To what extent can we analyze Kotlin programs using existing Java taint analysis tools?

Ranjith Krishnamurthy  
Fraunhofer IEM  
ranjith.krishnamurthy@iem.fraunhofer.de

Goran Piskachev  
Fraunhofer IEM  
goran.piskachev@iem.fraunhofer.de

Eric Bodden  
Paderborn University & Fraunhofer IEM  
eric.bodden@uni-paderborn.de

Abstract—As an alternative to Java, Kotlin has gained rapid popularity since its introduction and has become the default choice for developing Android apps. However, due to its interoperability with Java, Kotlin programs may contain almost the same security vulnerabilities as their Java counterparts. Hence, we question: to what extent can we use an existing Java static taint analysis on Kotlin code? In this paper, we investigate the challenges in implementing a taint analysis for Kotlin compared to Java. To answer this question, we performed an exploratory study where each Kotlin construct was examined and compared to its Java equivalent. We identified 18 engineering challenges that static-analysis writers need to handle differently due to Kotlin’s unique constructs or the differences in the generated bytecode between the Kotlin and Java compilers. For eight of them, we provide a conceptual solution, while six of those we implemented as part of SECUCHECK-KOTLIN, an extension to the existing Java taint analysis SECU CHECK.

Index Terms—static analysis, security, kotlin, taint analysis

I. INTRODUCTION

Ten years since its introduction, Kotlin has been one of the fastest-growing programming languages (PLs). As of June 2022, it is the twelfth most popular PL by the PYPL index1. Additionally, over 60% of the Android apps are written in Kotlin, earning it the title of the default PL for the Android framework2. One of the Kotlin advantages as a JVM-based PL is its interoperability with Java and its unique constructs like data classes, coroutines, null safety, extensions, etc.

Like Java, Kotlin code may be vulnerable to security vulnerabilities, such as SQL injection [1]. Therefore, statically analyzing Kotlin code can be a helpful method for detecting bugs and security vulnerabilities as early as possible. Despite its popularity, very few static-analysis tools can analyze Kotlin code, such as KtLint [2], Detekt [3], Diktat [4], and SonarQube [5]. These tools only perform pattern-based analyses using simple rules, such as the rules of SonarQube [6]. We are not aware of any tool that performs deep data-flow analyses on Kotlin code. For example, taint analysis has proved to be very useful for detecting many prevalent security vulnerabilities [7] such as injections [1], [8], [9] and XSS [10]. This versatility of the taint analysis is due to its capacity to set various inputs in the form of rules. At its core, the analysis follows the path between so-called sources, where the taint is created, until so-called sinks, where the taint is reported. The information for the sources and sinks is often encoded in a rule via a domain-specific language (DSL).

For Java, there are many existing taint analyses [11], [12] that can be used to detect many taint-style security vulnerabilities. Since Kotlin compiles to the Java bytecode, theoretically, one can use existing Java taint analyses on Kotlin code. However, the Kotlin compiler generates the bytecode differently than that of Java. This leads to the question: can one use taint analysis tools intended for Java to analyze Kotlin programs, or must one reinvent the wheel?

In this paper, we report the result of an exploratory study that we conducted to address this question. We analyzed the Kotlin-generated bytecode for each language construct and compared it to the Java equivalent. We used the Jimple intermediate representation generated by the Soot framework [13] for this comparison. For completeness, we used the official Kotlin documentation [14] and created a micro benchmark with 294 simple Kotlin programs and 135 simple Java programs, where each program demonstrates a single language construct. When considering taint analysis, we found that most Kotlin constructs can be analyzed the same way as the Java equivalents. However, we also found 18 engineering challenges that require a different approach. For example, functions declared as top-level elements do not have a parent class in the source code. However, the compiler generates a parent class in the Java bytecode, which the taint analysis should be aware of to locate the function correctly. We propose solutions for eight of these challenges that analysis writers can implement. As a proof of concept, we extended an existing Java taint analysis tool, SECUCHECK [12], by implementing six of our eight solutions, creating a taint analysis tool SECUCHECK-KOTLIN that supports the standard language constructs. Finally, we evaluated the applicability of SECUCHECK-KOTLIN with the Kotlin version of the PetClinic application3.

We present the details of our methodology in Section II. Then, in Section III, we report on our findings from the study. Next, we present details of our implementation of SECUCHECK-KOTLIN in Section IV. Finally, we conclude and present our future work in Section V. An extended Technical Report [15] gives more detailed information on the

1https://pypl.github.io/PYPL.html  
2http://surl.li/cfrcc

3https://github.com/spring-petclinic/spring-petclinic-kotlin
Table I: List of Kotlin’s features discussed in Kotlin’s official documentation.

| Constructs                                      | #Sub-constructs | Supported |
|-------------------------------------------------|-----------------|-----------|
| Types, Control flow, Packages & imports, Null safety, Equality, This expression, Destructuring declarations, Ranges and Progressions | 11              | ✓         |
| Classes and objects (except for Delegated properties) | 17              | ✓         |
| Functions (except for Blinders)                 | 5               | ✓         |
| Asynchronous programming techniques, Coroutines, Annotations, and Reflections | 4               | x         |
| Collections and Iterators                       | 2               | ✓         |

18 challenges and the list of Kotlin constructs affected by those challenges.

II. METHODOLOGY

We examined the intermediate representation (IR) of the Kotlin code and—if existing—the equivalent Java code. Our methodology consists of automatic IR generation with metadata useful for our examination, which is a manual step that follows. We examined the following: (1) whether the generated IR for Kotlin is valid and can be analyzed the same way as the IR from equivalent Java code, (2) whether there are difficulties due to the definition of sources and sinks, and (3) whether there are language constructs in Kotlin that the analysis needs to handle in a new unique way when compared to Java. We did not consider challenges that can occur due to the call-graph-generation algorithms or computing alias information algorithms.

We used Kotlin’s official documentation [14] to examine each language construct. During the examination, we covered all constructs from the “Concepts” section and a few from the “Standard library” section (Collections, Iterators, Ranges, and Progressions). We did not consider constructs that were in the experimental stage at the time of this study. Table I summarizes Kotlin’s constructs discussed in the official documentation and the those we manually examined.

Kotlin targets most Java Development Kit (JDK) versions. However, the annual developer ecosystem survey conducted by JetBrains in 2020 shows that 73% of Kotlin developers target JDK 8 [16]. Furthermore, Kotlin targets JDK 8 by default. Therefore, we consider JDK 8 for this exploratory study. Additionally, we consider the Kotlin version 1.5.10.

The Kotlin compiler has various options and annotations for modifying the compilation process, which alters the output of the compiler, Java bytecode. For this study, we used the default configuration of the compiler.

For the IR generation, we built a tool that generates Jimple IR using the Soot framework [13]—JimpleProvider. The Jimple code is organized based on the package name. Furthermore, for each class, JimpleProvider generates metadata in a JSON file that contains information such as class name, super class, implemented interfaces, method count, method signatures, local variables, invoke expressions, etc. This metadata helps to identify the challenges easily and quickly. For deeper examination, we then examine the IR and Java bytecode.

A. Micro benchmark

Using real-world projects for the manual examination is infeasible because a real-world project has a complex mix of many constructs, making it hard to identify them clearly in Jimple. Therefore, we built a micro benchmark suite classified into two groups—Kotlin suite and Java suite. The Kotlin suite consists of small Kotlin programs, each focusing on one particular Kotlin construct. If a corresponding feature exists in Java, then an equivalent program is present in the Java suite. The suits contain six main categories: basics (43 Kotlin & 36 Java files), classes and objects (118 Kotlin & 80 Java files), functions (27 Kotlin & 4 Java files), generics (8 Kotlin & 10 Java files), unique to Kotlin (87 Kotlin files), and collection (11 Kotlin & 5 Java files).

B. Manual examination

The manual examination of the Jimple code was performed by the first author, who has more than 4.5 years of software development experience and is a Ph.D. student focusing on programming languages and static analysis. The more complex constructs, especially those specific to Kotlin or with differences from Java, were discussed with the second author, a Ph.D. student in the last year with expertise in the static analysis, and an external researcher with professional experience in Kotlin development. The examiners used the JimpleProvider to generate the IR for the entire micro benchmark. Then, each construct was inspected manually. First, the generated metadata that provides information related to taint analysis is studied. Next, the generated IR is checked for a deeper examination. If more information is needed, then the generated bytecode is examined. Based on this, the examiner concluded whether a construct requires special handling in Kotlin taint analysis compared to Java taint analysis.

C. Threats to Validity

Our study involves a manual step, making it possible that some of the findings are incomplete or incorrect. Furthermore, the programs written in the micro benchmark suite are based on personal experience. Therefore, some advanced use cases may be missing. As discussed earlier in this section, we considered the Kotlin version 1.5.10 and the target JDK 8. However, there is a risk that for some of the constructs, the Kotlin compiler may generate the bytecode differently for different versions. Also, for some constructs, the compiler may generate bytecode differently if some compiler options are used. As stated earlier, we only used the default configuration.

III. FINDINGS

First, we present the engineering challenges we identified and to which we have proposed a solution. Second, we present the engineering challenges, which we leave as open issues.

A. Engineering challenges with proposed solution

1) Data type mapping: On the bytecode level, the basic data types in Kotlin are mapped to Java data types. For example, the non-nullable kotlin.Int is mapped to Java’s
The complete mapping of the data types is described in the official documentation\footnote{https://kotlinlang.org/docs/java-interop.html#mapped-types}. Due to this mapping, the users must provide valid method signatures based on the Java bytecode to specify the source, sink, and other relevant method calls. However, it is cumbersome for the users to find the valid method signatures in big projects, making the tool not usable. 

**Proposed solution:** To handle this challenge, static-analysis developers can implement a data type transformer, which takes a method signature provided by the users as input. Then, the transformer checks for the parameters and return type in the given method signature. If the parameters type and return type are valid Kotlin data types, the transformer replaces the Kotlin data type with the respective Java data type.

2) **Type alias:** A type alias allows developers to give a new name to the existing type. For example, in the Kotlin standard library, `ArrayList` is defined as a type alias to `java.util.ArrayList`. Therefore, `ArrayList` does not exist in the bytecode. However, the experts in the Kotlin programming language know which types are defined as type alias in Kotlin standard libraries. Furthermore, domain experts in custom libraries such as cryptographic APIs know what type aliases are defined in their libraries. On the other hand, users of the existing Java taint analysis tools may not know such type aliases and may give invalid method signatures.

**Proposed solution:** Static-analysis developers can implement a feature as part of the DSL that allows domain experts to specify type aliases—type alias specifications. The DSL semantics replaces all the type aliases found in the given method signatures with the original type specified in the given type alias specifications.

3) **Property:** In Kotlin, a property is a field with an accessor. By default, Kotlin provides a getter and setter for mutable properties; for immutable properties, the getter only. Whenever there is access to a property in Kotlin source code, the Kotlin compiler uses the respective accessor method in the Java bytecode. Similar to variables, properties can be tainted. Therefore, the getter and setter of properties can be the source, sink, or propagator methods. Thus, the user needs to be aware of these signatures.

**Proposed solution:** Static-analysis developers can implement a feature in the DSL that enables users to specify a property by providing the fully qualified class name in which the property is defined, the property name, and the property’s type. Then, the valid accessor method signature can be built automatically. The pattern for the getter method is `<given fully qualified class name> <given property’s type> get<given property name with first letter caps>()`. Similarly, the setter method’s pattern is `<given fully qualified class name> void set<given property name with first letter caps>(<given property’s type>)`.

4) **Top-level members:** In Kotlin, top-level members are defined in a Kotlin file under a package. Kotlin functions and properties can be top-level members. These members are not declared in any class, object, or interface. Therefore, in Kotlin source code, top-level members can be accessed directly without creating any object or using a class to access it. However, the Kotlin compiler generates a class in the Java bytecode and declares those top-level members as static members in the generated class. Suppose a novice user wants to specify top-level members as the source, sanitizer, propagator, or sink methods. In that case, the user must identify the valid class name in the method signature of top-level members.

**Proposed solution:** Static-analysis developers can provide a feature in the DSL that enables users to specify a function or a property as a top-level member by providing the package name and the file name in which a top-level member is defined. Then, the DSL component can build a valid class name for a top-level function or accessor of a top-level property. The rule to build the valid class name is `<given package name>.<given file name>Rt`.

5) **Default arguments:** In Kotlin, a constructor or function can have default arguments. The Kotlin compiler generates two implementations in the Java bytecode: (1) the actual implementation with all the parameters defined in the source code, and (2) the implementation with additional arguments that determines the default arguments’ value, which calls the actual implementation. For a constructor, the compiler adds two additional arguments at the end—`int` and `kotlin.jvm.internal.DefaultConstructorMarker`. Similarly, for a function, the compiler adds `int` and `java.lang.Object` at the end. Additionally, if the function is a member function, the compiler adds the first argument of the type in which the function is defined. The compiler adds the suffix `$default` to the function name of the second implementation of a default argument top-level or member function. The compiler calls the second implementation if a developer does not pass value to default arguments. Suppose users want to specify a default argument constructor or function as a source or sink method. In that case, the analysis component should identify the second implementation generated by the compiler and track the variables correctly.

**Proposed solution:** If the analysis fails to identify a method call as a source, sink, or other relevant method specified in the taint-flow specifications, then the analysis checks for the default argument feature. For each function or constructor in taint-flow specifications, add the additional arguments and modify the function name as described in Sub-Section III-A5. Subsequently, if the method signature matches the method call’s signature, track the respective variables. For constructor and top-level function, track the variables based on the specified rules of the matched method in specifications. However, for member functions, if the `this-object` is specified to track, then track the first argument in the Java bytecode. Likewise, track the second argument in the Java bytecode if the first argument is specified to track and so forth.

6) **Extensions:** In Kotlin, the extension construct allows extending an existing class with new members without using inheritance. However, extensions will not modify and add a new member to an existing class; instead, the new member is
made accessible using the dot-notation. In the Java bytecode for a top-level extension function, the Kotlin compiler adds the receiver type as the first argument, followed by the actual parameters defined in the source code. Similarly, the compiler adds the receiver type as the first argument to the getter method of a top-level extension property. Note: The compiler generates only the getter method for an extension property. Furthermore, for top-level companion object extension, the compiler also adds the receiver type as the first argument, followed by the actual argument defined in the source code. However, the added first argument type is the wrapper class generated for a companion object. The companion object is discussed in detail in Section III-B1. Like top-level extension members, the compiler also adds the receiver type as the first argument for an extension defined as a class member. Furthermore, Kotlin supports qualified this-object to access the outer class's this-object. For this, the compiler considers the actual this-object (outer class's this) in the Java bytecode as a qualified outer class's this-object in the source code and the first argument in the Java bytecode as a receiver this-object in the source code. Suppose users want to specify an extension member as a source or sink method, then users might give an invalid method signature since users might not be aware of the added first argument of receiver type. Furthermore, if users specify to track the this-object in an extension member, then the analysis should track the first argument. Similarly, if users specify to track the first argument in an extension function, then the analysis should track the second argument and so forth.

**Proposed solution:** To handle extension functions and extension properties, static-analysis developers should make their taint-flow specifications aware of these. If this is done through the DSL for taint-flow specifications, the DSL can build the valid method signature by adding the given fully qualified class name as the first argument. Furthermore, the users should not be able to obtain a setter method from an extension property since an extension property can not have a setter method. However, developers should make their taint-flow specifiers aware of these. If this is done through the DSL for taint-flow specifications, the DSL can build the valid method signature by adding the given fully qualified class name as the first argument. Furthermore, the users should not be able to obtain a setter method from an extension property since an extension property can not have a setter method.

To handle companion object extensions, static-analysis developers can provide a feature in the DSL. This feature enables the users to specify a function or property as a companion object extension member by providing the fully qualified class name and the name of the companion object for which the extension is defined. If the name of the companion object is not given, then by default, the name is Companion. From these inputs, the generated wrapper class for the companion object can be built as <given fully qualified class name>$<given companion object name>. Then, the valid method signature can be built by adding this wrapper class as a first argument.

To handle the qualified this-object in extensions as members, the DSL should be able to track the this-object as extension receiver or dispatch receiver (outer class’s this-object). If users specify to track this-object as an extension receiver, modify the taint-flow specification to track the first parameter in the Java bytecode. Similarly, if users specify to track this-object as dispatch receiver, modify the taint-flow specification to track the actual this-object in the Java bytecode. Similarly, for an extension function, if user specify to track the first parameter, then analysis should track the second parameter and so forth.

7) **Infix function:** In Kotlin, infix functions are called using the infix notation, i.e., without the dot notation and the parentheses. The infix function must be a member function or extension function and must have a single parameter without a default value. Similar to a standard function, an infix function can be a source, sink, and other relevant methods. However, a novice user of taint analysis tools may not know how the infix function works in the Java bytecode and may provide invalid method signatures.

**Proposed solution:** Static-analysis developers can provide a DSL feature that enables users of taint analysis tools to specify a function as an infix function by providing a function name, receiver type, parameter type, and return type. Then, DSL can build a valid method signature as <given receiver type>: <given return type> <given function name>(<given parameter type>).

8) **Operator overloading:** Operator overloading redefines the implementation of the built-in operators with specific types. For example, one can overload the ++ operator by defining the function inc on a custom class. The compiler calls the implemented inc function in the Java bytecode. The official documentation\(^5\) describes the mapping between the built-in operator and the function name. An overloaded operator function can be a sanitizer or propagator method. However, users of taint analysis tools may not know the mapping of the built-in operators to the function name and may provide invalid method signatures.

**Proposed solution:** Static-analysis developers can provide a feature in DSL that enables users to specify an overloaded operator by providing the symbol of an operator, type of the receiver, return type, and the parameter(s) type based on an operator. Then, DSL can build the valid method signature by mapping the given operator symbol to the function as described in the official documentation\(^6\).

B. **Engineering challenges without solution (open issues)**

1) **Companion object:** In Kotlin, a companion object binds members to a class rather than the instance of a class. Kotlin’s companion object is similar to Java’s static members. However, the Kotlin compiler generates a wrapper class for each companion object in the Java bytecode. The compiler places the implementation of that companion object’s members in that wrapper class. Furthermore, to allow that wrapper class to access the private members of the actual class and vice versa, the compiler generates additional functions for each private member. Due to this implementation of companion objects in the Java bytecode, users of taint analysis tools might find it difficult to identify valid method signatures. Additionally, for the function that takes a companion object as a parameter, users must give that parameter type as a generated wrapper

\(^5\)https://kotlinlang.org/docs/operator-overloading.html
class in the method signature, which is not visible in the source code. Furthermore, the analysis should be aware of additional functions for private members, which might be a possible source, sink, or propagator.

2) **Destructuring declaration:** In Kotlin, an object can be destructured into multiple variables in a single statement using the destructuring declaration. To allow a class to destruct, that class must have the `componentN` functions. These component functions return the properties of a class. The widely used convention for the order of `componentN` functions is the order of properties defined in a class. However, it is not mandatory, and developers can make component functions return any properties of a class. Suppose users of taint analysis tools specify the getter method of the first functions return any properties of a class. Suppose users of taint analysis tools specify the getter method of the first property as a source method. Then the analysis component should be able to identify the `component1` function as a source method. Therefore, the analysis must know the mapping between `componentN` functions and properties of a class to identify the taint-flow in a destructuring declaration.

3) **Internal modifier:** In Kotlin, a member declared with an `internal` modifier is only visible inside the module in which the member is defined. Kotlin defines a module as a group of Kotlin files that are compiled together. In the Java bytecode, the Kotlin compiler appends the symbol hyphen followed by the module name for the accessor of an internal property and to an internal member function. However, we did not observe this behavior for classes, interfaces, top-level functions, or accessors of top-level properties, which are declared as `internal`. Suppose users of taint analysis tools specify an internal member function or accessors of internal property as a sink method. In that case, the analysis component should identify the modified name with the appended module name as a sink method. Otherwise, the analysis component fails to detect taint-flow in `internal` member functions and properties. Note: if there is a symbol hyphen in the module name, the Kotlin compiler replaces it with the underscore before appending it to the internal member functions and accessors of internal property in the Java bytecode.

4) **Inline class:** Kotlin’s inline class wraps an existing class with improved performance compared to a manually created wrapper class. In the Java bytecode, the Kotlin compiler generates some of the member functions for an inline class—constructor, accessor for a property (wrapped class), `toString`, `hashCode`, and equality check. These functions are generated to support the interoperability with Java. However, the compiler generates the alternative version of these functions to improve the performance by inlineing the wrapped class in place of wrapper class usage. In addition, the compiler adds the suffix `-impl` to the improved version of these functions and to the overridden function of an interface. Additionally, the compiler generates `box-impl` and `unbox-impl` function for boxing and unboxing the wrapped class. The Kotlin compiler calls the `-impl` version of member functions wherever it is possible to improve the performance.

With the actual implementation, the analysis should identify its `-impl` version as a source. Otherwise, the existing Java taint analysis tools fail to detect taint-flows in an inline class.

5) **Function returning anonymous object:** In Kotlin, object expressions create objects of an anonymous class. Every object expression has at least one base class. The Kotlin compiler generates a wrapper class for each instance of object expression in the Java bytecode similar to Java. However, in contrast to Java, the return type in Kotlin’s function is not mandatory to specify, and the compiler can infer the type. Suppose a function is private and returns an anonymous object. In that case, the compiler infers the return type as the generated wrapper class, which is not visible in the source code. This is a challenge to identify a valid method signature.

6) **Local functions:** Kotlin supports local functions, which are functions inside other functions. The Kotlin compiler generates a static function in the Java bytecode for a local function. The naming of this function depends on the outer and local function name. Furthermore, the compiler appends digits to make the function unique if there are multiple local functions with the same name. Additionally, local functions can access outer function’s local variables. These accessed variables are passed as arguments to the generated static function. Furthermore, if the accessed variable is mutable, the compiler passes the reference type to reflect the change in the outer scope. Suppose the users want to specify local functions as a source method. In that case, it is challenging to identify the valid method signature of a local function. Furthermore, the analysis must handle the accessed local variables of the outer functions to track the tainted variable.

7) **Higher-order functions:** Kotlin provides a function type that enables the higher-order function. These function types are mapped to `kotlin.jvm.functions.Function*` types in the Java bytecode. Furthermore, there are five ways to create an instance of a function type—lambda expression, anonymous function, function literal with a receiver, callable reference, and instances of a custom class that implements a function type. The Kotlin compiler generates a wrapper class for each instance of a function type in a Kotlin source code.

Java uses `invokedynamic` instruction for lambda expression. Therefore, the existing Java taint analysis tools detect taint-flows in lambda expressions in Java by handling the `invokedynamic` instruction in the Java bytecode. However, by default, the Kotlin compiler does not use `invokedynamic` instruction for an instance of a function type, which leads to the existing Java taint analysis tools failing to detect taint-flows in higher-order functions. Therefore, the analysis must handle the generated wrapper class for an instance of a function type to track the tainted information. Furthermore, the analysis should handle the `receiver` property to track the tainted receiver object. Similar to local functions (III-B6), the analysis should handle the accessed local variables of the outer functions to track the tainted variable.

8) **Inline function:** The implementation of a higher-order (III-B7) function leads to runtime overhead. However, in some scenarios, such runtime overhead can be eliminated.
by inlining the lambda expression rather than creating an instance of a function type. For this purpose, Kotlin provides inline functions. For example, the `println` function in Kotlin is declared as inline, which calls the Java’s function `System.out.println`. Therefore, in the Java bytecode, we find the `System.out.println` function call in place of Kotlin’s `println` call site. Similarly, custom higher-order functions can also be declared as inline in Kotlin. Suppose users of taint analysis tools specify an inline function as a sink method. In that case, taint analysis tools fail to detect taint-flow that reaches this sink method since there is no actual method call of an inline function in the Java bytecode. Therefore, taint analysis tools must know the propagation rule for all the method calls in the body of that inline function. Otherwise, it fails to detect taint-flows in inline functions.

9) **Sealed class**: A sealed class restricts users from inheriting a class or interface, and all the derived classes are known at compile time. To achieve this, the Kotlin compiler makes the constructor private and overloads the constructor with an additional parameter at the end—`kotlin.jvm.internal.DefaultConstructorMarker`. This allows the compiler to call the overloaded constructor for the known derived class and restricts developers from creating a new derived class. Suppose users of taint analysis tools specify the constructor of a sealed class as a propagator method. In that case, the analysis must identify the overloaded constructor as a propagator. Otherwise, taint analysis tools fail to detect taint-flows in a sealed class’s constructor.

10) **Package**: In contrast to Java, the package name in Kotlin can be different than the path of that Kotlin file. Once the analysis component completes and returns the found results, some existing Java taint analysis tools use the package name to build the path of the Java file to display the errors in an IDE. However, if the Kotlin file’s path is different from its package, then taint analysis tools fail to display the found taint-flows in an IDE.

IV. **SCUCHECK-KOTLIN**

Based on our findings from the previous section, we extended an existing Java taint analysis tool called SCUCHECK [12] by implementing six of the eight solutions proposed in Sub-Section III-A: data type mapping (III-A1), type alias (III-A2), property (III-A3), top-level members (III-A4), extensions (III-A6), and default argument (III-A5). We handled these engineering challenges without modifying the existing architecture of SCUCHECK. Our implementation is available as an open-source Github project.

To evaluate the applicability of SCUCHECK-KOTLIN, we found a vulnerable version of the Spring PetClinic application written in Kotlin. The project contains 27 Kotlin files with six known hibernate injections. SCUCHECK-KOTLIN found all the six taint-flows with a run time of 11.05 seconds (average of 10 runs). Out of six implemented solutions, the solution for data type mapping and property enabled an effective taint analysis on the vulnerable Spring PetClinic application written in Kotlin. In addition, SECUCHECK-KOTLIN successfully displayed the valid line numbers of the source and sink methods and the customized error messages.

V. **CONCLUSION AND FUTURE WORK**

In this paper, we presented our exploratory study for Kotlin taint analysis, which shows that most of the Kotlin constructs can be analyzed by an existing Java taint analysis tool. However, we found 18 engineering challenges that must be handled differently than the Java taint analysis. For eight of these challenges, we proposed solutions. Finally, as a proof of concept, we extended an existing Java taint analysis, SECUCHECK, by implementing six of these solutions, which led to SECUCHECK-KOTLIN. We evaluated the applicability of SECUCHECK-KOTLIN, which found all the six expected taint-flows. In the future, we plan to work on the open issues from Sub-Section III-B and extend the implementation of SECUCHECK-KOTLIN, after which a thorough evaluation with real-world applications can be performed.

REFERENCES

[1] “CWE-89: Improper Neutralization of Special Elements used in an SQL Command,” https://cwe.mitre.org/data/definitions/89.html, accessed: 2021-June-22.
[2] “KTLINT: An anti-bikeshedding Kotlin linter with built-in formatter,” https://github.com/pinterest/ktlint, accessed: 2021-December-14.
[3] “DETEKT: static analysis for Kotlin,” https://github.com/detekt/detekt, accessed: 2021-December-14.
[4] “DIKTAT: Kotlin linter,” https://github.com/diktat/diktat-static-analysis/diKTat, accessed: 2021-December-14.
[5] “SONARQUBE: automatic code review tool to detect bugs, vulnerabilities, and code smells,” https://docs.sonarqube.org/latest/, accessed: 2021-December-14.
[6] “SONARQUBE: rules for Kotlin,” https://rules.sonarsource.com/kotlin, accessed: 2021-December-14.
[7] G. Piskachev, J. Späth, I. Budde, and E. Bodden, “Fluently specifying taint-flow queries with fluenttql,” Empir. Softw. Eng., vol. 27, no. 5, p. 104, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10165-y
[8] “CWE-77: Improper Neutralization of Special Elements used in a Command (‘Command Injection’)”, https://cwe.mitre.org/data/definitions/77.html, accessed: 2021-October-25.
[9] “CWE-476: NULL Pointer Dereference,” https://cwe.mitre.org/data/definitions/476.html, accessed: 2021-June-18.
[10] D. Endler, “The evolution of cross site scripting attacks,” Technical report, iDEFENSE Labs, Tech. Rep., 2002.
[11] S. Arzt, S. Rathofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” Acm Sigplan Notices, vol. 49, no. 6, pp. 259–269, 2014.
[12] G. Piskachev, R. Krishnamurthy, and E. Bodden, “Secucheck: Engineering configurable taint analysis for software developers,” in 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), 2021, pp. 24–29.
[13] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, “The Soot framework for Java program analysis: a retrospective,” in Cetus Users and Compiler Infrastructure Workshop (CETUS 2011), vol. 15, no. 35, 2011.
[14] “Kotlin’s official documentation,” https://kotlinlang.org/docs/home.html, accessed: 2021-November-21.
[15] R. Krishnamurthy, G. Piskachev, and E. Bodden, “To what extent can we analyze kotlin programs using existing java taint analysis tools? (extended version),” arXiv, 2022. [Online]. Available: https://arxiv.org/abs/2201.09379
[16] “The State of Developer Ecosystem 2020,” https://www.jetrains.com/lp/devcosystem-2020/kotlin, accessed: 2021-October-27.