Master Thesis Nr. MA0213

Exploratory research on the Dataflow analysis for Kotlin

on: 14.07.2021

FRAUNHOFER INSTITUTE FOR
MECHATRONIC SYSTEMS DESIGN
Zukunftsmeile 1
D-33102 Paderborn

Contents

1 Introduction

Kotlin has been gaining high popularity over the past few years [3]. The first stable version of Kotlin was released in 2016 by JetBrains[1]. In 2017, Google announced first-class support for Kotlin in Android[2]. One of the main reasons for Kotlin's popularity is that Kotlin leads to concise code, which reduces the code smells compared to Java. Also, Kotlin is a statically typed programming language that mainly targets the Java Virtual Machine (JVM) and is fully interoperable with Java. Kotlin also provides additional features like operator overloading, data classes, extension functions, null safety, and other features. Kotlin offers a type-system that prevents the bugs like NULL Pointer Dereference (CWE-476) [2] by providing a non-nullable type.

However, Kotlin can contain security vulnerabilities like SQL injection [6], misuse of cryptography API [7], cross-site scripting (XSS) [5], etc. Listing 1.1 demonstrates the simple log injection (CWE-117) [1] in Kotlin. In Line 7, the function *logoutLogging* logs that the given user name is successfully logged out if the provided user name is valid and already logged in. If the provided user name is not valid, then the function logs (Line 9) that the given user name is invalid. In both cases, the provided user name is not sanitized before passing it into the logger. An attacker can give an invalid user name and injects malicious content into the logger—log injection.

```kotlin
1  package de.fraunhofer.iem
2
3  private val logger = KotlinLogging.logger("MyApp")
4
5  fun logoutLogging(userName: String) {
6      if (validLoggedInUserName(userName)) {
7          logger.info("User $userName is logged out successfully")
8      } else {
9          logger.info("$userName is invalid username.")
10     }
11 }
```

Listing 1.1: Simple log injection example in Kotlin.

[1]https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android

[2]https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official

Such security vulnerabilities can be detected early in the development stages using static code analysis tools that perform complex dataflow analysis such as taint analysis [12], typestate analysis [11], etc. These dataflow analyses are being used to find security vulnerabilities in Java and other programming languages for many years. There is an ample amount of static code analysis tools available for Java to find such security vulnerabilities[3]. For Kotlin, there are very few static code analysis tools available. Generally, these tools perform code smells identification, code formatter, etc. However, to our knowledge, there is no static code analysis tool available for Kotlin that performs complex dataflow analysis such as taint analysis to find security vulnerabilities like SQL injection [6], XSS [5], etc.

Most static code analysis tools developed for Java analyze the code using an Intermediate Representation (IR) generated from the Java bytecode. Since Kotlin is mainly a JVM-based programming language that compiles to the Java bytecode, theoretically, one can use static code analysis tools developed for Java to analyze Kotlin and identify the security vulnerabilities. However, Kotlin uses its compiler to compile the Kotlin source code to the Java bytecode. In other words, the Kotlin compiler's logic to generate the Java bytecode is different than the Java compiler's logic to generate the Java bytecode. This leads to the question, *can we use static code analysis tools intended for Java to analyze Kotlin, or one must "reinvent the wheel" for Kotlin?*

In this exploratory research, we will manually analyze the different Kotlin constructs to find how the IR appears for those constructs. Then we will identify the Kotlin constructs that can be analyzed using the existing static code analysis tool developed for Java. We will also identify the Kotlin constructs that need modification in the existing static code analysis tool to analyze those Kotlin constructs. For this purpose, we chose SecuCheck [4]—an existing static code analysis tool that performs taint analysis on the Java bytecode. We will propose a conceptual extension to the existing SecuCheck implementation to support Kotlin specific constructs. If the proposed solution is feasible in the given time frame of this thesis, we will implement the solution.

The rest of this proposal document describes the methodology and the plan of this exploratory research on the dataflow analysis for Kotlin. In Chapter 2, we discuss in detail why do we need to perform this research. In Chapter 3, we provide the goal of this thesis work. We present the methodology and the evaluation of this research in Chapter 4. In Chapter 5, we discuss the initial thesis structure. Finally, in Chapter 6, we present the time-plan of this thesis work.

[3]`https://github.com/codefactor-io/awesome-static-analysis`

2 Problem Description

Kotlin is a statically typed, general-purpose, and mainly a JVM-based programming language. Kotlin can also compile to the JavaScript or native code. In this research, we consider Kotlin that targets the JVM. Figure 2-1 shows how the JVM-based programming language works. The JVM understands the Java bytecode irrespective of the JVM-based programming language used to generate the valid Java bytecode. For instance, the Java source code is compiled using a Java compiler and generates the Java bytecode. The generated Java bytecode is given to the JVM that runs the program. Similarly, the Kotlin code is compiled using the Kotlin compiler and generates the Java bytecode to run on the JVM.

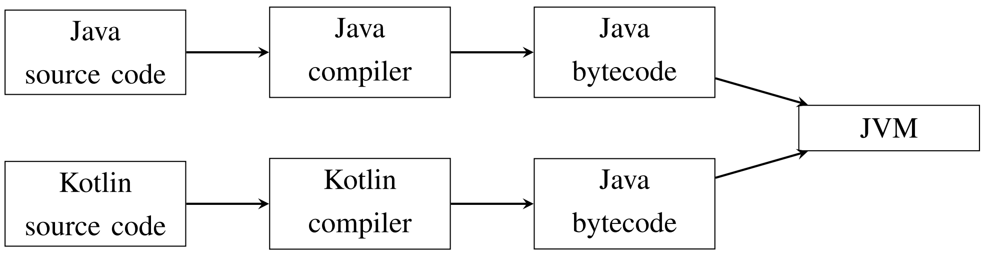

Figure 2-1: Working of JVM-based programming language.

As discussed in Chapter 1 with Listing 1.1, Kotlin can contain security vulnerabilities. Listing 1.1 uses the kotlin-logging[1] logger that is written in Kotlin. Additionally, as Kotlin is fully interoperable with Java, all the possible security vulnerabilities in Java can also be present in Kotlin. For example, one can use the Java standard logging library in Kotlin such that the Kotlin program may contain the log injection.

For Java, these security vulnerabilities are identified using static code analysis tools. For this research, we consider SecuCheck [4] that performs the taint analysis on the Java programs. The taint analysis is a type of dataflow analysis to track the taint flow—a data flow from source to sink. A source is a method that gives sensitive information: methods like *getParameter*, *getPassword*, etc. are typical source methods. A sink is a method that performs sensitive operations that may lead to a security issue or data leaks: methods like *executeQuery*, *sendEmail*, etc. are typical sink methods.

[1] `https://github.com/MicroUtils/kotlin-logging`

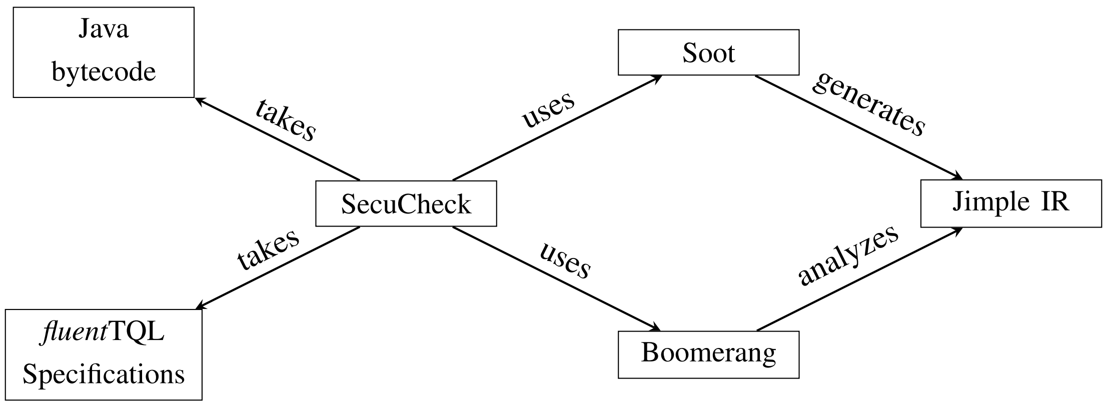

Figure 2-2: Working of SecuCheck on high level.

Figure 2-2 shows how SecuCheck works on a high level. SecuCheck uses Soot [9] framework to generate the Jimple [13] code from the Java bytecode. The generated Jimple code is the IR of the Java program that needs to be analyzed. The user can specify the taint flows using the domain-specific language called *fluent*TQL. These *fluent*TQL specifications are given to SecuCheck for the taint analysis. SecuCheck uses Boomerang [10] to find the specified taint flow by analyzing the generated Jimple code.

Since Kotlin compiles to the Java bytecode, theoretically, one can analyze Kotlin using SecuCheck that generates the Jimple code from the Java bytecode. We generated the Jimple code for a few of the basic constructs of Kotlin and compared it with the Jimple code of the respective constructs in Java. We observed that both Jimples are similar and, SecuCheck can run the analysis for Kotlin.

However, we also found that for some of the Kotlin constructs, the Jimple code is different than Java's Jimple code. For instance, companion objects, extension functions, lambda expression, etc., all of these Kotlin construct's Jimple code is different than Java's Jimple code. One of the potential problems that we observed from the Jimple code is, for some Kotlin constructs, the Kotlin compiler generates additional classes in Java bytecode compare to Java. If the user wants to specify a particular method as a source or sink in *fluent*TQL, then for the user, it is hard to get the correct method signature.

Therefore, we need this exploratory research on the dataflow analysis for Kotlin to decide whether we can extend the existing static analysis tool intended for Java to support the analysis of Kotlin, or is it better to develop the static analysis tool for Kotlin from the beginning.

3 Thesis goals

3.1 Primary Goal

The primary goal of the thesis is to conduct exploratory research on the dataflow analysis for Kotlin. This research is necessary to help the static code analysis developers to develop the static analysis tool that performs complex dataflow analysis such as taint analysis for Kotlin. We need to conduct this research systematically with the intention that other researchers can perform more related research in the future. For this purpose, we need to build microbenchmark programs and the JimpleGenerator.

Microbenchmark programs: To achieve the primary goal, we need to manually examine the different Kotlin constructs to see how the IR appears for those constructs. Taking the real-world projects for the manual examination is not feasible because of two reasons. First, the real-world project may not contain the Kotlin construct that we decide to examine and may lead to the gathering of multiple real-world projects. Second, manual examination of the Kotlin constructs in real-world projects consumes a lot of time and effort. Therefore, we need to build the microbenchmark programs for each of the Kotlin constructs that we decide to examine in this research period. We also need to build the microbenchmark programs for the respective constructs in Java to compare with the Kotlin constructs. These microbenchmark programs help the other researcher to conduct more exploratory research on dataflow analysis for Kotlin in the future.

JimpleGenerator: For the manual examination, we need to generate the Jimple code for the microbenchmark programs. As there will be many programs, it is time-consuming to generate the Jimple code for each program. Therefore, we need to build a tool to automate the generation of the Jimple code for the Java bytecode present in the given classpath. The JimpleGenerator must be easily extensible to support different pre-transformer, thus other researchers can use the tool for future research.

3.2 Secondary Goal

The secondary goal of the thesis is to extend SecuCheck to support the analysis of Kotlin. This goal is the optional goal of the thesis. Once we identify a problem for a Kotlin construct in the primary goal, if time permits, we suggest the implementation solution conceptually. Then we estimate the time to implement the suggested solution in SecuCheck to support that Kotlin construct. If the estimated time is feasible for this thesis, then we implement the solution.

4 Solution Idea

To know whether SecuCheck supports the analysis of Kotlin, we should first know whether each component of SecuCheck supports the analysis of the Kotlin constructs. Below are the main components of SecuCheck.

- ANALYSIS: This component is the core of SecuCheck, which performs the taint analysis using Boomerang [10].
- *fluent*TQL: This component is a domain-specific language to specify the taint flow for the taint analysis. SecuCheck searches for the specified taint flow in the program using Boomerang.
- IR GENERATOR: SecuCheck with Boomerang uses Soot [9] to generate the Jimple code. For the generated Jimple code, Boomerang applies its pre-transformer. For this research, we consider the Jimple code generated by Soot without the Boomerang pre-transformer as there is not much difference in the pre-transformed Jimple code.
- CALLGRAPH GENERATOR: A callgraph (CG) is the fundamental data structure that enables inter-procedural static program analysis [8]. This component generates the callgraph for the analysis.

The first three components are the main focus of this research. Due to the time limit for this thesis, we do not consider the callgraph generator component for this research as this component needs more time and effort.

4.1 Primary goal

4.1.1 Methodology

For the primary goal, below are the steps.

- Create microbenchmark programs for the different Kotlin constructs and the respective Java constructs if available.
- With the help of the JimpleGenerator tool, generate the Jimple code for the microbenchmark programs.
- Manually examine the Jimple code and check whether the construct is possible to analyze using SecuCheck or if it requires some modification in any of the SecuCheck components.

- If possible, for the given construct, introduce a simple taint flow and run the SecuCheck analysis to see whether SecuCheck finds the taint flow or not.
- After the manual inspection of the Kotlin construct's Jimple code, categorize the construct into one of the research questions mentioned in the Sub-Section 4.1.2.

EXAMPLE FOR MANUAL EXAMINATION OF JIMPLE CODE:

In Listing 1.1, the function *logoutLogging* is defined as a function without a class in the file "Logger.kt" with the package "de.fraunhofer.iem". The function takes a parameter of type *kotlin.String* and returns the *kotlin.Unit* (equivalent to *void* in Java). If a user wants to specify the function *logoutLogging* as a source or sink method in *fluent*TQL. Then the user might give the method signature for the *logoutLogging* as <*de.fraunhofer.iem.Logger: kotlin.Unit logoutLogging(kotlin.String)*>.

However, if we observe the Jimple code for the *logoutLogging*, we find that the method signature provided by the user is not valid. Listing 4.1 contains the excerpt of the Jimple code for the *logoutLogging* function. Below is the observation made from the Jimple code.

- The return type is Java's *void*, not *kotlin.Unit*.
- The parameter type is *java.lang.String*, not *kotlin.String*.
- And when we observed the generated Java bytecode for the "Logger.kt" file, it generates the class name *de.fraunhofer.iem.LoggerKt* since we do not define the function *logoutLogging* in any class.

```
public static final void logoutLogging(java.lang.String) {
    java.lang.String userName;
    ...

    userName := @parameter0: java.lang.String;
    ...
}
```

Listing 4.1: Excerpt of the Jimple code for Listing 1.1

Therefore, the correct method signature for *logoutLogging* is <*de.fraunhofer.iem.LoggerKt: void logoutLogging(java.lang.String)*>. We found such a problem in different Kotlin data types. For instance, for the *kotlin.Int* type, the Kotlin compiler uses Java's primitive *int* type in the Java bytecode. For some scenarios, the compiler uses Java's *Integer* class type instead of the primitive *int* type. If we expect the user to provide the valid method signature, then it's more effort on users and may not wish to use SecuCheck to analyze Kotlin. Therefore, we need to handle this problem in SecuCheck. This problem can be handled either in *fluent*TQL or the analysis components. However, *fluent*TQL is the best component to handle this problem because conceptually *fluent*TQL handles the programming language features. Also, if we try to solve this problem in the analysis component then, the analysis run time may increase.

Therefore, the suggested solution for the Kotlin data types is: whenever *fluent*TQL finds the Kotlin's data types in the method signature provided by the user, it has to modify the method signature to contain the valid data types based on the Java bytecode. For example, if the user provides *kotlin.String* in the method signature, *fluent*TQL must change the method signature to include *java.lang.String*. This suggested solution is an example to show how the methodology of this exploratory research works.

4.1.2 Evaluation

Answering the below research questions helps to evaluate the primary goal of the thesis.

RQ1. Which Kotlin constructs can be analyzed using SecuCheck without an extension to *fluent*TQL of SecuCheck?

Manually inspect the Jimple code and check whether existing *fluent*TQL can specify the Kotlin construct without any additional workaround by the user. If the Kotlin construct can be specified and analyzed by SecuCheck without any workaround by the user, then the Kotlin construct is categorized into this research question.

RQ2. Which Kotlin constructs can be analyzed using SecuCheck without an extension to the SecuCheck-core taint analysis?

If the SecuCheck-core taint analysis implementation does not need an extension to support the analysis of a Kotlin construct, then the construct is categorized into this research question.

RQ3. Which Kotlin constructs can be analyzed using SecuCheck without an extension to the IR generator of SecuCheck?

If the Soot framework successfully generates the valid Jimple code, then the construct is categorized into this research question.

4.2 Secondary goal

4.2.1 Methodology

The secondary goal is the optional goal of the thesis. If time permits, after the manual examination of the Jimple code for a Kotlin construct that needs modification in any of the components, then we will conceptually propose the solution. If the proposed solution is feasible to implement within the given time limit of this thesis work, then we implement the solution. If the construct needs modification in the IR generator component, we do not perform the secondary goal since the IR generator belongs to the Soot framework project.

4.2.2 Evaluation

Gather the real-world vulnerable Kotlin projects and run SecuCheck with extension to evaluate how much SecuCheck can analyze Kotlin after the extension.

5 Preliminary Structuring

1. **Introduction**
    - Domain: Starts with Kotlin and its popularity in recent years over Java.
    - Problem Description: Show with example the security vulnerabilities in Kotlin. Then explain why theoretically we can use static analysis tool intended for Java to analyze Kotlin. Then, explain that there will be complication in using the existing static analysis tool. Therefore we need this research.
    - Thesis structure: Give a brief structure of the complete thesis.
2. **Background**
    - **Static code analysis**: Briefly explain about static code analysis with high level architecture to show that tool uses Jimple code for analysis.
    - **Taint analysis**: Explain in brief what is taint analysis, source, sink, sanitizer etc.
    - **Soot framework**: Answer-What is Soot framework?
    - **Boomerang**: Answer-What is Boomerang?
    - **SecuCheck**: Answer-What is SecuCheck tool and what it does. Also explain its each components like *fluentTQL*, SecuCheck-core analysis etc.
3. **Exploratory research**
    - Briefly explain the methodology of the research i.e. manual examination of the Jimple code. Then, write the results of each inspected constructs.
4. **Evaluation**
    - For primary goal, Explain in brief how did we categorize a Kotlin construct into one of the research questions. Finally, Answer the research questions.
    - For optional goal, gather real world Kotlin projects with vulnerabilities and run the extended SecuCheck to answer how many taintflows found.
5. **Conclusion and Future Work**
    - **Summary**: Provide the summary of the thesis work and the end result.
    - **Future Work**: Provide the future work of this research that guides the researcher to perform more research on dataflow analysis for Kotlin.

6 Time Plan

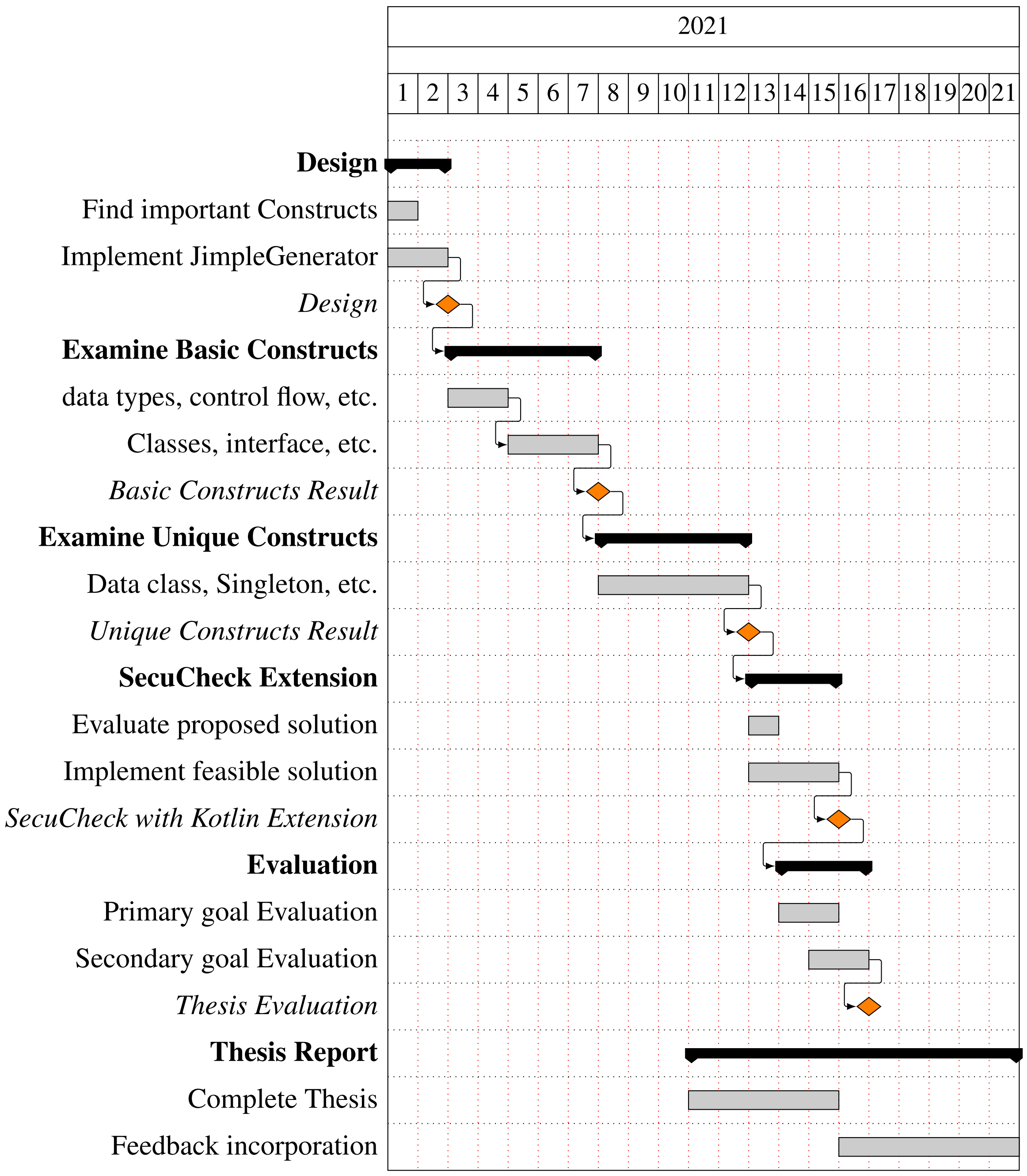

Figure 6-1: "Proposed weekly thesis schedule."

7 Bibliography

[1] CWE-117: Improper Output Neutralization for Logs. `https://cwe.mitre.org/data/definitions/117.html`. Accessed: 2021-June-22.

[2] CWE-476: NULL Pointer Dereference. `https://cwe.mitre.org/data/definitions/476.html`. Accessed: 2021-June-18.

[3] IEEE SPECTRUM: The Top Programming Language 2020. `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020`. Accessed: 2021-June-18.

[4] SecuCheck-SC-1.1.0. `https://github.com/secure-software-engineering/secucheck/tree/SC-1.1.0`. Accessed: 2021-June-18.

[5] D. Endler. The evolution of cross site scripting attacks. Technical report, Technical report, iDEFENSE Labs, 2002.

[6] W. G. Halfond, J. Viegas, A. Orso, et al. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE international symposium on secure software engineering*, volume 1, pages 13–15. IEEE, 2006.

[7] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 2019.

[8] S. Kummita, G. Piskachev, J. Späth, and E. Bodden. Qualitative and Quantitative Analysis of Callgraph Algorithms for Python. In *2021 International Conference on Code Quality (ICCQ)*, pages 1–15. IEEE, 2021.

[9] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[10] J. Späth, L. Nguyen Quang Do, K. Ali, and E. 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.

[11] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1):157–171, 1986.