialization process more readable than the classic instantiation process. This is quite useful as FluentTQL contains objects such as `Method` or `TaintFlowQuery` which have many fields that need to be initialized during creation, and a user would read with ease the state of objects in a FluentTQL specification. The method calls in these objects are not very descriptive but only show their full strength in this fluent action and help the user of FluentTQL keep track of the fields that need to be typed in their respective methods. Moreover, method completion in an IDE that supports Java, helps the user of FluentTQL in what to type next; this is necessary for ensuring that every FluentTQL object is fully instantiated as the taint analysis later uses it. If the classic initialization process would instead be used, a user could use the constructor without arguments and then set the fields using the setter methods; this would lead to high chances of forgetting to call the appropriate setter methods.

Entry Point: An entry point in the context of static analysis is a method that the analysis starts to analyze. An analysis can continue its operation in more than one entry point, such as is the case with the analysis in SecuCheck. This concept makes the root of one of the extensions of this thesis, where users will be able to normally set entry points in the analysis through FluentTQL.

Soot: Soot [VRCG+10] is a framework that is primarily used to build static analyses for Java and Android applications and then perform analyses on such applications. The way how it works is that it receives a Java or Android program that will be statically analyzed and transforms its code into a bytecode representation. Soot supports four different types of bytecode by default, where Jimple [VRH98] is the most prominent one. Then, it employs a chosen static analysis on the transformed bytecode. Most static analyses are built on the Jimple level, meaning that they only work on Jimple bytecode. Soot also features other functionalities like call-graph construction through the use of many algorithms like CHA [DGC95] or Spark [Lho02], points-to analysis, taint analysis through the usage of Boomerang, and many more. Our taint analysis in SecuCheck heavily utilizes the Soot framework to match specified flow participants with the statements in the analyzed Java programs.

Boomerang: Boomerang [SNQDAB16] is another framework that is used to implement the taint analysis in SecuCheck. It performs a highly efficient and precise demand-driven pointer analysis that computes such information only where required. This framework determined points-to and alias information on demand, with the latter being quite beneficial for the analysis implemented in SecuCheck. Our taint analysis uses Boomerang to keep track of all aliases of a tainted variable specified through FluentTQL. In cases where our tainted variables are used, it then utilizes the custom-built analysis that checks whether such variables are part of flow participants in given FluentTQL specifications. Our custom-built analysis also uses Boomerang, besides Soot, for many of its check-up operations.

MagpieBridge: MagpieBridge [LDB19] is a framework for integrating static analyses into an integrated development environment (IDE) or editor by making use of the Language Server Protocol (LSP). MagpieBridge serves as a bridge between the taint analysis in SecuCheck and many developer tools. The SecuCheck analysis implemented through the MagpieBridge framework produces analysis results such as messages, source code positions, or other types of information that can be converted into LSP messages and be interpreted from an editor or IDE that supports the LSP. Using this framework, taint analysis writers do not need to think about the editor or IDE for which they are implementing the analysis and their respective user interface features, but rather only about the analysis they are developing.

Language Server Protocol: The Language Server Protocol [Mic] is a protocol that is used between editors/IDEs and servers that provide programming language-specific features. This protocol allows programming language support to be implemented and distributed independently on any given editor or IDE. LSP decouples language services from the editor so that the services may be contained within a general-purpose language server. MagpieBridge is built by using this protocol, and the SecuCheck analysis that is implemented through MagpieBrige serves as a service that can be consumed by every development tool that has LSP installed in it. LSP is not restricted to programming languages. It can be used for any text-based language, like general-purpose or domain-based, as is the case with FluentTQL.

System Usability Scale: The System Usability Scale (SUS) [B$^+$96] is a simple tool that is utilized for measuring usability. It consists of a questionnaire that contains ten statements, and each of them can be answered by participants through five possible options, ranging from *Strongly Disagree* to *Strongly Agree*. This tool can evaluate the usability of different products and services. We have used it to evaluate the usability of two of our extensions in the FluentTQL language. A typical SUS questionnaire contains the following items that are listed below:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The odd-numbered items must always contain positive statements about the usability of the system or service, whereas the even-numbered items must contain negative declarations. The above list is just a template, and the ten items can be constructed about the system or service we are evaluating. Upon completion of a questionnaire with alternatives ranging from 1 (Strongly Disagree) to 5 (Strongly Agree), we can calculate the SUS score in the following manner:

- For each odd-numbered question, we subtract one from the answer (ranging from 1 to 5).
- For each of the even-numbered questions, subtract their answer (ranging from 1 to 5) from 5.
- We take the calculated values for all of the ten answers and add them up together.
- Lastly, we multiply the final results by 2,5 and get the SUS score.

The SUS score ranges from 0 to 100 but should not be interpreted as a percentage. A score above 68 is considered above average, whereas anything below 68 is considered below average in terms of usability.

Net Promoter: The Net Promoter (NP) [Rei03] score is used to measure user experience regarding a given product or service. We use the NP score to evaluate how much a user of FluentTQL would recommend two of our extensions. The usability score consists of a simple and important question that goes:

How likely are you to recommend [product/service] to a friend or colleague?

This question can be answered on a scale from 0 to 10. The question can be formulated differently depending on the product or service that one might want to evaluate; the key query of finding how much a product or service is promoted should still be there. Participants who answer the question are grouped as follows:

- **Detractors:** those who answer between 0 and 6 are the ones who do not like and do not further recommend the product or service.
- **Passives:** those who answer between 7 and 8 are pleased but unenthusiastic participants who are vulnerable to other alternatives.
- **Promoters:** those who answer between 9 and 10 are the ones who would recommend the product or service to others.

To calculate the NP score for a given product or service, we subtract the percentage of Detractors from the percentage of Promoters, as shown in the formula below:

NP score = Total % of Promoters – Total % of Detractors

The NP score can range from -100, which means that every customer is a Detractor, to 100, which means that every customer is a Promoter.

3 Extensions

This chapter includes information on all extensions that were done to FluentTQL and SecuCheck, which were shortly covered in Chapter 1. Each upcoming section will cover one extension and contain details on the respective extension, the changes introduced to the language and analysis, and implementation details. Below we have listed the remote repositories in where we have conducted the implementations for our extensions:

- **secucheck**
- **secucheck-core**
- **secucheck-catalog**

The **secucheck** repository contains the taint analysis of SecuCheck and the internal DSL of FluentTQL. Both elements work together to perform taint analyses in Java programs, which are started through FluentTQL specifications. The analysis also utilizes the MagpieBridge framework, which is used to run the SecuCheck tool in multiple IDEs, as was shortly covered in the Background chapter at Chapter 2. This repository is a Maven project which holds many other Maven modules. However, the most important one for our extensions is the *InternalFluentTQL* module which holds the implementation of the FluentTQL DSL. The **secucheck** repository can be found under the following link.

Moreover, the **secucheck-core** repository contains the core implementation of the taint analysis, hence the name *secucheck-core*. This repository is used in the main **secucheck** repository when implementing the SecuCheck analysis through the MagpieBridge framework. The core implementation of the taint analysis is kept in a separate repository to allow for better maintainability and usability as a dependency, in case another tool would want to use the Application Programming Interfaces (APIs) that the repository offers for performing taint analyses. Other details about this repository can be found under the following link.

Lastly, the **secucheck-catalog** repository holds demo projects and excerpts that contain well-known taint-related vulnerabilities, together with respective specifications in FluentTQL, which demonstrate such vulnerabilities. The specifications can be given as input to the SecuCheck tool. The latter runs its taint analysis to demonstrate the usage of the specifications and the performance of the analysis in finding the specified taint flows in the analyzed demo programs. More information about the repository can be found under the following link. All three of these repositories will be mentioned in the following sections when explaining the locations where we performed the extensions when covering essential implementation details.

```java
package account;

// Imports

public class Account {

    // Fields
    // Methods

    // SOURCE
    public void updateAccount(String id)
        throws SQLException
    {
        String query = "UPDATE people SET name='John' "
                       +"WHERE id = '"+ id + "'";
        Connection c = dataSource.getConnection();
        // SINK
        int update = c.createStatement().executeUpdate(query);
    }
    // Some more methods
}
```

Listing 3.1: Java example containing an SQL Injection vulnerability

```
// SOURCE
String sourceSignature = "account.Account: "
                         +"void updateAccount"
                         +"(java.lang.String)";
Method source = new MethodConfigurator(sourceSignature)
                .out().param(0);

// SANITIZER
String sanitizerSignature = "account.Account: "
                         +"java.lang.String sanitize"
                         +"(java.lang.String)";
Method sanitizer = new MethodConfigurator(sanitizerSignature)
                .in().param(0).out().return();

// SINK
String sinkSignature = "account.Account: "
                         +"int executeUpdate"
                         +"(java.lang.String)";
Method sink = new MethodConfigurator(sinkSignature)
                .in().param(0);

// TAINT FLOW QUERY
TaintFlowQuery taintflow = new TaintFlowQueryBuilder("SQL-Injection");
taintflow.from(source).notThrough(sanitizer).to(sink)
                .report("SQL injection vulnerability.").at(Location.SINK)
                .build();
```

Listing 3.2: Specification for SQL Injection vulnerability in Listing 3.1

Listing 3.1 shows a code excerpt that contains an SQL Injection vulnerability in it, similar to the one in Listing 1.1. The vulnerability is manifested from the source method of `updateAccount` in line 11, up to the sink method of `executeUpdate` in line 18. Next, Listing 3.2 shows a FluentTQL specification that can be used from the taint analysis to find the vulnerability in the above code. The specification uses the old features of the DSL before our extensions were implemented. These listings will be used as a reference when explaining the extensions in the next sections.

3.1 Alternative option of writing a method signature

This section covers details on the alternative option we have implemented in FluentTQL, which can be used to specify method signatures. Initially, we explain the structure and drawbacks of the old alternative, which motivated the implementation of our first extension.

In Listing 3.2 we see how signatures of flow participants like source, sink, and sanitizers are specified in the language through the usage of the `String` object. Let us closely examine the structure of the signature of the source method in the above listing, which is shown in Listing 3.3. We initially write the class where the respective method of `updateAccount` is located; in this case, it is `account.Account` and should include the whole namespace of `Account` class. Next, we insert the `:` symbol that indicates the completion of the writing of the class. Moreover, we should insert a space and then the return type of the method, which in this case is `void`. Furthermore, we insert another space and write the name of the `updateAccount` method. Lastly, we insert all argument types between parentheses as follows: `(java.lang.String)`. The signature of this source method must be written in this order and this form because it will later be given as input to the taint analysis in SecuCheck. When the analysis analyzes the source method in line 11 of the code excerpt at Listing 3.1, it will be transformed into a Jimple bytecode and look identical to the specified signature in the listing below. We write in this format because it is the only way to match the specified signature with the method we want to find in the analyzed program. If we do not specify the same order of elements in the signature or miss keywords such as `:` or `( )`, then the analysis does not find a match and returns no results, even though the method is still there in the program. The drawbacks when specifying signatures through the `String` object motivated our first extension.

```
1 // SOURCE
2 String sourceSignature = "account.UpdateAccount: "
3                          +"void updateAccount"
4                          +"(java.lang.String)";
```

Listing 3.3: Signature of the source method as `String` from Listing 3.2

Our new extension introduces a new element in FluentTQL, which is used as an alternative for specifying method signatures. The element is called `MethodSignature` and one can initialize signatures through the usage of builder pattern, similarly to how the methods and queries are written through `Method` and `TaintFlowQuery` objects in Listing 3.2. How the signature of the source method is specified through our DSL extension is shown below in Listing 3.4. To specify the signature, we initially call the constructor of `MethodSignature` and then use method completion in our IDE to get the first method suggestion of `atClass`. The latter accepts the name of the class where the source method is located, and we do not need to insert the `:` symbol, as was the case with the old signature through `String`. The fluent operation and method suggestion also happens for inserting the remaining parts of the signature. The names of the methods where we need to insert the parts of the signature are chosen to be as descriptive as possible. In `atClass` we insert the class name, in `returns` we write the return type that the method returns, in `named` we insert the name of the method, and in `accepts` we specify all parameters types that the method accepts. We do not need to write the parentheses `( )` as was the case in the old specification. Also, we can write more than one parameter by only separating them with a comma `,`, which one can also do in the old alternative through `String`. We also had the alternative to implement our extension in a manner that we could call `accepts` every time that we wanted to specify one of the arguments, but we rather chose this interface.

```
// SOURCE
MethodSignature sourceSignature = new MethodSignatureConfigurator()
                                      .atClass("account.UpdateAccount")
                                      .returns("void")
                                      .named("updateAccount")
                                      .accepts("java.lang.String")
                                      .configure();
```

Listing 3.4: Signature of the source method as `MethodSignature` from Listing 3.2

To conclude, our new extension of `MethodSignature` can be used to specify signatures in FluentTQL. It is implemented to avoid typos since users do not need to specify symbols such as `:` or `( )`, as it happens when specifying signatures through the old `String` alternative. Moreover, we added this extension because we think that it will assist inexperienced users in inserting all elements of the signature easily and not forget their order since each method name of `MethodSignature` is named to help users to remember what signature element should be written next. Lastly, the users of FluentTQL can combine this fluent manner for specifying method signatures and the old straightforward `String` alternative. Listing 3.5 below shows the FluentTQL specification for the vulnerability in Listing 3.1, when using the new language extension of `MethodSignature` object for specifying all signatures.

```
// SOURCE
MethodSignature sourceSignature = new MethodSignatureConfigurator()
                                      .atClass("account.UpdateAccount")
                                      .returns("void")
                                      .named("updateAccount")
                                      .accepts("java.lang.String")
                                      .configure();
Method source = new MethodConfigurator(sourceSignature)
                    .out().param(0);

// SANITIZER
MethodSignature sanitizerSignature = new MethodSignatureConfigurator()
                                      .atClass("account.UpdateAccount")
                                      .returns("java.lang.String")
                                      .named("sanitize")
                                      .accepts("java.lang.String")
                                      .configure();
Method sanitizer = new MethodConfigurator(sanitizerSignature)
                    .in().param(0).out().return();

// SINK
MethodSignature sinkSignature = new MethodSignatureConfigurator()
                                      .atClass("account.UpdateAccount")
                                      .returns("int")
                                      .named("executeUpdate")
                                      .accepts("java.lang.String")
                                      .configure();
Method sink = new MethodConfigurator(sinkSignature)
                    .in().param(0);

// TAINT FLOW QUERY
TaintFlowQuery taintflow = new TaintFlowQueryBuilder("SQL-Injection");
taintflow.from(source).notThrough(sanitizer).to(sink)
                    .report("SQL injection vulnerability.").at(Location.SINK)
                    .build();
```

Listing 3.5: Specification of vulnerability in Listing 3.1 through `MethodSignature`

3.1.1 Implementation

This subsection includes general information on the conducted implementation for the element of `MethodSignature` in FluentTQL. Figure 3.1 below shows a class diagram of the main language elements of FluentTQL and the additions that were done to it. There are classes represented in the while color like `TaintFlowQuery` and `Method`, which were already present in FluentTQL's syntax, as well as our additions which we have represented in the yellow color; the grey-colored `FlowParticipant` is an interface. The `MethodSignature` class was added to the language and implementation-wise we incorporated it on the *InternalFluentTQL* module of the **secucheck** repository, where all implementation of FluentTQL DSL can be found. An object of `MethodSignature` class can be initialized via the builder pattern by chaining all methods of `atClass`, `returns`, `named`, `accepts` in the noted sequence and typing the method `configure` at the end which returns the initialized object made up of the given signature elements. Moreover, we added in the language the possibility for the `Method` class to accepts signatures in `String` and `MethodSignature` formats. This can be seen in action in line 5 of Listing 3.2 and line 8 of Listing 3.5, where `MethodConfigurator` accepts signatures in both formats for the source method.

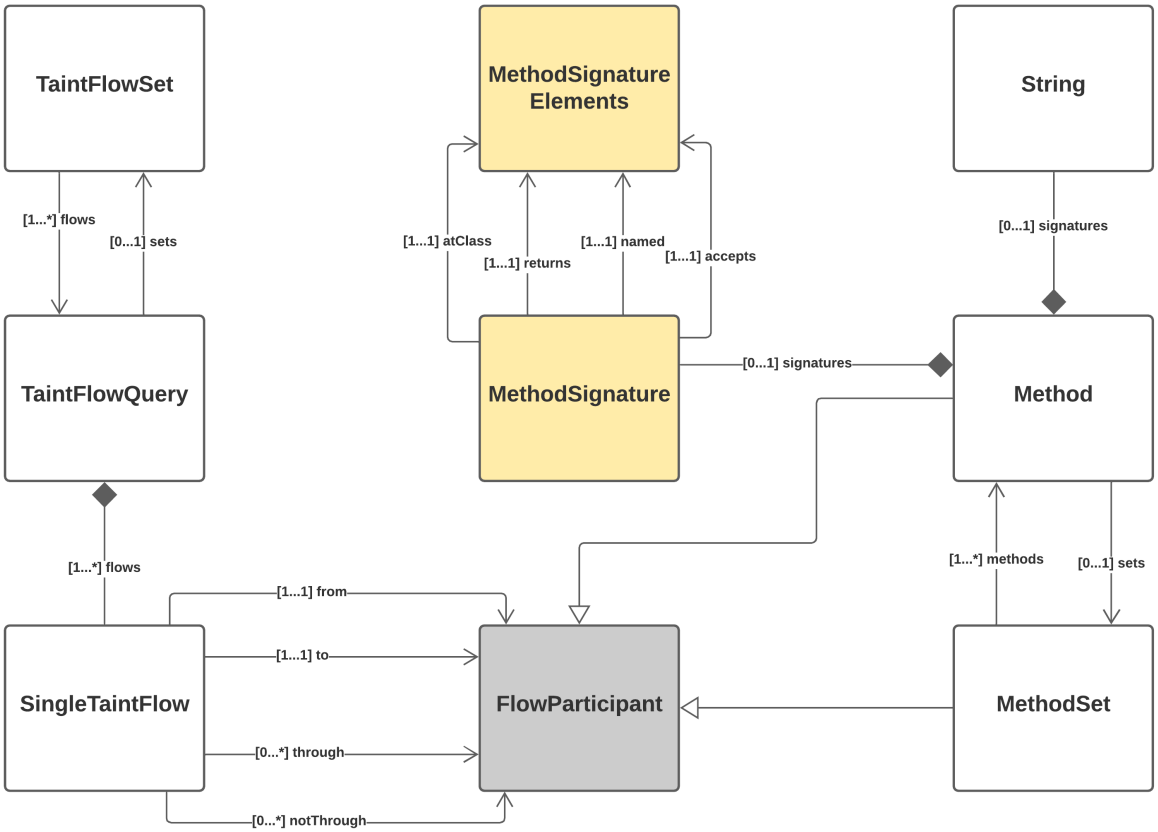

Figure 3.1: Class diagram of FluentTQL containing our first extension

Further additions were also done to the *secucheck-commons-utility* module in **secucheck** repository. This module performs a mapping between the specified taint queries from the *InternalFluentTQL* module in **secucheck** repository and the conceptual taint queries from the *query* module in **secucheck-core** repository, which are the ones that the taint analysis in

SecuCheck uses. We perform this transformation because we have a separation between the FluentTQL DSL that is implemented in the **secucheck** repository and the language elements stored in `secucheck-core`, which the analysis uses. This way, one can implement another DSL in **secucheck** repository and only needs to map the query elements of the new DSL to the ones in **secucheck-core** that the analysis utilizes. To continue to the implementation that we did in the *secucheck-commons-utility* module in **secucheck** repository, we added this mapping support when signatures were specified through `MethodSignature` objects. In such cases, we constructed a `String` signature from the elements of the `MethodSignature` object. We then mapped it to the respective signature in one of the classes of *query* module in **secucheck-core** repository, which the analysis uses. The constructed `String` signature looks exactly to how one would specify it via the `String` object. This way, the analysis works the same when both types of signatures are set in the FluentTQL language.

3.2 Increasing the expressivity when specifying signatures

This section details the second extension that we have performed, which is about the increase of expressivity in the language when specifying method signatures, specifically when writing signatures of overloaded methods.

In line 18 of Listing 3.1, the `executeUpdate` method is a sink in the given code excerpt, and the signature is specified in lines 16-20 in the respective specification in Listing 3.2. This sink method is part of the `java.sql.Statement` interface and is overloaded four different times as shown in the respective Java documentation [Ora]. The first overloaded version is the one used in the code excerpt and contains a query argument of type `String`. In contrast, the other three versions contain additional arguments besides the query one. Suppose all overloaded versions of `executeUpdate` would be part of the given taint flow. In that case, we need to include all of them in the respective specification if we are using FluentTQL without our second extension. The old manner of specifying the overloaded methods is demonstrated in Listing 3.6 below. All signatures of the overloaded methods are written as `String` objects. We could also use `MethodSignature`, but we chose the old manner in this specification.

```
1  // SINK 1
2  String sinkMethodSign1 = "account.UpdateAccount: "
3                           +"int executeUpdate"
4                           +"(java.lang.String)";
5  Method sinkMethod1 = new MethodConfigurator(sinkMethodSign1)
6                      .in().param(0)
7                      .configure();
8
9  // SINK 2
10 String sinkMethodSign2 = "account.UpdateAccount: "
11                          +"int executeUpdate"
12                          +"(java.lang.String,int)";
13 Method sinkMethod2 = new MethodConfigurator(sinkMethodSign2)
14                     .in().param(0)
15                     .configure();
16
17 // SINK 3
18 String sinkMethodSign3 = "account.UpdateAccount: "
19                          +"int executeUpdate"
20                          +"(java.lang.String,int[])";
21 Method sinkMethod3 = new MethodConfigurator(sinkMethodSign3)
22                     .in().param(0)
23                     .configure();
```

```
24
25 // SINK 4
26 String sinkMethodSign4 = "account.UpdateAccount: "
27                          +"int executeUpdate"
28                          +"(java.lang.String,java.lang.String[])";
29 Method sinkMethod4 = new MethodConfigurator(sinkMethodSign4)
30                      .in().param(0)
31                      .configure();
32
33 // SINKS
34 MethodSet sinks = new MethodSet("sinks")
35                  .addMethod(sinkMethod1)
36                  .addMethod(sinkMethod2)
37                  .addMethod(sinkMethod3)
38                  .addMethod(sinkMethod4);
```

Listing 3.6: FluentTQL specification that uses multiple overloaded `String` signatures

We see that we need to write many lines to specify the sinks in this specification. The above specification is not even complete because we still need to add the source, sanitizer, and the taint flow query specification. FluentTQL did not have a better way of specifying signatures of overloaded methods. A user needed to write each overloaded method, which made the process a bit repetitive. In the case of the `executeUpdate` method, the signatures of the overloaded methods are almost the same, apart from the arguments the various methods accept, which change slightly from one method to the other.

We have developed another approach for specifying overloaded methods in the above example and have presented it in Listing 3.7. In lines 1-3, we have specified the signatures of all versions of the overloaded `executeUpdate` method and achieved it in only one statement. This is achieved by using our second extension, which is specifically helpful in such cases. The signature is constructed from elements such as the class where the method is located, the return type, and the method name, which remain the same among all overloaded methods. In the part of the argument of the signature, we use the keyword `ANY`, which encapsulates all possible parameters that the overloaded `executeUpdate` methods can have. In lines 4-5, we have specified the overloaded sink methods, and the fluent way of specifying the other actions can continue as usual with `in()` and `param(0)`. The latter specifies that we are only interested in the first `query` argument of every overloaded method. The taint analysis in SecuCheck can recognize the usage of the `ANY` keyword and take appropriate actions about it. We also could choose other keywords such as `ANY_ARGUMENTS` or `ANY_PARAMETERS` but decided to stick to a shorter word. The signature in the specification below can also be specified via our first extension of `MethodSignature`. The user has alternatives when wanting to use the `ANY` keyword in the signature. We were motivated to implement this extension for users who want to specify overloaded versions of a method via a shorter alternative, compared to the old one. It can also be used by those who do not want to take the time to specify all arguments of a signature, as there may be cases where a method signature contains a sizeable amount of parameters. This may also help in avoiding typos during the writing of arguments in a signature.

```
1 String sinkMethodSigns = "account.UpdateAccount: "
2                          +"int executeUpdate"
3                          +"(ANY)";
4 Method sinks = new Method(sinkMethodSigns)
5                .in().param(0);
```

Listing 3.7: Specifying overloaded methods via the `ANY` keyword

Another included feature in our second extension is the underscore (`_`) keyword that can be used when specifying arguments. This keyword is valuable when we do not want to specify a

specific parameter but rather use this keyword, representing that any parameter can be included there. Suppose we would want to write all overloaded versions of the `executeUpdate` method that only contain two parameters. In that case, we can achieve it by using this feature as demonstrated in Listing 3.8 below. In this specification, we have not taken into account the `executeUpdate` that only contains the `query` argument, and the taint analysis will only look for those overloaded versions that have two arguments.

```
String sinkMethodSigns = "account.UpdateAccount: "
                         +"int executeUpdate"
                         +"(_,_)";
Method sinks = new Method(sinkMethodSigns)
                 .in().param(0);
```

Listing 3.8: Specifying method arguments via the `_` keyword

Besides this second extension, we also eased how signatures must be generally specified in FluentTQL. Before, the signature specification had to follow strict rules regarding the spacing of the signature elements, like class, return type, method name. All of these elements in the signature had to be separated by only one space; if there were more than one, then the taint analysis would not match the analyzed code. Similarly, the arguments of a method signature had also to be separated by only one space. As we see, the signature specification was not very user-friendly, and that was our motivation to improve the signature specification experience. Our second extension also includes very lenient rules when typing signatures. A user can insert as many whitespace characters as they want to separate the signature elements. Listing 3.9 below includes an example of how the signature in Listing 3.7 can be written with the lenient rules regarding whitespace characters.

```
String sinkMethodSigns = "account.UpdateAccount: "
                         +"int   executeUpdate  "
                         +"(  ANY  )";
```

Listing 3.9: Specifying overloaded methods via the `ANY` keyword and more white spaces

The proposal of this thesis stated that we had only to implement the feature regarding the `ANY` keyword. All other features included in this second extension, like the introduction of the `_` keyword and more lenient rules when typing signatures, are added additionally to this extension in order to increase the expressivity when specifying method signatures.

3.2.1 Implementation

All of the implementations about this extension are done in the **secucheck-core** repository. In there, we have created a new module that is named *parser*. The main functionality of the *parser* module is that it checks whether a given FluentTQL flow participant is equal to an analyzed method in the application. This functionality is used inside the *implementation* module of **secucheck-core** repository, which contains the implementation of the taint analysis in SecuCheck. The *parser* module does a better checking on whether a specified signature is equal to an analyzed method in a program. It includes all of our features like the detection of `ANY` or underscore (`_`) keywords that are usually used when specifying overloaded signatures. It also includes tolerant rules regarding the whitespace characters that may be present in a method signature. Previously, the checking on whether a given signature was equal to an analyzed method was performed only via the `equals` method from `String` class. This led to many False Negatives as the analysis would report no vulnerabilities even when they were there. Now, our parser receives two arguments: a specified signature and an analyzed method in a program. Then, it parses them both and only after applying all the rules and features that we have implemented, decides whether they match with each other or not.

3.3 Utilization of entry points for taint flow queries

This section covers our third extension, which is about the specification of entry points in the language for given taint flow queries. We cover the usages of our extension and its implementation details.

Let us consider a very large program that contains the `updateAccount` method from Listing 3.1 in it. When we start the taint analysis in SecuCheck with the given specification in Listing 3.2 in order to find any vulnerabilities in our big application, the analysis queries the complete statements of the application in search of the specified taint flows. The larger the application, the more time it will take for the analysis to query the program, and the more time will be spent till be analysis is completed. In the old version of SecuCheck, by default, the analysis would start its search in all public methods of this extensive application to find our SQL Injection vulnerability, even though it is only located in the `updateAccount` method. A considerable amount of time would be spent in scanning methods that do not contain the vulnerability. This served as motivation for our third extension.

In the terminology of static analysis, the methods where the analysis starts its search are called entry points. Our third extension makes it possible to specify entry points for particular taint flow queries so that the taint analysis only looks in these methods for the vulnerabilities. If specified taint flows are present in such entry points, vulnerabilities will be reported, and the analysis will finish. In Listing 3.10 below, we have demonstrated a specification that contains the usage of our entry point extension. Since the SQL Injection vulnerability is located in the `updateAccount` method, that is the one that is set as an entry point, as can be seen in lines 23-27. Initially, we have specified the signature of the entry point method in lines 23-25. The signature of an entry point method is written the same way as the one of a flow participant method. It can be initialized as a `String` or `MethodSignature` object. The signature is then passed to a constructor of the `EntryPoint` interface in line 26, which in this case is `MethodEntryPoint`. The latter is finally added to a list containing `EntryPoint` objects. We can add more than one entry point in this list, which is at last associated with the `TaintFlowQuery` specification via `atOnlyDSLEntryPoints` method.

The entry point enhancement is helpful because the users who specify the taint flows in the language might know where the vulnerability resides or want to examine specific areas in the program.

```
1  // SOURCE
2  String sourceSignature = "account.Account: "
3                           +"void updateAccount"
4                           +"(java.lang.String)";
5  Method source = new MethodConfigurator(sourceSignature)
6                  .out().param(0);
7
8  // SANITIZER
9  String sanitizerSignature = "account.Account: "
10                          +"java.lang.String sanitize"
11                          +"(java.lang.String)";
12 Method sanitizer = new MethodConfigurator(sanitizerSignature)
13                 .in().param(0).out().return();
14
15 // SINK
16 String sinkSignature = "account.Account: "
17                          +"int executeUpdate"
18                          +"(java.lang.String)";
19 Method sink = new MethodConfigurator(sinkSignature)
20                 .in().param(0);
21
```

```
22 // ENTRY POINT
23 String methodEntryPointName =  "account.Account: "
24                                 +"void updateAccount"
25                                 +"(java.lang.String)";
26 MethodEntryPoint entryPoint = new MethodEntryPoint(methodEntryPointName);
27 List<EntryPoint> entryPoints = Arrays.asList(entryPoint);
28
29 // TAINT FLOW QUERY
30 TaintFlowQuery taintflow = new TaintFlowQueryBuilder("SQL-Injection");
31 taintflow.atOnlyDSLEntryPoints(entryPoints)
32         .from(source).notThrough(sanitizer).to(sink)
33         .report("SQL injection vulnerability.").at(Location.SINK)
34         .build();
```

Listing 3.10: Specification containing an entry point for finding the weakness in Listing 3.1

3.3.1 Implementation

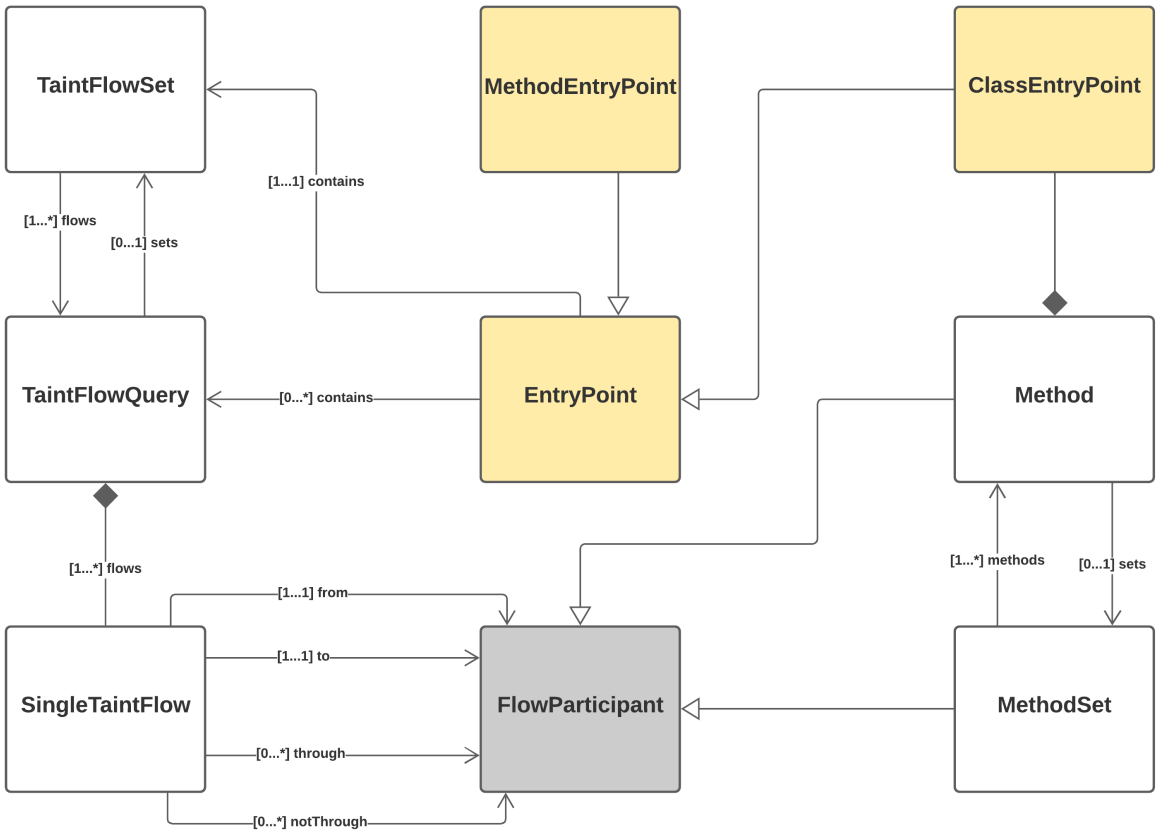

Figure 3.2: Class diagram of FluentTQL containing our third extension

This subsection contains implementation details regarding our third extension. Initially, we have performed additions to the language in the *InternalFluentTQL* module of **secucheck** repository. In Figure 3.2 above, we have a representation of the elements that were added to the FluentTQL language. In yellow, we can see the elements that were added: `EntryPoint` interface is the main one, which comes with its two implementations of `MethodEntryPoint` and `ClassEntryPoint`. The functionality of `MethodEntryPoint` is already demonstrated in

the previous listing, where we specify methods via signatures as `String` or `MethodSignature` elements. This way, we can also use the first extension of `MethodSignature` in the entry point extension. On the other hand, we have enhanced the syntax of FluentTQL and can specify class entry points as `ClassEntryPoint` objects. In here, we only specify the class part of the signature, for example, `account.Account`. In such cases, the analysis takes the class and loads all its public methods as entry points. Between both entry point alternatives, the `MethodEntryPoint` is more fine-grained for specifying entry points since one can choose specific methods rather than classes. Next, a list of entry points is associated with a `TaintFlowQuery` or `TaintFlowQuerySet` via the `atOnlyDSLEntryPoints` method, as covered in the previous listing.

Changes that are done the *InternalFluentTQL* module of **secucheck**, are also similarly performed to the *query* module in **secucheck-core** repository. As covered in Subsection 3.1.1, the *query* module holds the conceptual elements of the language, like taint flows queries, which the analysis uses to start. Since the `EntryPoint` element is a new element in FluentTQL, we also added it to the *query* module. Finally, for the analysis to use the entry point information, we did the required mapping between the taint flows in *InternalFluentTQL* and the ones in *query* module, which the analysis uses; the mapping was done in the *secucheck-commons-utility* module in **secucheck**, similarly to how it was done for the first extension of `MethodSignature` in Subsection 3.1.1.

Lastly, required changes were added to the *implementation* module in **secucheck-core** repository, which contains the implementation of the taint analysis in SecuCheck. In here, when entry points are set in the language via the `atOnlyDSLEntryPoints` method, the analysis only considers the specified entry points, and not all public methods in a program.

3.4 Ability to detect HCC and NPD vulnerabilities

This section covers our fourth extension of specifying new types of vulnerabilities in FluentTQL and using the extended analysis to find them. We initially cover the previous drawbacks of our tools, the extension conducted to them regarding this extension, and related implementation details.

One of the drawbacks of FluentTQL was its inability to express taint flows related to Hard-coded Credentials (HCC) and NULL Pointer Dereference (NPD) vulnerabilities, as described in the introductory Chapter 1. In such vulnerabilities, it is quite common for the source in a taint query to be a variable containing a constant or a null value. The code excerpt in Listing 3.11 (slightly modified from [CWEb]) is written in Java and illustrates a Hard-coded Credentials vulnerability. In lines 9 and 10, the username and password are hardcoded as String types and are later used in line 11, where security-relevant operations are performed. This is a fault since anyone who has the source code can get hold of the username and password of this user. Even if someone has the bytecode of this program, they can still easily unscramble it through the `javap -c` command and obtain the user credentials. In the example below, variables in lines 9 and 10 can be considered sources, and the method in line 11 can be seen as a sink. FluentTQL is used to model sources and sinks to the granularity of methods. Therefore, it could not specify a variable as a source, as it happens in Hard-coded Credentials and Null Pointer Dereference vulnerabilities.

Listing 3.12 below shows a taste of how the taint flows are specified in FluentTQL through our fourth extension, for catching the vulnerability in Listing 3.11. In line 2, the specified source is an enumeration of name `Variable` and accesses its constant with name `HARDCODED`. This defines that any constant or hard-coded variable will be set as a source. This includes the

`username` and `password` variables in lines 9 and 10 of Listing 3.11. The sink in lines 5-12 of the listing below is specified the same way as one would in a FluentTQL specification; the method signature and the parameters that should be tracked are supplied, where the parameters are the second and third ones. The taint flow is presented in lines 15-19 and is used to query possible flows of hard-coded data to the sink method. This specification can be reused to query many programs that contain the usage of the `getConnection()` sink method in line 11 of Listing 3.11, as the taint analysis would only be interested in finding hard-coded credentials that are flowing into the sink method.

```
1  package connection;
2  // imports
3
4  public class Connection {
5      // fields and methods
6
7      public void performConnection(String url) {
8          ...
9          String username = "scott";
10         String password = "tiger";
11         DriverManager.getConnection(url, username, password);
12         ...
13     }
14     // other methods
15 }
```

Listing 3.11: Java example containing a Hard-coded Credentials vulnerability

```
1  // SOURCE
2  Variable source = Variable.HARDCODED;
3
4  // SINK
5  String sinkSign = "connection.Connection: "
6                    +"java.sql.Connection "
7                    +"getConnection"
8                    +"(java.lang.String, "
9                    +"java.lang.String, "
10                   +"java.lang.String)";
11 Method sink = new MethodConfigurator(sinkSign)
12                 .in().param(1).param(2);
13
14 // TAINT FLOW QUERY
15 TaintFlowQuery taintflow = new TaintFlowQueryBuilder("Hard-coded Credentials");
16 taintflow.from(source).to(sink)
17         .report("Hard-coded Credentials vulnerability.")
18         .at(Location.SOURCE)
19         .build();
```

Listing 3.12: Specification for finding the HCC vulnerability in Listing 3.11

Correspondingly, we can also write specifications for finding NPD vulnerabilities. In such cases, the source would be the `Variable` enumeration that accesses the `NULL` constant in it. Same as with the HCC vulnerability, the taint analysis will be carried out to find all NULL values that are flowing into the specified sink method.

To conclude, the fourth extension makes it possible to specify HCC and NPD vulnerabilities and find them through the analysis. In this extension, we can also use our third extension of entry points to set specific entry points and try to make the analysis complete faster.

3.4.1 Implementation

In this part, we present some implementation details about our fourth extension. Figure 3.3 shows a class diagram with additions to FluentTQL. These additions were performed in the *InternalFluentTQL* module in **secucheck** repository, which holds the implementation of FluentTQL. As we notice from the figure, we have created an enumeration called `Variable` which takes the possible values of `HARDCODED` and `NULL`. This language element is a `FlowParticipant`, the same way how `Method` and `MethodSet` are. The `Variable` enumeration can only be set as a source via the `from` method in a query and cannot be a sink, propagator, or sanitizer. This is because such enumerations are used from the implemented analysis to find HCC and NPD vulnerabilities.

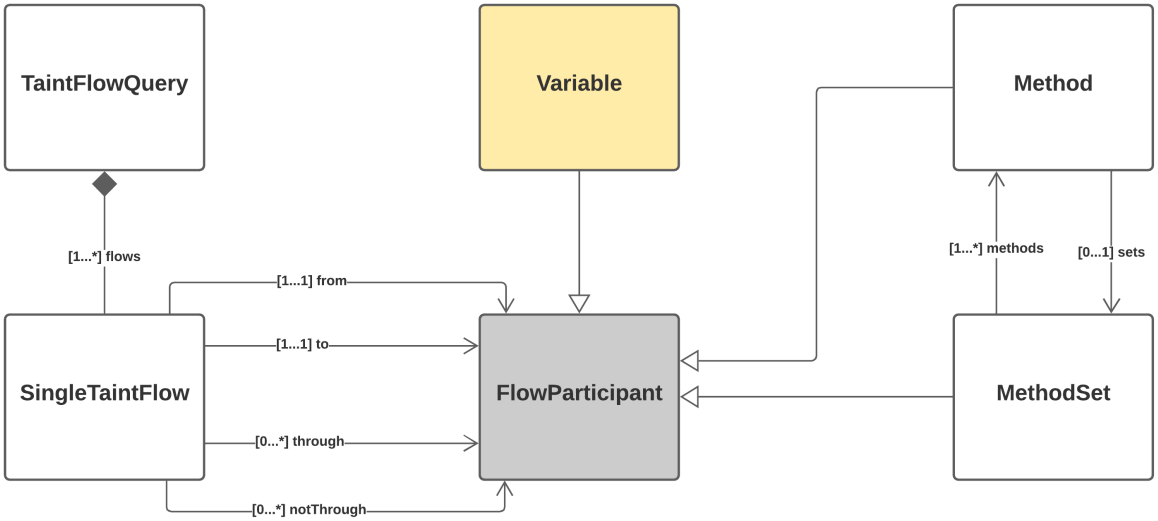

Figure 3.3: Class diagram of FluentTQL containing our fourth extension

Besides the additions to the language, respective implementation is also performed in the *query* module in **secucheck-core** since it did not previously contain the usage of variables as flow participants. Moreover, we added the standard mapping for this extension in the *secucheck-commons-utility* module of **secucheck** repository, as similarly performed in the first and third extensions. This mapping is between the added additions in *InternalFluentTQL* module and the *query* module so that the analysis can utilize the usage of `Variable` in the language.

The last and most important additions for this extension were made to the taint analysis that is found in the *implementation* module in **secucheck-core** repository. Previously, the taint analysis built through the Boomerang framework only worked forward, where it would initially search for all specified sources. Upon finding any, it would then start a demand-driven analysis via Boomerang for searching other specified elements in a query like propagators, sanitizers, and sinks. If the analysis found the specified flow participants and formed a taint flow, it would be reported. For our extension, we have extended the analysis in Boomerang and added a backward analysis that will be used in cases when `Variable.HARDCODED` and `Variable.NULL` are used as sources in the language. In such cases, the analysis first starts to search for the specified sinks and, when finding any, starts a backward analysis to find the other flow participants. When `HARDCODED` is specified, it looks for hardcoded variables, such as `String`, `int`, or `byte[]` objects. On the other hand, when `NULL` is used, it searches for variables initialized to null.

3.5 Addition of a backward taint analysis

This section covers our fifth and final extension related to implementing a fully functional backward taint analysis. This extension was not included in the initial proposal of this thesis, like was the case with the other four extensions, but was added voluntarily. We initially cover the drawbacks of forward-only analysis and then explain some implementation details.

The old version of the analysis only worked in a forward fashion. This was generally all right for most scenarios, but it had its drawbacks in some specific cases. Let us take as an example the demo specification in Listing 3.13. As we notice, it contains ten different source methods that are accumulated into a `MethodSet` object specified as `sources`. On the other side, it only has a sink method. Both flow participants, sources and sink, form the basis of the `TaintFlowQuery` in lines 25-29. Let us further imagine a large application which the SecuCheck tool will analyze by taking the specification below as input. The imaginary program has in it all the of the specified sources but no sink. As a result, the analysis will run but produce no vulnerabilities, as there are not any. However, there is a drawback when using the forward analysis for these kinds of scenarios. The analysis going in this direction will initially search for all of the specified sources and, in our case, find all ten of them in the given program. Then, for each source that is found, it will start a demand-driven and forward analysis for finding the other flow participants, which in our case is only the sink. Unfortunately, for each of the ten demand-driven analyses that will start, the analysis will waste time querying the extensive program as there is not any sink available. For such scenarios, it would be better if the analysis runs in the backward direction, as is explained in the implementation part of our extension in the following subsection.

```
1  // SOURCE 1
2  String sourceSign1 = "...";
3  Method source1 = new MethodConfigurator(sourceSign1)
4                       .out().param(0);
5  .
6  .
7  .
8  // SOURCE 10
9  String sourceSign10 = "...";
10 Method source10 = new MethodConfigurator(sourceSign10)
11                      .out().param(0);
12
13 // SOURCES
14 MethodSet sources = new MethodSet("sources")
15                      .addMethod(source1)
16                      ...
17                      .addMethod(source10);
18
19 // SINK
20 String sinkSign = "...";
21 Method sink = new MethodConfigurator(sinkSign)
22                  .in().param(0);
23
24 // TAINT FLOW QUERY
25 TaintFlowQuery taintflow = new TaintFlowQueryBuilder("SQL-Injection");
26 taintflow.from(sources).to(sink)
27          .report("SQL Injection vulnerability.")
28          .at(Location.SOURCEANDSINK)
29          .build();
```

Listing 3.13: Example specification containing ten sources and one sink

3.5.1 Implementation

This part includes details on our implemented fifth extension. It consists of a backward taint analysis that is used by the SecuCheck tool when some conditions are met. This backward direction would make more sense in the previous scenario than the forward direction. If we had used the backward direction, the analysis would initially search for the specified sinks in the program. Since we have specified only one sink and it is not available in the program, the analysis would terminate and report no findings. This is much more effective than the waste of resources in the case of the forward analysis.

Our backward analysis is implemented in the *implementation* module in the **secucheck-core** repository. It works the same as the forward analysis, just in the opposite direction. Both analyses are executed through the Boomerang framework and are chosen automatically via simple rules that the user cannot choose. The backward analysis is the same as the one implemented in the fourth extension and works for all types of specified flow participants. The forward analysis is always started for a given taint flow that contains an equal number of sources and sinks or contains fewer sources than sinks. We chose this rule since we still want to have the forward analysis as the default one and also take full advantage of other similar scenarios like the one in Listing 3.13, in case there are fewer sources than sinks given in a specification. On the other hand, the backward analysis that we have extended is always started when there are more sources than sink specified in a given FluentTQL specification. We offer both directions of the taint analysis in SecuCheck, which would increase its overall performance by following these simple rules. We conclude with a pseudo-code shown in Listing 3.14 which encapsulates how the direction of the analysis is chosen in SecuCheck.

```
...
int sources = getSources();
int sinks = getSinks();

if(sources <= sinks) {
    startForwardAnalysis();
}
else {
    startBackwardAnalysis();
}
...
```

Listing 3.14: Pseudo-code about how the direction of the analysis is chosen

Since this extension was added voluntarily to this thesis, we did not carry out an evaluation. The fifth extension was only tested manually, where we used the specifications in the *demo-project-specification* module in **secucheck-catalog** repository. Then, we only ran our backward analysis to find the specified taint flows in the *demo-project* module in the **secucheck-catalog** repository. Test results showed that our extension is fully functional and produces the same analysis results as the analysis run in the forward fashion.

4 Evaluation

This chapter includes details about the evaluation performed to the extensions made in FluentTQL and SecuCheck tools. Out of the five enhancements, only four of them have had an evaluation. The first and second extension which were covered in Section 3.1 and 3.2 were evaluated by the means of a user study. The third extension that was covered in Section 3.3 was evaluated through an experimental approach. The fourth extension enclosed in Section 3.4 was evaluated through a more sporadic approach. The following sections contain information on the user study and experiments that were performed, how the respective enhancements were evaluated, the obtained results, and the research questions that were answered.

4.1 User Study

A user study was performed to evaluate the first two extensions: an alternative option of writing method signatures and the increase of expressivity when specifying method signatures in the language. Both features included essential changes to the syntax of FluentTQL. They are meant to be used frequently by the users of the language. For this reason, the user study was meant to evaluate the usability of the extensions in the language. The user study included a few participants due to the time constraints of the thesis and the effort required to process the results. Since FluentTQL was developed as a straightforward DSL meant to be used by developers without extensive security expertise, we chose ordinary language users to participate in the study and evaluate our extensions. Both extensions offer solutions to specification tasks that can already be solved through the old version of FluentTQL before the extensions were introduced. For this reason, the user study was meant to compare both new enhancements to their respective old ones, by utilizing practical tasks where participants were asked to write specifications by using new and old features. At the end of each task, a questionnaire followed with questions that helped us gather usability information through metrics such as SUS [B+96] and NP [Rei03], as well as general comments on the old and new language features. Through the results of this study, we were able to answer the following research questions:

1. How usable is each new extension for software developers?

2. How does the new extension compare to the old one for specifying method signatures for taint flow queries?

The following subsections contain information on the participants that were chosen in this study, the tasks that participants needed to solve through our extensions, details on the questionnaires that were built, the overall setup of each user study session, and interpretation of the results that were derived from the gathered usability data.

4.1.1 Participants

The user study hosted seven participants in total. One of them was used for a test session to determine the difficulty of the specification tasks and better format the time length of the user study sessions. Three were computer science students, two were professional software developers, and one was a computer science researcher. All of them were knowledgeable in software security, cryptography, taint analysis, and DSL usage. As mentioned before, we tried to choose participants meant to be ordinary users of FluentTQL and without extensive experience in security. This is because the DSL itself and its evaluated extensions are meant to be used frequently by developers with an average understanding of taint analysis and DSL usage.

Each user study session started with an informational survey that the participant initially needed to fill. The questions were meant to gather general information on the participant. In the list below, we can find the questions that each one of them answered:

1. How many years of coding experience do you have? (Answer: 1-2, 3-5, 6-9, 10+ years)
2. How do you see yourself in software security? (Answer: Beginner, Intermediate, Advanced, Expert)
3. What is your knowledge in static analysis? (Answer: Beginner, Intermediate, Advanced, Expert)
4. How much interested are you to learn Domain Specific Languages? (Answer: 1-10)

In the first three questions, each participant could only choose one answer where we gathered data about their coding, software security, and static analysis knowledge. In the last question, we collected information on the participants' interest in learning DSLs, which also hints at their eagerness to use DSLs. The questionnaire was built by using Google Forms [Goo] because it is a simple tool for building surveys, as was the case with our informational one, and also offers a cloud platform for storing the answers of the participants. One can click in the following link in order to visit the informational survey.

	Participant Answers					
Questions	**P. 1**	**P. 2**	**P. 3**	**P. 4**	**P. 5**	**P. 6**
Q. 1	6-9 years	3-5 years	3-5 years	1-2 years	3-5 years	3-5 years
Q. 2	INT	INT	INT	BEG	INT	INT
Q. 3	INT	INT	INT	BEG	INT	INT
Q. 4	6	7	8	5	8	7

Table 4.1: Results of the informational survey

Table 4.1 above shows the results of the informational survey and the data that was gathered from all participants. For the first question that asks about coding experience, most participants have an average experience of 3-5 years, meaning that they could easily understand the syntax of FluentTQL, which uses a common language such as Java as its host language. Also, regarding their knowledge in security and static analysis, most participants rated themselves as

Intermediate, which hints that they could handle the concepts of method signatures and flow participants in the DSL to complete the given tasks. Lastly, all participants answered with five or above on a rate from 1 to 10 about their interest in DSLs, which shows that they were eager to use our language. Overall, the data collected from the initial survey shows that we successfully picked participants that resemble usual users of FluentTQL, who are interested in using DSLs and have good knowledge in software security. This means that even though the usability results come from a small number of participants, they can be considered meaningful.

4.1.2 Tasks

Each participant went through a set of two tasks in each of their sessions to use our new features and the respective old ones to solve the given problems. All the code excerpts and the specifications that the participants needed to write were collected in a single user study project under the following link. This project contains three Maven [Fou] modules that the participant could import in their IDE or editor:

- **demo-project-1**
- **demo-project-2**
- **demo-project-specifications**

The first demo project contains code excerpts related to the first task that the participants needed to solve. The first task is about writing a simple method signature in FluentTQL through two different alternatives: the old one of using a `String` object and the new one that we implemented, which is through the `MethodSignature` object. As shown in Listing 4.1, the project contains a code example that shows a taint flow from a source (line 7) to a sink (line 12) method. The tainted variable `tainted1` is propagated through the program till the variable `tainted3`, which is then used in the `sink` method. The taint flow itself does not show a real-life vulnerability but is rather a demonstration to show the participant how a taint flow can manifest in a program. For this reason, the naming of the variables and methods was chosen to be meaningful to a taint flow scenario.

```
1  package example;
2
3  public class Main {
4
5    public static void main(String[] args) {
6      // SOURCE
7      String tainted1 = source(args);
8      String tainted2 = tainted1;
9      String tainted3 = tainted2;
10
11     // SINK
12     sink(tainted3);
13   }
14
15   // more code
16 }
```

Listing 4.1: Example code in the first demo project

The FluentTQL specifications that show the vulnerability in Listing 4.1 are located in **demo-project-specifications** and were used to demonstrate to the participants how to model the

vulnerability in our language. Most attention during the explanation was given to specification alternatives of method signatures, such as `String` and `MethodSignature`. This way, each participant could grasp the concepts and later use them for the exercises they had to solve.

The first demo project also contains another code excerpt which is shown in Listing 4.2. This is an exercise that contains an SQL injection vulnerability for which the participants needed to write specifications by using the `String` and `MethodSignature` options for specifying the signatures. The coding exercise is similar to the example code in Listing 4.1, by being simple to follow and understand; it just contains some lines of code and only a taint flow from the source (line 8) to the sink (line 13) method, in order for the participants to quickly grasp the scenario. Then, the participants had to write the specifications that were related to this code excerpt.

```
1  package exercise;
2  // imports
3
4  public class Main {
5    // some code
6
7    // SOURCE
8    public static ResultSet getUser(String userId) throws SQLException {
9      String query = "SELECT * FROM Users WHERE UserId = "+ userId;
10     Connection c = DriverManager.getConnection(DB_URL, USER, PASS);
11
12     // SINK
13     ResultSet result = c.createStatement().executeQuery(query);
14     return result;
15   }
16
17 }
```

Listing 4.2: Coding exercise in the first demo project

In Listing 4.3 we can find the specification that each participant needed to complete related to the code in Listing 4.2. This specification is located in **demo-project-specifications** and is, for the most part, filled out. The respective sink method and its signature is already specified by using the older `String` option. The source method is also written, with only its signature left blank, which the participant had to write. The participant could still use how the sink signature was specified as a hint to complete this exercise. This was the first half of the first task where a participant had to write a signature using one of the two alternatives.

```
1  // SOURCE
2  // ToDo: specify the method signature for the source
3  String sourceMethodSign;
4  Method sourceMethod = new MethodConfigurator(sourceMethodSign)
5                        .out().param(0)
6                        .configure();
7
8  // SINK
9  String sinkMethodSign = "exercise.Main: "
10                         +"java.sql.ResultSet "
11                         +"executeQuery"
12                         +"(java.lang.String)";
13 Method sinkMethod = new MethodConfigurator(sinkMethodSign)
14                       .in().param(0)
15                       .configure();
16
17 // other specifications
```

Listing 4.3: FluentTQL specification related to Listing 4.2 that uses `String`

Furthermore, Listing 4.4 contains the specification which is again related to Listing 4.2. This specification is as well located in the **demo-project-specifications** and had to be completed by the participants. Similar to the specification in Listing 4.3, it is filled out for the most part, and each participant had to write only the signature for the sink method. The respective signature had to be written by using the new option of `MethodSignature` and the participant could follow the manner of how the sink method was specified in case of difficulties during the exercise. This exercise concluded the first task of writing the signature with the second alternative.

```
1  // SOURCE
2  // ToDo: specify the method signature for the source
3  MethodSignature sourceMethodSign;
4  Method sourceMethod = new MethodConfigurator(sourceMethodSign)
5                        .out().param(0)
6                        .configure();
7
8  // SINK
9  MethodSignature sinkMethodSign = new MethodSignatureConfigurator()
10                                  .atClass("exercise.Main")
11                                  .returns("java.sql.ResultSet")
12                                  .named("executeQuery")
13                                  .accepts("java.lang.String")
14                                  .configure();
15 Method sinkMethod = new MethodConfigurator(sinkMethodSign)
16                       .in().param(0)
17                       .configure();
18
19 // other specifications
```

Listing 4.4: FluentTQL specification related to Listing 4.2 that uses `MethodSignature`

The second demo project contains code excerpts related to the second task that the participants needed to write. It was about writing method signatures in FluentTQL when overloaded methods were flow participants in a taint flow. The participants would use the old manner of specifying all overloaded methods one by one and the new option where all signatures of given overloaded methods would be written on a single line of code. Listing 4.5, similar to the code example in the first demo project, contains a taint flow example from a source (line 7) to three overloaded sink methods (lines 12, 14, and 16).

```
1  package example;
2
3  public class Main {
4
5    public static void main(String[] args) {
6      // SOURCE
7      String tainted = source(args);
8      String strValue = "string value";
9      int intValue = 1;
10
11     // SINK 1
12     sink(tainted);
13     // SINK 2
14     sink(tainted, strValue);
15     // SINK 3
16     sink(tainted, strValue, intValue);
17   }
18
19   // more code
20 }
```

Listing 4.5: Example code in the second demo project

Respective specifications for the vulnerability in Listing 4.5 are located in **demo-project-specifications** where it is shown to each participant how to write the signatures of the sink methods by using the old and new options. This way, each participant initially learns how both alternatives are used in the language before a similar task is presented in the next step. Also, the code example in the second project does not contain a real-world vulnerability. Instead, it shows how overloaded versions of a method can participate as sinks in a taint flow.

Moreover, Listing 4.6 tells about the code excerpt that each participant needs to write FluentTQL specifications. This is located in the second demo project, alongside the code excerpt in Listing 4.5. Similar to the code exercise in the first demo project, this exercise also contains an SQL Injection. The exercise code below has a similar structure to its counterpart (the example code), where it accommodates one source and three overloaded sink methods. The source (line 8) is `updateAccount` which accepts a parameter named `customerId`, that is in fact a user input. This parameter is then appended to the `query` variable, that in turn ends up as argument in the three overloaded methods named `executeUpdate`. Each participant had to write the signatures of the sink methods in FluentTQL by using both alternatives: old and new. This task was performed in order for the participants to try both options and test their usability.

```
1  package exercise;
2  // imports
3
4  public class Main {
5    // some code
6
7    // SOURCE
8    public static void updateAccount(String customerId) throws SQLException {
9      String query = " UPDATE people SET name = 'John' where id = " + customerId;
10     Connection c = DriverManager.getConnection(DB_URL, USER, PASS);
11
12     if(customerId.length() < 10) {
13       // SINK 1
14       c.createStatement().executeUpdate(query);
15     }
16     else if(customerId.length() == 10) {
17       Random random = new Random();
18       // SINK 2
19       c.createStatement().executeUpdate(query, random.nextInt(10));
20     }
21     else if(customerId.length() > 10) {
22       // SINK 3
23       c.createStatement().executeUpdate(query, new int[]{ 1,2,3,4,5 });
24     }
25     else {
26       System.out.println("Cannot update the customer ID.");
27     }
28   }
29
30 }
```

Listing 4.6: Exercise code in the second demo project

In Listing 4.7 we can see the specification that each participant needed to write related to the coding exercise in the second demo project. The respective source method and its signature is already specified by using the `String` option. The `MethodSignature` option could have also been used. However, we chose to use only one alternative for specifying all signatures related to the second task, so the participants would rather focus on the objective of this task. The sink methods are also written beforehand, with only their respective signatures left blank, which the participant needed to write. The participant could utilize the experience gathered in the first

task where the specification of signatures through `String` object was used. As demonstrated, the participant had to specify the signatures for each of the given sinks, one after the other. In the end, all sink methods were collected into a `MethodSet` object, which is in turn given to taint flow query containing the specified source and sinks. This concluded the first half of the second task, where a participant had to write signatures of overloaded methods participating in a taint flow, through the old alternative.

```
// other specifications
// SOURCE
String sourceMethodSign = "example.Main: "
                        +"void updateAccount"
                        +"(java.lang.String)";
Method sourceMethod = new MethodConfigurator(sourceMethodSign)
                    .out().param(0)
                    .configure();

// SINK 1
// ToDo: specify the method signature for the first sink
String sinkMethodSign1;
Method sinkMethod1 = new MethodConfigurator(sinkMethodSign1)
                    .in().param(0)
                    .configure();

// SINK 2
// ToDo: specify the method signature for the second sink
String sinkMethodSign2;
Method sinkMethod2 = new MethodConfigurator(sinkMethodSign2)
                    .in().param(0)
                    .configure();

// SINK 3
// ToDo: specify the method signature for the third sink
String sinkMethodSign3;
Method sinkMethod3 = new MethodConfigurator(sinkMethodSign3)
                    .in().param(0)
                    .configure();

// SINKS
MethodSet sinks = new MethodSet("sinks")
                .addMethod(sinkMethod1)
                .addMethod(sinkMethod2)
                .addMethod(sinkMethod3);

// other specifications
```

Listing 4.7: FluentTQL specification related to Listing 4.6 that uses multiple `String` signatures

Lastly, Listing 4.8 below, also shows a FluentTQL specification for the coding exercise in the second demo project. This specification is as well located in the **demo-project-specifications** and had to be completed by the participants. It is filled out for the most part, and each participant had to write only the signature for the sink methods, in the same fashion to the specification in Listing 4.7. The sink signatures had to be written by using the new option of specifying overloaded methods; each participant had to use the keyword `"ANY"` instead of writing the parameters of the method signature in order to complete the given specification. In comparison to its fellow specification in Listing 4.7, it has fewer lines of code since all the signatures for the three sink methods could be written in one line. Both completed exercises were built to offer participants two different perspectives of writing specifications for the same type of problem. This exercise concluded the last part of the second task. The second task was

also the last task of the user study.

```
// SOURCE
String sourceMethodSign = "example.Main: "
                        +"void updateAccount"
                        +"(java.lang.String)";
Method sourceMethod = new MethodConfigurator(sourceMethodSign)
                    .out().param(0)
                    .configure();

// SINKS
// ToDo: specify the method signature for all three sinks
String sinkMethodSign;
Method sinkMethod = new MethodConfigurator(sinkMethodSign)
                    .in().param(0)
                    .configure();

// other specifications
```

Listing 4.8: FluentTQL specification related to Listing 4.6 that uses a keyword for signatures

To conclude, the specifications related to the code examples in both demo projects demonstrate how our two new FluentTQL extensions should be used, as well as their old counterpart alternatives. After understanding the respective FluentTQL elements, the participants have to complete the specifications in the coding exercises and use both language alternatives for the two given tasks. For each task, the participant only had to fill out a part of the FluentTQL specification related to the extensions. This is done to not overwhelm the participant with unnecessary information and language elements that are not evaluated. Following the same logic, the code examples and exercises do not contain complex taint-related vulnerabilities but rather a handful of taint flow participants to understand the core of the problem and later complete the respective specifications.

4.1.3 Questionnaires

Participants answered two usability questionnaires that were related to both tasks that were completed, as covered in the previous Subsection 4.1.2. For the first extension, each participant finished both exercises and then answered the first questionnaire. The same course of action was followed for the exercises related to the second extension, where each participant filled the second usability questionnaire after completing the second task. The questions of the questionnaires were related to the old and new options that the participant had used for each respective feature. The same questionnaire was therefore filled twice by the participant, one time for the first task and an additional time for the second task; only the names of the questionnaires were different: Questionnaire 1 and Questionnaire 2, respectively. Similar to the informational survey covered in Subsection 4.1.1, both usability questionnaires were built by using the Google Forms tool since it was easier to properly format the questionnaires and store the answers in a cloud platform for post-processing. We chose this tool because it is pretty popular and provides an understandable interface to record the answers. Moreover, the participants could change their answers more easily if they changed their minds. If the questionnaires were presented physically on paper, that would make the data gathering messier. One can find the first questionnaire under the first link and the second questionnaire under the second link. The questions that a questionnaire contained are as follows:

1. I found the first option easy to use. (Answer: 1-5)
2. I found the first option unnecessarily complex. (Answer: 1-5)

3. I imagine that most people would learn to use the first option quickly. (Answer: 1-5)
4. I would need the support of an expert to use the first option. (Answer: 1-5)
5. I found the first option to be clear and simple. (Answer: 1-5)
6. I think that the first option is prone to doing typos during specification. (Answer: 1-5)
7. I felt confident using the first option. (Answer: 1-5)
8. I needed to learn a lot of things before I could get going with the first option. (Answer: 1-5)
9. I think that I would like to use the first option frequently. (Answer: 1-5)
10. I found the first option frustrating to use. (Answer: 1-5)
11. I found the second option easy to use. (Answer: 1-5)
12. I found the second option unnecessarily complex. (Answer: 1-5)
13. I imagine that most people would learn to use the second option quickly. (Answer: 1-5)
14. I would need the support of an expert to use the second option. (Answer: 1-5)
15. I found the second option to be clear and simple. (Answer: 1-5)
16. I think that the second option is prone to doing typos during specification. (Answer: 1-5)
17. I felt confident using the second option. (Answer: 1-5)
18. I needed to learn a lot of things before I could get going with the second option. (Answer: 1-5)
19. I think that I would like to use the second option frequently. (Answer: 1-5)
20. I found the second option frustrating to use. (Answer: 1-5)
21. How much would you recommend the first option over the second option to a friend for the type of task you performed? (Answer: 0-10)
22. How much would you recommend the second option over the first option to a friend for the type of task you performed? (Answer: 0-10)
23. Are there syntax elements from the first option that are not understandable and that you would change? (Feedback)
24. Are there syntax elements from the second option that are not understandable and that you would change? (Feedback)

On the questionnaire above, the first 20 questions were based on the System Usability Scale (SUS) [B$^+$96], with some minor tweaks related to the data that we want to gather on our features, as was the case with questions 5 and 6. The first ten questions were for the first option used, and the following ten questions were for the second one. The participants answered on a scale from 1 to 5, from strongly disagree to strongly agree. Furthermore, the following two questions, questions 21 and 22, were from Net Promoter (NP) [Rei03] scale that measured the participant's experience towards the respective options and how likely they were to recommend one over the other. Both questions were answered on a scale from 0 to 10. Lastly, the last two questions were open ones and were meant for the participant to give additional feedback and improvements that could be done to the represented options in the DSL.

4.1.4 Setup

The user study was conducted as a set of video-conference meetings and recorded on audio in those cases where participants gave their permission. Only three out of six participants accepted to be recorded; in the other remaining cases, we took a written protocol of the sessions. Each user study session comprised of a practical part where the participants completed the given FluentTQL specifications for the two related extensions, as explained in Subsection 4.1.2. After completing the respective tasks, each participant answered the respective questionnaire that was covered in Subsection 4.1.3, which helped us to gather metrics and to evaluate our two extensions. Each participant first completed a given task and then answered the respective questionnaire so that we could record fresher answers; this course of action was followed for both extensions in our study. We chose this approach because we feared that an extensive questionnaire filled at the end of both tasks, which would cover all four specifications of the two extensions, would have probably tired the participants and endangered the truthfulness of their answers.

Each session initially started with the participant filling the informational survey that was covered in Subsection 4.1.1. Before continuing the practical part, the participants had to download the user study project, as mentioned in Subsection 4.1.2; then, its three Maven projects could be imported into an IDE of choice. Each participant was informed about the outline of the user study where they would need to complete two tasks, and each task would need to be finished by using two alternative options.

Afterward, we continued with the first task, which covered method signature options in the FluentTQL language. This task was performed in order for the participants to try our first extension of writing method signatures as `MethodSignature` object and also use the old option of doing the exact specification through `String` objects. The steps for using our first extension and completing the first task are presented in the list below:

- The code example of the first demo project was covered.
- The specification related to the first code example was demonstrated, and specific attention was given to how signatures were specified through the `String` object in FluentTQL.
- Then, the coding exercise of the first demo project was demonstrated, and its taint-related vulnerability was explained.
- The FluentTQL specification, which was related to the first code exercise, was displayed, and the participant had to complete it by using the `String` object for writing a method signature; this concluded the first half of the first task.
- The code example of the first demo project was again mentioned, and its other respective FluentTQL specification was also explained, and specific attention was given to how method signatures were specified by using the `MethodSignature` object.
- Next, the code exercise of the first demo project was once more announced, and its other specification was presented, where the participant had to complete the signatures by using the `MethodSignature` object; this second exercise marked the completion of the first task.
- Upon completion of the first task, the participant had used both `String` and `MethodSignature` objects for specifying method signatures in the language.
- Lastly, the participant proceeded to fill the first questionnaire.

Moreover, the same steps were also taken for completing the second and final task of the user study. The second task was related to the usage of our second extension, where the participants

tried to specify the signatures of multiple overloaded methods through the use of the `ANY` keyword. They also utilized the old manner of specifying the overloaded method signatures to compare both alternatives. After the second task, the participant was then directed to fill the second usability questionnaire, as covered in Subsection 4.1.3.

To conclude how a session was undertaken, our screen was shared with the participants when demonstrating all code examples and their respective FluentTQL specifications, which used the old and new alternatives. We also demonstrated the vulnerabilities in the coding exercises and then invited them to solve the respective specifications. Everything was fully explained before continuing to the exercises that the participants had to complete. We made sure that the participants were ready to take over the completion of the given FluentTQL specifications. The participants shared their screens during the exercises so we could answer possible questions during the process. They were not aided by us in any particular way to complete the tasks. While using the language alternatives, the participants had no information on whether they were our extension or existed previously in the DSL. The order in which the alternatives were presented to the participants was interleaved to account for the learning curve. In some cases, participants started the tasks by using the respective new extensions and in other cases with the utilization of the old alternatives. This was done to receive meaningful results since people tend to solve a task faster a second time. The second time that the participant would write a given specification related to a coding exercise, whether using the old or new language feature, would write it faster than when writing it for the first time since it would utilize the gathered knowledge from the first completed specifications. All exercises that the participants completed were timed to see how fast each participant could complete them using the old and new language alternatives. In each usability questionnaire, the participant could easily distinguish between the presented alternatives since they had already solved the respective FluentTQL specifications, in the previous tasks.

4.1.5 Results

Due to the low number of participants, the collected data from both usability questionnaires were processed manually. We transferred all the participants' answers from the forms into a spreadsheet, which offers many usable and straightforward options for processing the usability metrics. From their answers, we were able to answer both research questions related to each feature, which were covered at the beginning of Section 4.1. We answered these research questions since we collected the usability scores from the SUS-related questions and the NP scores from the NP-related questions. Moreover, we could also add some more qualitative data from the open questions in each questionnaire. Lastly, we also recorded the necessary time for the completion of each specification on each task.

In Tables 4.2, 4.3, 4.4, and 4.5 that are drawn below, we have presented the results that were collected regarding both tasks of the user study, when participants used the respective old and new features of FluentTQL. In each table, the participants' answers were processed. The usability scores of the System Usability Scale and Net Promoter were calculated, together with the necessary time needed for the participants to complete the given specifications. The SUS scores were calculated from the 10 SUS-related questions that the participant answered in their respective questionnaires. The same went for the NP scores, which were calculated from the NP-related questions respective to the evaluated language features in each questionnaire. In Table 4.6, we have presented the aggregated metrics from each represented table.

Table 4.2 represents the calculated metrics from the answers of the participants in the first questionnaire, which were about the old option of `String` object that was used for writing method signatures in the first task in the user study. As we notice, the SUS score ranges from 30 to 90 among participants, on a scale between 0 and 100. A score of 68 is above average,

and we see that 3 out of 6 participants have liked the usability of the old option for specifying method signatures. The other half has rated it below average, which means there are mixed feelings regarding the usability of the old option, as is initially shown by the large gap between the lowest and highest scores among participants. Similarly, there is a large gap between the NP scores between the participants' answers. They range from 0, which is the lowest, up to 9, which is the highest. The NP scale ranges from 0 to 10. In here, 3 Detractors have scored the old FluentTQL feature from 0 to 6 and do not recommend it. Then, there are 2 Passives with a score from 7 to 8 who are satisfied with the DSL option but are not firm believers in it. Also, there is only 1 Promoter with a score between 9 and 10, which is happy with the feature and would further promote it to a friend. Comparable to the SUS results, the participants also show mixed feelings about promoting the old feature for specifying method signatures in FluentTQL. Lastly, participants have mostly needed around 1 to 2 minutes for completing this specification exercise, with only the first participant requiring north of 4 minutes for its completion.

	Participant Data					
Questions	**P. 1**	**P. 2**	**P. 3**	**P. 4**	**P. 5**	**P. 6**
SUS (0 - 100)	30	80	67,5	90	90	65
NP (0 - 10)	0 (DET)	9 (PRO)	1 (DET)	4 (DET)	8 (PAS)	7 (PAS)
Time (min:sec)	04:37	00:47	00:55	01:39	01:30	02:01

Table 4.2: Results of the *first* questionnaire related to the *old* option

In this same manner, Table 4.3 shows the results from the answers of the participants in the first questionnaire related to the usage of our new extension, which is about the usage of `MethodSignature` object in the specification of method signatures. The SUS score fluctuates from 67,5 to 97,5 among participants, which shows a lower gap than the old alternative. We notice that almost all participants have liked the usability of the new option, with only one scoring it 67,5. This is very near to 68 and almost above average like the other results. This means that the participants quite liked the usability of our extension. Similarly, there are better NP scores than the old option; there is a lower gap in their answers, which runs from 3 to 10. The number of Detractors is the same as the one in the old feature, but there are two more Promoters than its alternative and no Passives. Lastly, almost all participants have taken more than 2 minutes to complete this specification exercise, with only one participant requiring south of 1 minute for finishing it. This is expected as there is more typing involved in the writing of the `MethodSignature` initialization, compared to the `String` one.

	Participant Data					
Questions	**P. 1**	**P. 2**	**P. 3**	**P. 4**	**P. 5**	**P. 6**
SUS (0 - 100)	90	77,5	90	97,5	67,5	92,5
NP (0 - 10)	10 (PRO)	3 (DET)	10 (PRO)	6 (DET)	5 (DET)	9 (PRO)
Time (min:sec)	02:19	02:05	00:26	02:26	02:10	02:51

Table 4.3: Results of the *first* questionnaire related to the *new* option

Table 4.4 below conveys half of the results from the second questionnaire, which are about the old option of specifying all overloaded methods participating as flow elements in a taint flow. These methods' signatures were specified one by one as `String` objects in the second task of the user study. In the third row of the table, the SUS scores range from 42,5 to 80 among participants. We notice that half of the participants' score is below the average score of 68 and the other half above that same value. We can determine that the participants are indecisive

about the usability of the old option, similar to how it also happened with the old feature in the first task. There also exists a considerable difference between the lowest and highest scores among participants, as was previously mentioned. Moreover, there are unimpressive results regarding the NP scores that were collected from the participants' answers. All participants have scored the old option between 0 and 2, on a scale from 0 to 10. This means that all of them are Detractors and do not recommend using this feature for the given task. This is expected as much more code was written than when using the corresponding newer extension. This is also manifested in the needed time for completing the exercise, where most of the participants have required more than 2 minutes to do so, with only one participant needing less than 1 minute and a half.

	Participant Data					
Questions	**P. 1**	**P. 2**	**P. 3**	**P. 4**	**P. 5**	**P. 6**
SUS (0 - 100)	42,5	72,5	82,5	80	42,5	47,5
NP (0 - 10)	0 (DET)	2 (DET)	0 (DET)	0 (DET)	2 (DET)	0 (DET)
Time (min:sec)	02:33	02:10	01:28	02:26	02:59	02:29

Table 4.4: Results of the *second* questionnaire related to the *old* option

Following the same idea, Table 4.5 shows the results from the answers of the participants in the second questionnaire about the usage of our new extension, which is about the usage of the keyword `ANY` when specifying the parameters for overloaded methods that participate in a taint-related vulnerability. The third row of the table shows that the SUS scores move slightly from 90 to 100 among participants, which is way better than the more significant gap present in its respective older alternative. We can conclude that all participants enjoyed the new option regarding the given exercise since all have scored it 90 or above in a scale from 90 to 100. Similarly, we have obtained almost perfect results for the NP scores compared to the old feature; 5 out of 6 participants have given it a score of 10, with only one rating it at 9. From the NP scores, we see that all participants are Promoters of our new extension, and there are no Detractors or Passives. This is a huge gap compared to the old option, where all participants do not recommend it for the given task. Lastly, all participants have taken less than 1 minute for the completion of this specification exercise. This is way faster than the respective old alternative, where participants usually took more than 2 minutes to complete the given specification. It is expected since the writing of the method signature could be achieved in one statement through the new extension.

	Participant Data					
Questions	**P. 1**	**P. 2**	**P. 3**	**P. 4**	**P. 5**	**P. 6**
SUS (0 - 100)	92,5	92,5	95	100	90	97,5
NP (0 - 10)	10 (PRO)	9 (PRO)	10 (PRO)	10 (PRO)	10 (PRO)	10 (PRO)
Time (min:sec)	00:37	00:31	00:40	00:49	00:34	00:47

Table 4.5: Results of the *second* questionnaire related to the *new* option

Table 4.6 shows the aggregated results that were collected from the questionnaires and that were individually shown for all evaluated options of the language, as covered so far in the previous tables. The subsequent results show the aggregated usability metrics of SUS and NP scores for all extensions and the mean time length needed to finish each given specification by using the given language features. These metrics helped us answer the research questions raised at the beginning of this user study as the following paragraphs for both extensions iterate.

Regarding the first extension of using `MethodSignature` object for specifying signatures in DSL, aggregated results show that this option is pretty usable as its SUS score of 85,8 represents, where a score of 68 is above average. Moreover, this new extension in FluentTQL is better compared to the old one of using `String` object for signature specification because it has a more significant SUS score, compared to that of 70,4 of the old one; the old alternative still achieves an above-average score. Similarly, the new extension is better than the old one because it achieves an NP score of 0% compared to that of -33% of the old one, on a scale of -100% to 100%. This means that our new extension is neither promoted nor unwanted, compared to the old one which leans to being not recommended. Overall results show that both alternatives are preferred and usable among each other, but our new extension still has the edge. Furthermore, overall results show that the mean time length for completing the specification with the new extension over the old alternative is more extensive, as 2 minutes and 2 seconds compared to 1 minute and 54 seconds shows. This is expected since there is a little more writing to be done when specifying signatures through `MethodSignature`. However, qualitative data that was collected from the open questions show that two participants liked the new extension as the method completion during the specification of `MethodSignature` helped them to better keep track of the parts of the signature that they were writing, like the return type or method name, and also the order on which they should be written. They also added that the new extension helped them avoid typos during specification since they did not need to remember to put the `:` symbol after writing the class where the method is located. Both of these comments were the main objectives on why this new extension was proposed and implemented. Another participant liked the old `String` alternative better because it was easier to write the signatures, in comparison to the `MethodSignature` initialization, which felt that would take more time and patience. However, the same participant mentioned that the new extension would make more sense for inexperienced users of FluentTQL as they would be better guided when writing signatures.

	Aggregated Data			
	First Task		**Second Task**	
Metrics	**Old Option**	**New Option**	**Old Option**	**New Option**
SUS (0 - 100)	70,4	85,8	61,3	94,6
NP (-100% - 100%)	-33	0	-100	100
Time (min:sec)	01:54	02:02	02:20	00:39

Table 4.6: Aggregated results of the *old* and *new* alternatives from both tasks

About our second FluentTQL extension of using a keyword for a faster specification, aggregated results of the table above show that this option almost has a perfect SUS score, with its score being 94,6. This shows that participants found our language extension extremely useful for solving the given specification in the user study. Moreover, our extension is also better than its alternative that specifies all overloaded methods one by one. It enjoys a considerably larger score than the old option, which only achieves a below-average SUS score of 61,3. Similarly, the new extension can be considered better than the old one because it achieves a perfect NP score of 100% compared to -100% of the old one, on a scale of -100% to 100%. This shows that our new extension is promoted by all participants, whereas no one promoted the old alternative. These results are expected since the participants did not need to write much code when using our new extension to specify the signatures of overloaded methods, as it happened when they used the old alternative. The mean time length also supports such a big gap between the usability of both alternatives. As seen from the table above, participants completed the specification with the new extension in 39 seconds on average. In contrast, they needed 2 minutes and 20 seconds

to finish the specification with the old alternative. Moreover, qualitative data that was gathered from the open questions showed that participants liked the usage of the new extension as it shortened the time for specifying overloaded methods through the usage of the `ANY` keyword. It also helped them in committing fewer typos as they did not need to write the arguments of the methods in the signature. One participant found it unnecessary to use the older language alternative for the given task due to the more significant number of lines needed when specifying overloaded signatures with the new extension. These comments were one of the driving objectives for improving upon the older alternative and developing the new extension.

To sum up the evaluation of the two extensions, participants found the first extension of specifying signatures through `MethodSignature` as useful and entirely usable by being even better than the old alternative of writing signatures through `String` objects. Also, they rated the second extension of utilizing a keyword when specifying overloaded methods as exceptionally useful and in opposite ends compared to the older alternative of writing the overloaded signatures, one by one. More detailed information concerning the ones that are represented in the tables above can be found in a spreadsheet file under the following link. Lastly, these evaluation results do not have a statistical strength since they were conducted with only 6 participants but are still meaningful since the participants were chosen to be knowledgeable in static analysis and interested in DSLs.

4.2 Experimental Approach

An experimental approach was used for evaluating our third extension, which is about the entry point feature that makes it possible in the FluentTQL language to associate entry points to specified taint flow queries, as covered in Section 3.3. Entry points can be a single method or a class, where in the latter case, all its public methods are considered entry points. This extension incorporates valuable additions in the FluentTQL language and the taint analysis of the SecuCheck tool. In comparison to the evaluation of the first two extensions done through a user study, we did not evaluate the usability of the entry point extension due to the time constraints that we had for this thesis. Since our third extension includes significant changes to the taint analysis, we evaluated the latter's performance through experiments. Before the start of this thesis, there already was a demo application that contained well-known taint-related vulnerabilities and respective FluentTQL specifications which showcased such vulnerabilities. We built another set of specifications that contained related entry points for the taint flow queries, which helped the analysis find the vulnerabilities faster. To determine its performance, the taint analysis was initialized with both types of FluentTQL specifications, with and without specified entry points in FluentTQL. Through the results of this experimental approach, we answered the following research questions:

1. Do we get the same results from the taint analysis when the entry point extension is used in the specifications?

2. Does the usage of entry points increase the performance of the analysis?

The following subsections contain information on the FluentTQL specifications used for each run of the taint analysis, the overall setup of each experiment when running the analysis with the given specifications, and the interpretation of the results gathered from the taint analysis runs.

4.2.1 Specifications

We built some specifications which utilized our entry point extension for finding vulnerabilities in a demo project. Such project was already present in our **secucheck-catalog** remote repository, which holds code examples and respective specifications in order to showcase various features of our DSL and the functionality of our taint analysis tool. The demo project is a to-do list Spring [Spr] application which is used to display many famous taint-related vulnerabilities. Related FluentTQL specifications were also present in the repository, which could be given to our SecuCheck tool to find the various vulnerabilities in the demo project. Such old specifications contained taint flow queries which iterated the whole application in search of the specified flows by starting the search in all possible entry points that the application had. In total, there were eight *old* FluentTQL specifications which did not contain the usage of our entry point extension in the DSL. In order to evaluate our extension, we extended the **secucheck-catalog** repository with eight *new* FluentTQL specifications which utilized our third extension. These new specifications were almost identical to the older ones, with the only difference being that the new ones contained entry points in each of the taint flow queries. This was, in turn, used by the SecuCheck tool to query the parts of the to-do list application dictated by the specified entry points. Both types of FluentTQL specifications, old and new, were built to find the same taint-related vulnerabilities, which are listed in the list below:

1. CWE20: Improper Input Validation
2. CWE22: Path Traversal
3. CWE78: OS Command Injection
4. CWE79: Cross-Site Scripting
5. CWE89: SQL Injection
6. CWE311: Missing Encryption
7. CWE601: Open Redirect
8. CWE643: XPath Injection

The list above contains the names of the vulnerabilities that were present in the demo project. The same naming was used for the old and new FluentTQL specifications. Each set of set specifications was collected into two different packages: one general package named *"specifications"* which hosted the old specifications and another one named *"specifications WithEntryPoints"* that contained our newly created FluentTQL specification that used our entry points extension; both packages can be found in the following link. Similarly, the demo project which was analyzed during our experiments with the given specifications can be checked under the following link.

The listings below show a code excerpt in the demo project which contains a CWE20-related vulnerability and two FluentTQL specifications, one from the old specifications and one from the new ones, which are used by SecuCheck to find the vulnerability in the given code.

```
1 package de.fraunhofer.iem.secucheck.todolist.controllers;
2 // imports
3
4 @Controller
```

```
public class LoginController {

  // field and methods

  // SOURCE
  @RequestMapping(value = "/signup", method = RequestMethod.POST)
    public ModelAndView createNewUser(@Valid User user, BindingResult
            bindingResult, HttpServletRequest request, HttpServletResponse
            response) {

        ModelAndView modelAndView = new ModelAndView();
        User userExists = userService.findUserByEmail(user.getEmail());
        if (userExists == null)
        {
            // SINK
            userService.saveUserDefault(user);
            modelAndView.addObject("successMessage", "User registered");
            modelAndView.addObject("user", new User());
            modelAndView.setViewName("authentication/signup");
        } else {
            bindingResult.rejectValue("email", "error.user",
                    "User already registered");
        }
        modelAndView.setViewName("authentication/signup");
        return modelAndView;
    }

    // other code
}
```

Listing 4.9: Excerpt from demo project containing a CWE20-related vulnerability

```
// SOURCE
Method sourceMethod = new MethodConfigurator(
            "de.fraunhofer.iem.secucheck.todolist.controllers.LoginController:"
            +"org.springframework.web.servlet.ModelAndView createNewUser("
            +"de.fraunhofer.iem.secucheck.todolist.model.User," +
            +"org.springframework.validation.BindingResult," +
            +"javax.servlet.http.HttpServletRequest," +
            +"javax.servlet.http.HttpServletResponse)")
        .out().param(0)
        .configure();

// SINK
Method sinkMethod = new MethodConfigurator(
            "de.fraunhofer.iem.secucheck.todolist.service.UserService: "
            +"void saveUserDefault("
            +"de.fraunhofer.iem.secucheck.todolist.model.User)")
        .in().param(0)
        .configure();

// TAINT FLOW QUERY
TaintFlowQuery myTF = new TaintFlowQueryBuilder("CWE20_ImproperInputValidation")
        .from(sourceMethod)
        .to(sinkMethod)
        .report("CWE-20 detected: Improper Input Validation from 'User user'")
        .at(LOCATION.SOURCEANDSINK)
        .build();
```

Listing 4.10: Specification without entry point related to listing 4.9

```
1  // SOURCE
2  Method sourceMethod = new MethodConfigurator(
3              "de.fraunhofer.iem.secucheck.todolist.controllers.LoginController:"
4              +"org.springframework.web.servlet.ModelAndView createNewUser("
5              +"de.fraunhofer.iem.secucheck.todolist.model.User," +
6              +"org.springframework.validation.BindingResult," +
7              +"javax.servlet.http.HttpServletRequest," +
8              +"javax.servlet.http.HttpServletResponse)")
9          .out().param(0)
10         .configure();
11
12 // SINK
13 Method sinkMethod = new MethodConfigurator(
14             "de.fraunhofer.iem.secucheck.todolist.service.UserService: "
15             +"void saveUserDefault("
16             +"de.fraunhofer.iem.secucheck.todolist.model.User)")
17         .in().param(0)
18         .configure();
19
20 // ENTRY POINT
21 String methodEntryPointName =
              "de.fraunhofer.iem.secucheck.todolist.controllers.LoginController: "
22       + "org.springframework.web.servlet.ModelAndView "
23       + "createNewUser "
24       + "(de.fraunhofer.iem.secucheck.todolist.model.User, "
25       + "org.springframework.validation.BindingResult, "
26       +"javax.servlet.http.HttpServletRequest, "
27       + "javax.servlet.http.HttpServletResponse)";
28 MethodEntryPoint entryPoint = new MethodEntryPoint(methodEntryPointName);
29 List<EntryPoint> entryPoints = Arrays.asList(entryPoint);
30
31 // TAINT FLOW QUERY
32 TaintFlowQuery myTF = new TaintFlowQueryBuilder("CWE20_ImproperInputValidation")
33         .atOnlyDSLEntryPoints(entryPoints)
34         .from(sourceMethod)
35         .to(sinkMethod)
36         .report("CWE-20 detected: Improper Input Validation from 'User user'")
37         .at(LOCATION.SOURCEANDSINK)
38         .build();
```

Listing 4.11: Specification with entry point related to listing 4.9

Listing 4.9 above demonstrates a code excerpt from our to-do list application and contains an Improper Input Validation vulnerability in it. The vulnerability is a famous one on CWE's [Com] website and has the id of CWE20. The method `createNewUser` in line 11 is a REST controller in the terminology of Spring Boot framework and is also a source because it accepts user input in its first parameter that is named `user`. This parameter may be tainted and flows into the sink method of `saveUserDefault` in line 18, in which case the vulnerability would be manifested. Moreover, Listing 4.10 above demonstrates one of the eight old specifications that were prepared for finding the vulnerabilities in our project and is used to find the vulnerability in the listed code excerpt. This specification contains the source method (lines 2-10), the sink method (lines 13-18), and the respective taint flow query (lines 21-26). The old specification is given as input to the SecuCheck tool. The analysis checks all possible entry points of our demo project to find the specified taint queries since no specific entry points are given in the FluentTQL specification. Lastly, Listing 4.11 above is one of the eight newly created FluentTQL specifications and is also used to find the same CWE20 vulnerability that its old counterpart in Listing 4.10 is specified to find. The new specification almost has the same structure as the old specification. The only difference is that we have written specific entry points (lines 21-29)

and added them into the given taint flow query (line 33). For these all-new specifications, like the one in Listing 4.11, we have only specified method entry points and not class entry points because we wanted to have a small entry point size so that the analysis would finish faster. Also, we have only added the respective entry points where each vulnerability is located for each new specification. For example, in our new specification above that tries to find the CWE20-related vulnerability in the project, we have only set one entry point, which is the `createNewUser` method because this is where the Improper Input Validation vulnerability is manifested. The analysis would only look at this specified entry point to find the specified vulnerability and nowhere else. We did so as we wanted to experiment with the best-case scenario when using our entry point extension and see how much faster the taint analysis would be.

4.2.2 Setup

The FluentTQL specifications that were covered in the above Subsection of 4.2.1 were given one after the other to the SecuCheck tool, and the taint analysis was run in my machine. The latter only had 8GB of memory. Only the taint analysis was run during the experiments, so we could get the best performance from the taint analysis since all hardware resources were given to this undertaking. The demo project was opened in the Eclipse IDE. The setup page for the analysis was opened in the Google Chrome browser, where we had also imported all the FluentTQL specifications. The analysis always started with only one FluentTQL specification, taken from old or new specifications. This was done in order to record the analysis results for each specification. The list below shows the steps that we performed for recording the results of the analysis when starting it with a particular specification:

- Choose a FluentTQL specification from the list of old and new specifications in the browser.
- Run the taint analysis from the browser for only the chosen specification.
- Check the analysis results in the IDE and record all analysis results data in a prepared spreadsheet.
- Repeat the same steps until now so that we have ten analysis runs in total for each specification.

The steps listed above were conducted for each specification, and the results were recorded in detail in a spreadsheet. The metrics that were documented are covered in detail in the following subsection.

4.2.3 Results

Results were recorded in a spreadsheet and processed there as it offered many valuable features for easier processing. Through the data collected from the experiments, we were able to answer the research questions that were listed at Section 4.2.

Tables 4.7, 4.8, 4.9, and 4.10 show the results that were recorded from the experiments. The results are shown in multiple tables because they could not be fitted into only one general table on this page. Each table contains the analysis results when started with two sets of specifications, old and new. Table 4.7 contains the results of the CWE20 and CWE22 sets, Table 4.8 hosts the results of the CWE78 and CWE79 sets, Table 4.9 accommodates the results of the CWE89 and CWE311 sets, and Table 4.10 demonstrates the results of the CWE601 and CWE643 sets. The old and new specifications for each set are demonstrated side by side to compare them better. The next sentences cover detailed information on the metrics that are present in each table. The first metric of *Runtime* represents the time that the analysis took to finish when started with a

given specification; the time is shown in seconds with two decimal places for better readability in the tables, although the actual results are available with four decimal places. The next metric is *Improvement* and shows how much faster the analysis was when run with the new specification, compared to the respective old one in the same set; the improvement is shown in percentage. Furthermore, the metric of *Queries* shows the number of taint flow queries that were written in the respective FluentTQL specification. In contrast, the metric of *Findings* indicates the number of taint flows, from sources to sinks, that were found in the analyzed application when queried with the given queries in the specifications. For both specifications in each set, old and new, these metrics did not change, which is why they are merged into one row. Moreover, *EP size* describes the entry point size as the number of methods in where the analysis started its work in search of the specified taint queries. Lastly, *CG size* shows the call graph size that was constructed from the given entry points, and the size is counted as the number of edges; we can notice from the tables that the more entry points we have, the larger the constructed call graph will be.

Metrics	**CWE20**		**CWE22**	
	CWE20 (Old)	CWE20 (New)	CWE22 (Old)	CWE22 (New)
Runtime (sec)	1,52	0,63	1,60	1,00
Improvement (%)	**59**		**37**	
Queries (#flows)	**1**		**1**	
Findings (#flows)	**1**		**1**	
EP size (#methods)	144	1	144	1
CG size (#edges)	1857	18	1857	836

Table 4.7: Results of the analysis on CWE20 and CWE22 specifications

Metrics	**CWE78**		**CWE79**	
	CWE78 (Old)	CWE78 (New)	CWE79 (Old)	CWE79 (New)
Runtime (sec)	2,67	1,72	1,57	0,61
Improvement (%)	**36**		**62**	
Queries (#flows)	**2**		**1**	
Findings (#flows)	**2**		**1**	
EP size (#methods)	144	1	144	1
CG size (#edges)	1857	836	1857	18

Table 4.8: Results of the analysis on CWE78 and CWE79 specifications

Metrics	**CWE89**		**CWE311**	
	CWE89 (Old)	CWE89 (New)	CWE311 (Old)	CWE311 (New)
Runtime (sec)	3,61	2,45	1,60	1,00
Improvement (%)	**32**		**38**	
Queries (#flows)	**3**		**1**	
Findings (#flows)	**3**		**2**	
EP size (#methods)	144	2	144	1
CG size (#edges)	1857	1006	1857	836

Table 4.9: Results of the analysis on CWE89 and CWE311 specifications

Metrics	**CWE601**		**CWE643**	
	CWE601 (Old)	CWE601 (New)	CWE643 (Old)	CWE643 (New)
Runtime (sec)	3,58	2,34	2,55	1,43
Improvement (%)	**35**		**44**	
Queries (#flows)	**3**		**2**	
Findings (#flows)	**3**		**1**	
EP size (#methods)	144	3	144	1
CG size (#edges)	1857	972	1857	877

Table 4.10: Results of the analysis on CWE601 and CWE643 specifications

Next, we comment on the meaning of the results of the experiments and answers that we can give to the research questions that were raised in Section 4.2. The metric of *Findings* indicates how many tainted flows the analysis found when queried with a given specification. In old and new specifications of CWE20, the analysis finds only one taint flow when queries with this specification pair. The same fashion follows in other sets, where the taint analysis finds an equal number of taint flows in the analyzed application when queries with the given pair of specifications. The latter information answers the first research question. It shows that the analysis gives the same results in taint flows found, be it when started with old specifications that do not contain entry points specified in the DSL or with our new specifications that use our entry point extension to specify entry points for the analysis. It is also essential to clarify why the analysis finds the specified taint flows in the analyzed application for both pairs of specifications in each set. The new specifications that we built are almost identical to the old counterparts. The only addition is the entry points added to each taint flow query. The queries only contain the entry points where the vulnerabilities that we are looking for will be found. This is why we find the same vulnerabilities in the old and new specifications in each set. We would probably not find all taint flows if we would specify other entry points in the new specifications. This happens because the analysis only started at the specified entry points in the new specifications and did not look into the whole entry points as in the old specifications. By default, if no entry points were specified in the DSL, the analysis would search into all possible entry points, which are all the public methods in an analyzed application. Moreover, the analyzed demo project was built using the Spring Boot framework, and we noticed that all entry points that we set in all of the new specifications were, in fact, REST controller methods. These methods work with and compute user input, which hints at why the vulnerabilities were found in such methods. These results show that entry points for other Spring Boot applications can be set in such REST controller methods in cases when a user of FluentTQL wants to use our entry point extension and does not know which entry points to specify.

After positively answering the first research question, we go to the next and last one. The second question worries whether the taint analysis has better performance when queried with FluentTQL specifications that include the usage of entry points. In all tables above, we can examine the *Improvement* metric, which shows that there is always an improvement when the analysis is run with the new specifications that contain entry points, compared to the respective old ones in each set. This answers the second research question and shows that we indeed have improvements in the performance of the analysis when entry points are used in the DSL since the analysis completes faster in such cases when compared to the old specifications. Results show that the analysis runtime improves in the range of 32% to 62% across the new specifications. The mean runtime improvement in the taint analysis is 43 percent, which means that the analysis was almost two times faster when entry points were correctly specified in the language. The reason why the analysis finished quicker through the new specifications was because of the number of

entry points that were set, as we see from the *EP size* metric. The analysis only searched in the methods we set as entry points in the DSL compared to all possible entry points that the demo project had. The tables show that in all the old specifications, the analysis started its search in 144 entry point methods, which increased the runtime since SecuCheck had to look in more places, compared to a maximum of 3 entry point methods that happened in the new specifications. The *EP size* metric directly affected the call graph size that was constructed, which was counted through a number of edges in the *CG size* metric. In the old specifications, the analysis searched in a big call graph which contained 1857 edges; this was constructed from the 144 entry points that were given by default. The new specifications made the call graph size smaller because it was constructed from way fewer entry points. This is one of the main motives that the analysis finished faster when queried with the new specifications since it had to search in a smaller call graph for the specified taint flows. If we compare the results of the new CWE20 with the new CWE22, in the case of new CWE20, the analysis finished in only 0,63 seconds, whereas in the new CWE22 in 1 second; the analysis finished faster in the case of CWE20 because the call graph constructed through the given entry point, only had 18 edges, as compared to the one in the new CWE22 which had 836. This is why the improvement in the CWE20 set was 59% and in the CWE22 set only 37%. Another reason that affected the performance of the analysis was the number of taint flows that were set in a given specification, as demonstrated by the metric of *Queries*. The more taint flow queries there were, the more times the analysis searched the constructed call graph. For example, we compared the analyses run with the new CWE22 and the new CWE78 specifications. We had the same entry point in both specifications, which resulted in a constructed call graph containing an equal number of 836 methods. However, in the new CWE22, the analysis completed in 1 second, whereas in the new CWE78 in 1,72 seconds. The latter happened because we had two specified queries in the new CWE78 specification, compared to only 1 query in the new CWE22 specification. The number of queries affected the length of the analysis runtime, as shown in this example. To conclude, the runtime of analysis is closely related to the constructed call graph size (which is in turn closely related to the specified entry points in the DSL) and the number of specified taint flow queries.

The results of the analysis that are shown in the tables above are covered in more detail in a spreadsheet that can be found under the following link. In this experimental evaluation, we concluded that with our entry point extension usage, FluentTQL users can still find the vulnerabilities in the given program as before, but with the advantage of a shorter analysis. It is essential to specify those entry points, which would lead to detecting the vulnerabilities. Results show that methods dealing with user input are an excellent guess to specify as entry points. Moreover, the demo project that SecuCheck analyzed was small, and we still got a mean improvement of 43% in the analysis runtime when queried with specifications that contained entry points. For larger projects and real-world ones that contain hundreds of thousands of lines of code, results might be even better if the user specifies good entry points.

4.3 Sporadic Approach

A sporadic approach was used to check our fourth extension, which is regarding the ability to specify variables as flow participant in a taint query, so that we can write Hard-Coded Credentials and NULL Pointer Dereference queries in the language. Later, we use the extended analysis to find them, as covered in Section 3.4. We were not able to conduct a proper evaluation as we did with the third extension in the previous Section 4.2, due to time constraints and the difficulty of findings real-world programs that contained such vulnerabilities. However, we created a project and inserted there code excerpts that displayed the HCC and NPD vulnerabilities. Besides that,

we added the respective specifications that used our extension to model the vulnerabilities in the code excerpts. Through this sporadic approach for the fourth extension, we answered the following research questions:

1. Can we specify HCC and NPD taint flows in FluentTQL?
2. Are the vulnerabilities detected when the analysis is queried with the respective taint flows?

From many tests that examined the functionality of our extension, we concluded that users could use FluentTQL to specify new vulnerabilities, such as HCC and NPD. Moreover, we have also extended the taint analysis. It takes as input the specifications that contain the HCC and NPD vulnerabilities and does a query for them in an application. If the vulnerabilities are present in an analyzed program, the analysis will detect and present the individual results.

We created the following projects for evaluating our extension:

- **new-project**
- **new-project-specifications**

On the new **new-project**, we included different code examples that contain Hard-coded Credential and NULL Pointer Dereference vulnerabilities in it. This project can be found under the subsequent link. Also, the **new-project-specifications** contains FluentTQL specifications that demonstrate the vulnerabilities in the **new-project**, through our new extension. The project can as well be checked under the following link.

To conclude, we did not conduct a similar evaluation on the scale we did with the first three extensions. However, tests have shown that enhancements done to the language and analysis regarding this extension do correctly work. Future work may include a large-scale evaluation for this extension with real-world programs containing the HCC and NPD vulnerabilities.

5 Related Work

This section mentions and discusses other Domain-Specific Languages that contain similar features to the ones that we have extended throughout the work of this thesis. These DSLs may not be used for the same purpose of taint analysis that FluentTQL is used for but still have these features used for in different scenarios, compared to the ones in which our enhancements are utilized.

About the first extension of specifying signatures for taint flow participants in FluentTQL through the `MethodSignature` object, many other Domain-Specific Languages have similar functionalities of specifying signatures of methods for their purposes. Signatures of other DSLs are not built with the same elements that our `MethodSignature` object contains. CrySL [KSA+19] is an external DSL that comes together with the CogniCrypt tool, the same manner that FluentTQL comes paired with the SecuCheck tool. The CogniCrypt tool uses CrySL rules or specifications in order to conduct a typestate analysis. CrySL is not used for taint analysis but for specifying the correct usage of cryptographic APIs. In CrySL, users can write method signatures of a given object that conducts some cryptographic operation and specify the correct order in which such methods should be used in a program. PQL [MLL05] is another DSL that is similar to CrySL, where its specifications are as well used for typestate analyses. It is a declarative DSL in which users can specify method signatures in a comparable manner to CrySL. Unlike CrySL, PQL does not come paired with tooling of its own. Next, CxQL [Che] is another DSL that comes together with commercial tooling called Checkmarx. The DSL is object-oriented similarly to FluentTQL but is used for different types of analyses and is therefore considered general-purpose. Being general-purpose, it contains many different ways on how method signatures are specified in it, and each manner is used for a particular purpose. Lastly, CodeQL [LGT] is a DSL that also comes paired with a commercial tool called LGTM. The DSL is declarative and supports object-oriented design. In this DSL, methods signatures are specified through explicit manipulation of the code via a graph representation.

Regarding the second extension that introduced symbols such as `ANY` or underscore (`_`) for specifying arguments of methods in signatures, there is another DSL that contains a very similar functionality to this. In the CrySL [KSA+19] DSL, users can also use the underscore character for representing a placeholder; this placeholder can take the form of any argument. For example, one can specify a method like this `sum(_,_)`, meaning that the `sum` method takes two arguments of any possible type. We use the underscore in FluentTQL for the same purpose when specifying signatures. This similarity is because we already knew the functionality of underscore in CrySL and were motivated to use it in FluentTQL.

Another DSL also has similar functionality about the fourth extension of specifying Hardcoded Credentials and NULL Pointer Dereference vulnerabilities in FluentTQL through enumerations. Same as before, CrySL includes a keyword for specifying variables in the specifications that one does not want to be hardcoded. The manner how this is achieved is done, for example, through the statement `notHardCoded[password]`. This is a constraint in the DSL and means that the variable `password` must not be hardcoded. Unlike the analysis included in SecuCheck that checks whether the specified hardcoded variables are flowing into a sink method, the tool of CrySL conducts a whole-program analysis to determine whether the `password` variable is at all hardcoded when being an argument in the signatures of the specified methods in the CrySL rules.

Finally, following our fifth extension that introduces a backward taint analysis, there exist tools of many DSLs that include a backward analysis. CogniCrypt [KNR+17], the tool which uses CrySL rules to conduct a typestate analysis, conducts a backward analysis when checking whether a variable is hardcoded, as covered as an example in the previous paragraph. Many other commercial tools, like Checkmarx [Che] and LGTM [LGT], also perform a wide range of static analyses and include a backward direction in their analysis when conducting similar tasks like the one in CogniCrypt.

6 Conclusion

This last chapter concludes our master thesis and contains the main messages that we can take from the conducted work. We examined and implemented extensions to an internal DSL called FluentTQL, which is used for specifying taint flows through a Java syntax. Respective enhancements were also carried out in the taint analysis in the SecuCheck tool that was used to detect the vulnerabilities that were written through the FluentTQL specifications.

The first extension was about having a different alternative in the language for specifying the signatures of methods such as sources, sink, and propagators, that participate in taint flows. Evaluation through a user study showed that our extension achieved an above-average usability score of 85,8 on the System Usability Scale and a Net Promoter score of 0%. These scores were better than the old alternative of specifying signatures which reached a SUS score of 70,4 and an NP score of -33%.

Moreover, a second extension included enhancements to the expressivity in the specification of signatures, such as better options for specifying overloaded methods and more lenient rules about whitespace characters in the signature. The evaluation was also performed through a user study, and results showed an almost ideal SUS score of 94,6 and a perfect NP score of 100%. These metrics indicated excellent usability of our extension and high promotion among users of the language that participated in the user study.

Furthermore, the third extension introduced entry points in the language and associated them with specific taint flow queries. As a result, the constructed call graph was smaller, and the analysis only started its search in the specified entry points. The evaluation of this extension was carried out through an experimental approach, where we used specifications with and without entry points to query the analysis in SecuCheck. Experimental results showed an average increase of 43% in the analysis performance when specifications with entry points were used. An important part was also specifying the right entry points for given taint flow queries so that the analysis could find the specified taint flows in the analyzed program.

Additionally, a fourth extension was developed, which allowed specifying Hard-coded Credentials and NULL Pointer Dereference vulnerabilities in the language and using the enhanced analysis to find such specified flows. The evaluation was executed similarly to the one in the third extension. The only difference was that we did not compare our new extension with an old version of our tools since the implemented concept was not present in the tools. We chose a more sporadic approach for evaluating our extension, with a handful of code excerpts that contained the HCC and NPD vulnerabilities and respective specifications that modeled such weaknesses. Results indicated that our analysis could detect all of the HCC and NPD vulnerabilities when queried with the respective specified taint flows.

Lastly, we added a fifth extension that enhanced the SecuCheck tool's general analysis by adding the backward direction in it. Previously, the taint analysis that was implemented through the Boomerang framework only functioned in the forward fashion. With our extension, the analysis works in forward and backward directions, and each direction is chosen automatically depending on the given taint flows. This extension was manually tested, and results showed that the analysis correctly works in both directions.

Future work may include a more extensive user study for evaluating our first two extensions, with more participants and a more extensive set of tasks, in order to evaluate the usability of more features in our extensions like the use of underscores (`_`) for parameters in signatures. A more significant number of participants would also produce results that have greater statistical importance. Moreover, research can be done regarding the first extension for dynamically suggesting signatures while a user is typing them; suggestions may come from a program that will be analyzed. Regarding the third extension, future work may include a more extensive experimental set containing large and real-world programs to see the full effects when entry points are specified in the language. Regarding the specification of entry points, future work can be carried out in automatically finding relevant entry points in projects that use frameworks such as Spring Boot, since we saw that entry points in such projects are usually in REST controller methods that work with user input. On the fourth extension, future work may include a more extensive evaluation with real-world programs that contain HCC and NPD vulnerabilities. Future extensions can also include an automatic hardcoded and null pointer analysis that does not need FluentTQL specifications and checks all possible HCC and NPD weaknesses in the analyzed application. Finally, future work for the backward analysis extension may be a proper evaluation approach that works with real-world code excerpts or programs.

Bibliography

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[B+96] John Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[Bae] Baeldung. Sql injection and how to prevent it?

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*, pages 85–103. IEEE, 2007.

[Cam] G. Ann Campbell. What is taint analysis and why should i care?

[Che] Checkmarx. Checkmarx and cxql dsl.

[Com] Online Community. Common weakness enumeration.

[CWEa] CWE. 2020 cwe: Top 25 most dangerous software weaknesses.

[CWEb] CWE. Cwe-798: Use of hard-coded credentials.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[Fou] Apache Software Foundation. Maven.

[Fow] Martin Fowler. Dsl guide.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[FRBS08] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head first design patterns*. " O'Reilly Media, Inc.", 2008.

[Goo] Google. Google forms.

[KNR+17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[KSA+19] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 2019.

[LDB19] Linghui Luo, Julian Dolby, and Eric Bodden. Magpiebridge: A general approach to integrating static analyses into ides and editors (tool insights paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[LGT] LGTM. Lgtm and codeql dsl.

[Lho02] Ondrej Lhoták. Spark: A flexible points-to analysis framework for java. 2002.

[Mic] Microsoft. Language server protocol.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using pql: a program query language. *Acm Sigplan Notices*, 40(10):365–383, 2005.

[Nel] Riaan Nel. Design patterns: The builder pattern.

[Ora] Oracle. Javadocs statement.

[OWA] OWASP. Owasp top ten.

[Pis] Goran Piskachev. Specifying taint flows: The software developers' way.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125, 2014.

[Rei03] Frederick F Reichheld. The one number you need to grow. *Harvard business review*, 81(12):46–55, 2003.

[SNQDAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[Spr] Spring. Spring framework.

[VRCG+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[VRH98] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.