Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sparse Value Flow Analysis for Java Bytecode: An Empirical Assessment

José C. Tafur

Dissertação apresentada como requisito parcial para qualificação do
Mestrado Profissional em Computação Aplicada

Orientador
Prof. Dr. Rodrigo Bonifacio

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sparse Value Flow Analysis for Java Bytecode: An Empirical Assessment

José C. Tafur

Dissertação apresentada como requisito parcial para qualificação do Mestrado Profissional em Computação Aplicada

Brasília, 3 de decembro de 2025

Abstract

Sensitive data leakage is a significant security concern in modern application interactions. Approaches like static taint analysis effectively identify potential leaks by tracking and analyzing the flow of variable values throughout a program. Sparse Value Flow Analysis (SVFA) is a technique within static analysis that has demonstrated effectiveness in identifying data leaks in C and C++ programs. However, its potential application for analyzing Java bytecode programs has not yet been investigated. A collaboration between the University of Brasilia and the University of Paderborn led to the development of an analyzer for Java bytecode, named JSVFA. This tool has been modified to detect semantic merge conflicts and incorporated into various research initiatives. The current test pass rate is 51.64% (63 out of 122) when assessed against the SecuriBench benchmark, which is one of the test suites included in the tainted analysis tool referred to as FlowDroid.

This thesis aims to evaluate how well JSVFA framework works compared to top tools for Java taint analysis, using metrics such as precision, recall, f-score, and pass rate. Furthermore, it seeks to enhance the existing JSVFA framework source code by utilizing the failing tests for problem characterization. To validate the enhancements, we will compare the new metrics values from the upgraded version of JSVFA framework using the SecuriBench benchmark alongside its previous version and against two leading analysis tools, FlowDroid and Joana. Furthermore, we will incorporate TaintBench, a benchmark featuring "Android Application," and subsequently assess its completeness.

We have implemented enhancements in the source code leading to a new version of JSVFA framework that improves on its predecessor. The new version, tested using the SecuriBench benchmark, improves on recall, f-score, and pass rate but somewhat reduces precision. Compared to FlowDroid and Joana, the new JSVFA framework shows enhancements in recall, f-score and pass-rate, and a decline in precision, which places it in a median position. In addition, two experiments were undertaken to assess the completeness of JSVFA framework in the Android environment; it has identified 76 and 655 new leakages, respectively, which were not in TaintBench ground truth reports but later confirmed as true positives. However, JSVFA framework fails to identify the source-sink paths reported as real leaks in the same ground truth. In conclusion, JSVFA framework appears to be competitive in academia for analyzing Java code; however, for detecting data leaks in APKs, more Android lifecycle exploration is needed even though these new findings seem promising.

Keywords: Static Analysis, Sparse Value Flow Analysis, Data Leakage, Taint Analysis, Java Byte Code

Análise de fluxo de valores para bytecode Java: uma avaliação empírica.

Resumo

O vazamento de dados confidenciais representa uma preocupação relevante em relação à segurança de aplicativos atuais. Abordagens como a análise estática denominada "taint" identificam de forma eficaz potenciais vazamentos, rastreando e analisando o fluxo dos valores das variáveis ao longo do programa. A análise de fluxo denominada "Sparse Value Flow Analysis (SVFA)" é uma técnica de análise estática que se mostrou eficaz na identificação de vazamentos de dados em programas em C e C++. Entretanto, sua aplicação potencial na análise de programas Java (bytecode) ainda não foi avaliada. Uma parceria entre a Universidade de Brasília e a Universidade de Paderborn resultou na implementação do JSVFA, um analisador de bytecode Java. Essa ferramenta foi alterada para identificar conflitos semânticos e é aplicada em diversos projetos de pesquisa. A taxa de "sucesso" atual nos testes é de 51.64% (63 de 122), em comparação com o benchmark SecuriBench, que é um dos padrões utilizados para avaliar o desempenho da ferramenta de análise estática denominada FlowDroid.

Esta tese busca avaliar o desempenho do JSVFA framework em comparação com as principais ferramentas de análise estática Java, empregando medidas como precisão, recall, f-score e taxa de sucesso nos testes. Além disso, busca-se aprimorar o código-fonte existente do JSVFA framework, utilizando os testes com falha para a caracterização do problema. Para validar as melhorias, compararemos as novas medidas da versão mais recente do JSVFA framework utilizando o benchmark SecuriBench com duas ferramentas de análise de destaque, FlowDroid e Joana. Adicionalmente, incorporaremos o TaintBench, um benchmark com "Aplicativos Android," e, em seguida, avaliaremos seu desempenho.

Realizamos aprimoramentos no código-fonte, resultando em uma nova versão do JSVFA framework, que apresenta melhorias substanciais em comparação com a versão anterior. A nova versão, avaliada por meio do benchmark SecuriBench, demonstra melhorias em recall, f-score e taxa de sucesso dos testes, embora resulte em uma leve diminuição da precisão. Em comparação ao FlowDroid e ao Joana, o novo JSVFA framework demonstra um equilíbrio entre precisão, recall e f-score, posicionando-se de forma intermediária. Adicionalmente, foram conduzidos dois experimentos para avaliar o desempenho do JSVFA framework no ambiente Android, os quais resultaram na identificação de 76 e 645 novos vazamentos, respectivamente, que não estavam presentes nos relatórios de “ground truth” do TaintBench, mas foram posteriormente confirmados como “true positives”. Entretanto, o JSVFA framework não consegue detectar os vazamentos apontados nos relatórios de TaintBench. Em suma, o JSVFA framework demonstra um desempenho competitivo na análise de código Java na academia; entretanto, a identificação de vazamentos de dados em APKs requer uma investigação mais aprofundada do ciclo de vida do Android, embora as novas descobertas sejam promissoras.

Palavras-chave: Análise estatico, Análise de fluxo, Vazamento de dados, Análise de taint, Código de bytes Java

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Research goal 3

1.3 Research Characterization 4

1.4 Thesis Organization 4

2 Background **5**

2.1 Static Analysis 5

2.2 Data Flow Analysis 6

2.3 Pointer Analysis 10

2.4 Taint Analysis 13

2.5 SVFA 15

2.5.1 Rules 16

2.5.2 Graph Representation 18

3 Research Method **20**

3.1 Benchmark 01: SecuriBench 22

3.2 Benchmark 02: TaintBench 28

3.3 Data Collection 31

3.4 Data Analysis 32

4 Study Result **35**

4.1 Study I 35

4.2 Study II 45

4.3 Study III 53

5 Final Remarks **59**

5.1 Answer the research questions 59

5.2 Future Work 60

5.3 Contributions 61

5.4 Reproducibility and Code Availability 62
5.5 Conclusion 63

References **65**

Appendix **67**

A JSVFA-v0.3.0 metrics **68**

B JSVFA-v0.6.0 metrics **74**

C FlowDroid metrics **81**

D Joana metrics **87**

List of Figures

2.1 Control Flow Graph 7
2.2 SPARK overview [1] 13
2.3 Intraprocedural SVFA 18
2.4 Interprocedural SVFA 19

3.1 Generic workflow for data collection 23
3.2 Class hierarchy of the JSVFA integration with the SecuriBench 24
3.3 Class hierarchy for the SecuriBench integration in FlowDroid 25
3.4 Class hierarchy for the SecuriBench integration in Joana 27
3.5 Dependency Chain: TaintBench integration in JSVFA framework 30
3.6 Metrics flow 31
3.7 Dependency Chain: SecuriBench integration in JSVFA framework 33

4.1 JSVFA-v0.3.0 comparison against FlowDroid and Joana 36
4.2 Case: ArrayTest1 38
4.3 Pass Case: StringBuilder 45
4.4 Fail Case: StringBuilder 45
4.5 Test categories: JSVFA-v0.3.0 vs JSVFA-v0.6.0 47
4.6 JSVFA-v0.6.0 comparison against FlowDroid and Joana 54

List of Tables

1.1 Comparative metrics 3

2.1 Useful terms for Reaching Definition equations 8
2.2 Reaching Definition - entries 9
2.3 Reaching Definition - exits 10
2.4 Andersen's Algorithm constraints 11
2.5 Steensgaard's Algorithm constraints 12
2.6 Sparse Value-Flow rules 17

3.1 JSVFA framework metric report for Aliasing tests 32
3.2 TaintBench metric report for Experiment-I 33

4.1 Metrics: JSVFA-v0.3.0 vs JSVFA-v0.6.0 46
4.2 Common Issues 53
4.3 TaintBench- Experiment I 55
4.4 TaintBench- Experiment II 58

Chapter 1

Introduction

Several techniques have been designed to improve software quality, ranging from manual and automated testing to formal methods. Testing is often incomplete, while formal methods are considered too expensive to apply to an entire system. Another prominent approach to software quality assurance relies on static analysis, which aims to estimate program behavior without requiring formal models or execution scenarios.

Static analysis gathers valuable information that can be applied across various domains, including bug detection, security vulnerability identification, and program optimization. It typically leverages a range of algorithms, some of which serve as the foundation for more advanced techniques. For example, the Reaching Definitions algorithm is a fundamental data-flow analysis used to determine which variable definitions may reach specific points in a program. Another foundational algorithm is Andersen's points-to analysis [2], which identifies the set of objects that a pointer may reference. These foundational techniques can be used to implement more advanced analyses, such as taint analysis. This analysis tracks the flow of sensitive data through a program and is commonly used to identify security vulnerabilities. It relies on both reaching definitions and pointer analysis to determine whether confidential information might propagate to locations where it could be exposed [3].

Taint analysis is a valuable technique for malware classification and for detecting common vulnerabilities, such as sensitive data leakage, SQL injection, and cross-site scripting (XSS). Listing 1.1 illustrates a simple example of a *data leak*, in which confidential information initially stored in variable `"a"` at `line:2` propagates through intermediate variables until it reaches a leakage point at `line:5`.

Listing 1.1: Example of data leakage that might be identified using taint analysis.

```
1  main() {
2      a = getPassword();
```

```
    b = a;
    c = b.toUpperCase();
    sendEmailToThridParty(c);
}
```

There are several successful taint analysis tools for Java, including FlowDroid [4], JDVS [5], and TAJ [6], which predominantly rely on Soot [7] as their underlying framework. Soot is a widely used framework for analyzing and transforming JVM-based programs [7], which supports multiple intermediate representations—Baf, Jimple, Shimple, and Grimp—and provides a range of analyses, including call graph construction, intra- and inter-procedural data-flow analysis, and points-to analysis. Since its release in 1999, Soot has remained a prominent tool in both academia and industry. *Sparse Value Flow Analysis* (SVFA) [3] is an alternative approach for implementing taint analysis. Although SVFA has been implemented for analyzing programs written in C and C++ [3], to the best of our knowledge, there is still no full-fledged SVFA-based taint analysis implementation for Java.

1.1 Problem Statement

As part of a collaboration between the University of Brasília and Paderborn University, we began developing an SVFA framework for analyzing Java programs. It is called JSVFA framework, and its infrastructure is designed to evaluate Java programs using Sparse Value Flow Analysis [8]. This project dates back to June 2019, when its development started through a collaborative effort involving students from both partner universities as well as the Federal University of Pernambuco. Nowadays, a small but active group of collaborators continues to work on its advancement; this framework has been implemented in Scala(v2.12) [9] because of its compatibility with Java programs and its facilities for working with pattern matching. In addition, it relies on the Soot framework(v4.5) [7] as its foundation.

Although JSVFA framework has been successfully applied to detect semantic conflicts [10], an empirical evaluation using the SecuriBench benchmark—comprising 122 unit tests—reveals that the current JSVFA framework implementation succeeds in 63 cases (51.64%), indicating limited completeness. Furthermore, when comparing the JSVFA framework with other established tools such as Joana [11] and FlowDroid [4] on the same benchmark, we observed that FlowDroid achieves the same precision but outperforms JSVFA framework in both recall and F-score; Joana, in contrast, exhibits lower precision but slightly higher recall, resulting in the same percent of F-score as JSVFA framework.

These comparative results are summarized in Table 1.1. The limited completeness of the current implementation is the primary issue driving this research.

Problem Statement: The current implementation of JSVFA framework exhibits limited completeness when assessed against the SecuriBench benchmark, resulting in a pass rate of 51.64%. This limitation reduces its effectiveness in practical static analysis tasks and highlights the necessity for enhancements in its metrics.

Frameworks	**Precision**	**Recall**	**F-score**	**Pass Rate**
JSVFA-v0.3.0	0.88	0.62	0.72	51.64%
Flowdroid	0.90	0.68	0.77	65.05%
Joana	0.82	0.72	0.77	69.67%

Table 1.1: Comparative metrics

A comprehensive report on its metrics is available in the appendices: Apêndice A, Apêndice B, Apêndi
and Apêndice D.

1.2 Research goal

The goal of this thesis is twofold. First, it aims to review technical design decisions behind the JSVFA framework implementation (JSVFA-v0.3.0), and second, it seeks to conduct empirical studies to evaluate the completeness of JSVFA framework and compare its metrics values such as *precision*, *recall*, *f-score*, and *pass rate* against others from static analysis tools, namely Joana and FlowDroid. To accomplish the overall aim of this research, we need to focus on the following specific objectives:

- Conduct a comprehensive literature review on static analysis, taint analysis, and sparse value flow analysis. This review forms the basis of Chapter 2.
- Reverse engineer the JSVFA framework to understand its core technical design decisions and establish a more robust foundation for improving the implementation.
- Fully integrate the JSVFA framework implementation with established benchmarks—SecuriBench [4] and TaintBench [12], which required substantial engineering effort. The task includes adding support for analyzing Android applications, a prerequisite for evaluating the JSVFA framework with the TaintBench benchmark.
- Systematically evaluate JSVFA framework using the SecuriBench and TaintBench benchmarks, assessing the improvements in its metrics resulting from the implementation patches introduced during this research. These empirical evaluations also revealed key limitations and blind spots in the current version of JSVFA framework.

1.3 Research Characterization

Engström et al. present a framework for characterizing research in software engineering through the lens of Design Science Research [13]. According to the authors, they identified five clusters of papers within a dataset of top-evaluated papers published at the International Conference on Software Engineering (2014–2018): *problem-solution pair*, *solution validation*, *solution design*, *descriptive*, and *meta*.

Our work is best characterized as *solution validation*, as it focuses on a new implementation of JSVFA framework and its evaluation, rather than on problem conceptualization or solution design. The general problem addressed by JSVFA framework is already well understood: how to efficiently compute an information flow graph that can be used to detect the leakage of sensitive information. "We draw our motivation for our research from a previous solution and its limitations, not from an observed problem instance."

This study is grounded in a literature review that understands essential concepts and pertinent research within the field. Standard software engineering practices are utilized to analyze and manage legacy source code, including automated testing and debugging techniques. Building on this foundation, we contributed to the implementation of JSVFA framework. We employed quantitative methods to evaluate its completeness and compare it with other tools utilizing established benchmarks.

1.4 Thesis Organization

In the list below, we present the chapters and a summarized description.

- Chapter 2 — Background: introduces concepts of static analysis, covering general approaches to more particular ones, including Taint Analysis and Sparse Value Flow Analysis (SVFA).
- Chapter 3 - Research Method: outlines the research questions and the benchmarks employed in the study. Furthermore, it delineates the methods employed for data collection and analysis to compute essential metrics.
- Chapter 4 - Study Result: addresses the outcomes of our experiments comparing the initial and newer versions of JSVFA framework with the state-of-the-art tools FlowDroid and Joana, followed by a discussion of their improvements and limitations.
- Chapter 5 - Final Remarks: discusses the implications of the findings along with potential threats to the validity of the research.

Chapter 2

Background

This chapter presents the fundamentals of static analysis and examines several foundational algorithms, including *Reaching Definitions* and *Andersen's points-to analysis*, which are critical for Sparse Value Flow Analysis (SVFA). The text addresses essential concepts from *Taint Analysis* and subsequently discusses SVFA, detailing the rules that define the modeling of operations such as *copy*, *load*, *store*, *call*, and *return* within the analysis.

2.1 Static Analysis

Static analysis refers to a class of algorithms designed to construct an abstract representation of a program's behavior without executing the program, thereby obviating the need for a run-time environment. This strategy gives a comprehension of code structure and identifies program behaviors by analyzing the code statically, while additionally attempting to detect potential vulnerabilities through techniques such as Taint Analysis and Data Flow Analysis.

Static analysis might be used to detect potential security weaknesses in code, ranging from SQL injection vulnerabilities to security flaws. Furthermore, it can detect coding standards breaches such as style and code smells, and also highlight performance issues. This approach enhances security by identifying potential exploitable vulnerabilities and security gaps, as well as detecting bugs early, which prevents issues from arising during runtime.

Due to the conservative nature of static analysis, it is expected to produce more false positives than dynamic analysis (such as testing). For instance, a false positive occurs when the algorithm identifies a potential vulnerability that, in reality, does not exist. The occurrence may arise due to the tool assuming a leak in the data flow. In contrast, a false negative arises when vulnerabilities exist, although the tool fails to identify them. This may occur because the analysis is unable to track the trajectory of the data breach. A further barrier is the substantial requirement for processing time and/or memory space that program analysis algorithms require to achieve accuracy and high precision.

2.2 Data Flow Analysis

Data flow analysis is a technique that examines how information flows through a program by tracking the values of variables as they are defined and used. It works by modeling the program as a graph, where nodes represent program statements in the code and edges represent the control flow between them. A set of rules and equations are then used to determine the values of the variables at each location in the source code as the data is propagated through the graph.

Data flow analysis aims to gather information about the possible values a variable may take on at different points (statements) within a program, primarily used to identify dependencies that can be uncovered across distinct segments of code. Moreover, it can analyze variable utilization by detecting code that cannot be executed or contains uninitialized variables, and it can undertake interprocedural analysis across functions within a program.

Control Flow Graph. A control flow graph (CFG) is a directed graph that represents the execution paths of a program. In this representation, *nodes* denote statements, while *edges* correspond to the control flows between statements [14]. Typically, a control flow graph is assumed to possess a singular *entry point* and several *exit points*; a node without incoming edges is designated as an *entry node*, while a node without outgoing edges is referred to as an *exit node.*

Figure 2.1 presents the CFG (*right*) corresponding to the program (*left*), which assesses the value of variable a. If greater than zero, the expression $a+b$ is allocated to the variable r; otherwise, r receives the expression $a - b$.

Any path in the control flow graph that begins at an entry node and terminates at an exit node is referred to as a *control flow path.* Because repetition structures, such as loops, are notoriously unpredictable, there is often more than one feasible control flow path, and in some cases, an infinite number of paths.

Data flow analysis is conducted using algorithms that formulate equations for each node in the control flow graph and resolve them by iteratively computing the output from the input at each node until the entire computation reaches a stable state (i.e., a fixed point). The precise result of this procedure is influenced by several decisions of the data flow algorithm, including the selected type of analysis, which can be either *forward analysis* or *backward analysis.* The former evaluates the flow of data from the

```
  function main() {
s1:  r = 0
s2:  if ( a > 0)
s3:    r = a + b
s4:  else
s5:    r = a - b
s6:  print(r)
  }
```

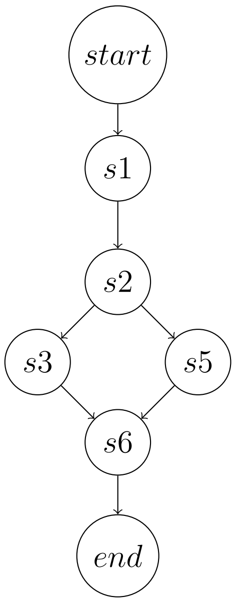


Figure 2.1: Control Flow Graph

initial statement to the final statements. Conversely, backward analysis evaluates the information from the final statements to the initial one. In what follows, we detail the Reaching Definitions algorithm (a core algorithm for the implementation of SVFA) and then give an overview of other data flow analysis algorithms.

Reaching Definitions. It is a forward analysis that aims to identify *"for each program point, which assignments may have been made and not overwritten when the program execution reaches a program point along some path"* [14]. An assignment statement reaches a specific program point if exist at least one path in which no other definition of the statement occurs along that path.

In order to explain the reaching definition analysis, we utilize the terms outlined in Table 2.1 to characterize the functions *gen*, *kill*, *entry*, and *exit* [14]. These functions accept a statement as an argument and yield a set of statements. The algorithm at last computes two sets of tuples that represent the "definitions" entering or exiting a given statement.

Term	Represent
s	Any statement.
X	A single variable.
E	An expression.
CFG	Control Flow Graph.
Definitions of X	A set of assignment statements where the variable X is defined.
Pred(target)	The set of statements S such that the edges $\{(source, target) \mid source \in S\} \in CFG$

Table 2.1: Useful terms for Reaching Definition equations

GEN: If s is an assignment statement, this equation yields a set that includes solely the current statement; if not, it generates an empty set.

$$GEN(s) = \begin{cases} \{X = E\}, & \text{if } s \text{ is an assignment statement} \\ \{\}, & \text{otherwise} \end{cases}$$

KILL: Examining an assignment statement of the form $X = E$ yields a collection of statements in the program that require elimination. Every statement that defines X is included in this set. Only assignment statements *kill* statements. The other ones only generate an empty set.

$$KILL(s) = \begin{cases} \text{Definitions of } X, & \text{if } s \text{ is an assignment like } X = E \\ \{\}, & \text{otherwise} \end{cases}$$

ENTRY: This equation calculates the entry results for the statement s. The set of values derived from the exit of the predecessors of the current statement s is combined using *union* as the meet operator. The reaching definition process is an example of forward analysis. According to the book "Principles of Program Analysis" [14], the initial statement consists of a set of tuples $(x, ?)$ for each variable x in the program, indicating that all variables are uninitialized at the entry of a program function. To facilitate comprehension, we represent the *entry* of the initial node as an empty set.

$$ENTRY(s) = \begin{cases} \{\}, & \text{if } s \text{ is Initial} \\ \cup\, Pred(s), & \text{otherwise} \end{cases}$$

EXIT: This equation computes the exit outcome for the statement s and generates the set of statements derived from the following equation.

$$EXIT(s) = (ENTRY(s) \setminus KILL(s)) \cup GEN(s)$$

We illustrate the *Reaching Definition* analysis using the program in Listing 2.1. Tables 2.2 and 2.3 present the *Entry* and *Exit* sets corresponding to each statement in the program.

Listing 2.1: Reaching Definitions case

```java
public class Main {
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        int c = (new Random()).nextInt();
        if (c > 0) {
            int x = b / c;
            System.out.println(b);
        } else {
            a = b * c;
            System.out.println(a);
        }
    }
}
```

Statement	∪ Pred	**Entry**
a = 1	{}	{}
b = 2	{a = 1}	**{a = 1}**
if (c > 0)	{a = 1, b = 2}	**{a = 1, b = 2}**
x = b / c	{a = 1, b = 2}	**{a = 1, b = 2}**
print(b)	{a = 1, b = 2, x = b / c}	**{a = 1, b = 2,x = b / c}**
a = b * c	{a = 1, b = 2}	**{a = 1, b = 2}**
print(a)	{a = b * c, b = 2}	**{a = b * c, b = 2}**

Table 2.2: Reaching Definition - entries

Statement	entry	kill	gen	**Exit**
a = 1	{}	{a = ?}	{a = 1}	{a = 1}
b = 2	{a = 1}	{b = ?}	{b = 2}	{a = 1, b = 2}
if (c > 0)	{a = 1, b = 2}	{}	{}	{a = 1, b = 2}
x = b / c	{a = 1, b = 2}	{x = ?}	{x = b / c}	{a = 1, b = 2, x = b / c}
print(b)	{a = 1, b = 2, x = b / c}	{}	{}	{a = 1, b = 2, x = b / c}
a = b * c	{a = 1, b = 2}	{a = ?}	{a = b * c}	{a = b * c, b = 2}
print(a)	{a = b * c, b = 2}	{}	{}	{a = b * c, b = 2}

Table 2.3: Reaching Definition - exits

The outcomes of the reaching definition analysis facilitate the construction of def-use and use-def chains, support reasoning regarding variable and constant propagation, and enable the implementation of optimizations like dead code elimination and code hoisting.

Available Expression. It is a forward analysis that aims to find out "for each point in the program, which calculations need to be done beforehand and not changed on any path leading to that point " [14]. An expression is available at a given point if its value has been previously computed and its variables remain unaltered till that point.

Very Busy Expression It is an example of backward analysis that aims to identify "for each program point, which expression must be very busy at the exit from the point" [14]. A very busy expression is an "available expression", which must be utilized in all paths converging at the current point.

Live Variable Another example of backward analysis that aims to determine "for each program point, which variables may be live at the exit from the point" [14]. A variable is considered live at a particular point in the program if its value is utilized prior to that point. In scenarios of redefinition, the variable must be used before it is redefined.

2.3 Pointer Analysis

The objective of pointer analysis algorithms is to determine the set of variables, storage, or memory locations that a pointer variable may reference [2]. That is, these algorithms determine the set of memory locations $\{l_1, l_2, ..., l_n\}$ that each pointer $\{p_1, p_2, ..., p_n\}$ may reference in a program. Two primary strategies exist for implementing points-to analysis algorithms: Andersen and Steensgaard.

Andersen's Algorithm The Andersen approach is a family of algorithms that performs a flow-insensitive and context-insensitive analysis. The algorithm is based on constraints that involve assignment statements—statements of the form "p = q" (see Table 2.4)

Name	**Case**	**Constraint**
Allocation	$p = alloc_i$	$alloc_i \in pt(p)$
Location	$p = \&q$	$q \in pt(p)$
Copy	$p = q$	$pt(q) \subseteq pt(p)$
Load	$p = *q$	$c \in pt(q) \Longrightarrow pt(c) \subseteq pt(p)$
Store	$*p = q$	$c \in pt(p) \Longrightarrow pt(q) \subseteq pt(c)$
Null	$p = null$	$pt(p) = \{\}$

Table 2.4: Andersen's Algorithm constraints

Allocation: A memory location is allocated for a new object ($alloc_i$); thus, this location must be included in the set of points-to from the pointer variable p on the left side of the assignment.

Location: The location indicated by the pointer on the right side, q, is assigned to the pointer on the left, p. Consequently, the location of q is included in the set of points-to from p.

Copy: It is an assignment operation, so the set of points-to from q becomes a subset of the set of points-to from p.

Load: This operation retrieves the points-to set from q, calculates the points-to set for each element, and incorporates them as a subset of the points-to set from p.

Store: This operation retrieves the set of points-to from p, and then each point in that set receives the corresponding set of points-to from q.

Null: This is a deferred operation, resulting in the points-to set associated with the pointer p being replaced by an empty set.

Steensgaard's Algorithm This algorithm performs a rapid flow-insensitive interprocedural analysis and employs term unification, also referred to as unification-based points-to analysis [2]. The method employs a "term variable" to represent each variable or pointer in the program and a "term constructor" ($\Longrightarrow$) to indicate a "pointer to" (Table 2.5). Steensgaard's Algorithm exhibits superior performance compared to Andersen's, yet it demonstrates lower accuracy.

Name	**Case**	**Constraint**
Allocation	$p = alloc_i$	$p \Longrightarrow alloc_i$
Location	$p = \&q$	$p \Longrightarrow q$
Copy	$p = q$	$p = q$
Load	$p = *q$	$p = a \;\&\; q \Longrightarrow a$
Store	$*p = q$	$q = a \;\&\; p \Longrightarrow a$

Table 2.5: Steensgaard's Algorithm constraints

Spark framework The Soot Pointer Analysis Research Kit (Spark) is a versatile framework for experimenting with point-to analyses in Java programs. It is designed to be modular, which means that different versions of its various components can be swapped out. Implementing extra Spark modules allows researchers to efficiently perform existing pointer analysis variations or design new ones. Spark provides support for equality- and subset-based analyses, field sensitivity variations, type recognition, call graph building variations, offline simplification, and various point-to-set propagation algorithms [1].

Spark is a component of the Soot bytecode analysis and optimization framework [7]. Various client analyses within Soot can use the pointer information that Spark computes. It uses the Jimple intermediate representation as input, and Spark's results can be used by other analyses and transformations in Soot [15].

Figure 2.2 describes that the pointer analysis in Spark proceeds in three stages:1) The pointer assignment graph is built based on the program being analyzed, 2) the pointer assignment graph is simplified, and 3) the simplified pointer assignment graph is used to compute points-to. This division into stages is key to the flexibility of Spark that makes implementing a wide variety of analyses as easy as possible. In addition, it is composed of building blocks on which new analyses can be based [15].

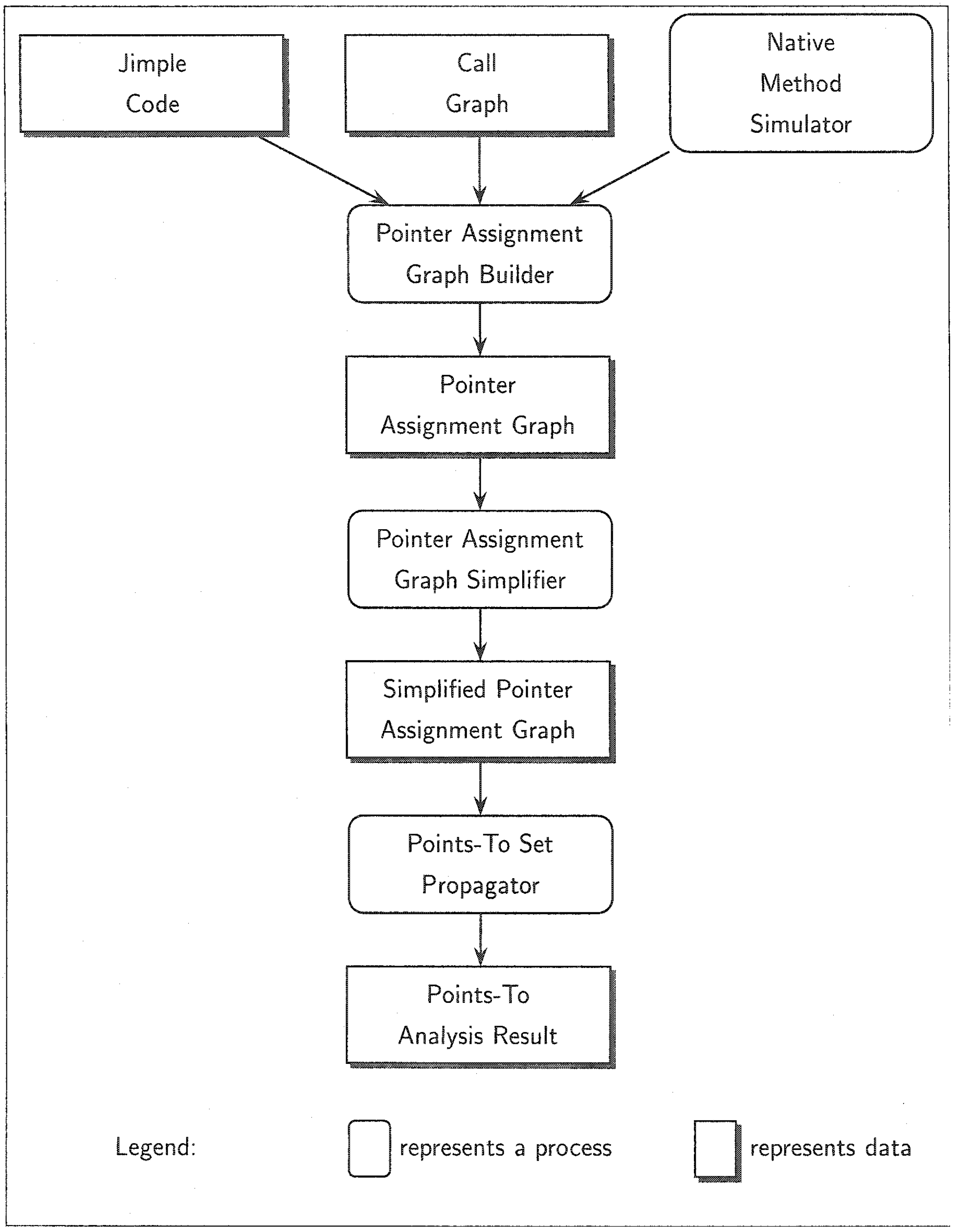


Figure 2.2: SPARK overview [1]

2.4 Taint Analysis

The goal of taint analysis is to track the propagation of potentially harmful data, referred to as *tainted data*, within a program. The system operates by tagging variables that accept sensitive or untrusted input as "tainted" and subsequently monitoring the propagation of the tainted value through assignments or function calls [4].

The objective is to identify the entry points of untrusted data, frequently originating from user input, and to trace any potential paths through which this data could access operations that might “sink” sensitive data or be compromised by untrusted input without being sanitized [12]. As such, the analysis reveals a potential vulnerability when untrusted data reaches a sink without proper sanitation measures.

Tainted analysis addresses security issues that may develop into vulnerabilities, including SQL injection, cross-site scripting (XSS), database calls, buffer overflow, and OS-level invocation. It is also recognized for its capability to identify leaks of sensitive information. This analysis enhances robustness and implements security measures by improving data flows and input sanitization. Taint analysis often considers two special types of statements: Sources and Sinks [4].

Sources: are statements in the code that represent points where potentially unsafe data can enter the system, including user input fields from web forms or external files. A source represents the origin of tainted data.

Sinks: are statements where untrusted data is used and may result in security vulnerabilities, such as the execution of SQL queries or the invocation of native methods from the operating system. A sink acts as a potential point of leakage for tainted data.

Listing 2.2 illustrates a basic example derived from the test suite of SecuriBench. The statement highlighted in gray (`line:3`) represents a *source* statement: user-supplied data is retrieved and assigned to `str`, thereby causing this variable to be immediately tainted by external input. At `line:5`, `str` is transformed using a primitive string operation and assigned to `s2`, which results in the tainting of this new variable as well. Finally, at `line:6`, `s2` is passed as an argument to a method that may lead to a data leak; such a statement is classified as a *sink* statement. Although the example in Listing 2.2 is straightforward, in many real-world scenarios, identifying the path between a source and a sink involves tainted data propagating across multiple functions. In such cases, both “calls” and “returns” are classified as tainted whenever the input or output data was originally marked as such.

Listing 2.2: Taint Analysis: Example

```
1 public class Basic3 extends BasicTestCase implements MicroTestCase {
2     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
          throws IOException {
3         String str = req.getParameter("name");
4         PrintWriter writer = resp.getWriter();
5         String s2 = str.toLowerCase();
6         writer.println(s2);
```

```
    }
}
```

2.5 SVFA

This section provides an overview of the Sparse Value Flow Analysis (SVFA), an interprocedural algorithm for taint analysis that tracks the flow of values within a program. Using SVFA, it is possible to model "tainted information flow," to reveal, for instance, the leakage of sensitive information [3]. In spite of that, Sparse value-flow analysis can identify a wide range of bugs, from simple issues like unreachable code, null pointer dereferences, and input validation flaws including SQL injection, cross-site scripting (XSS), and command injection to more complex problems such as memory management errors like double-free errors, use-after-free vulnerabilities, and buffer overflows/underflows [16], and concurrency bugs such as race conditions, deadlocks, and atomicity violations [17].

According to the literature [3], Sparse Value-Flow Analysis (SVFA) comprises two main steps: (1) performing a points-to analysis and (2) constructing the value-flow graph based on the results of the first step. In this chapter, we discuss two algorithms that constitute the core of the initial stage: *reaching definitions* and *points-to analysis*. Both are essential for computing the information upon which an SVFA implementation depends. This section illustrates the functioning of SVFA using two examples from the SecuriBench benchmark [18] reflected in Listing 2.3 and Listing 2.4, which, for the sake of simplicity, focus on tracking value flows in variables rather than pointers.

Listing 2.3 illustrates a typical XSS vulnerability that directly outputs user input (a `source` at Line 5), creating a leakage of sensitive data on Line 8. The flow begins by retrieving a query parameter from the HTTP request using the key `NAME`, which is then assigned to the variable `name`. Subsequently, it assigns the value of `name` to the variable `str`, and finally it writes the string `str` (containing the value from the request parameter) directly to the HTTP response body via an instance of the class `PrintWriter`.

Listing 2.3: SVFA: Intraprocedural XSS vulnerability

```
1 public class Aliasing1 extends BasicTestCase implements MicroTestCase {
2   private static final String NAME = "name";
3
4    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
         throws IOException {
5       String name = req.getParameter(NAME);
6       String str = name;
```

```
        PrintWriter writer = resp.getWriter();
        writer.println(str);  /* BAD */
    }
}
```

Listing 2.4 presents how an input flows through a method and reaches an output. Using the key `NAME`, the process first extracts a query parameter from the HTTP request and allocates it to the variable `s1`. Then, variable `s2` passes the user input through the method `id(...)`, and variable `s3` calls `id(...)` with a hard-coded string `abc`. Lastly, two values are sent back to the browser using the variable `writer`. While the second variable, `s3`, is a constant string (safe), the first one, `s2`, depends on user input (vulnerable). The aforementioned `id(...)` function is a trivial identity method, meaning it just returns the input.

Listing 2.4: SVFA: Interprocedural XSS vulnerability

```
public class Inter1 extends BasicTestCase implements MicroTestCase {
    private static final String NAME = "name";

    protected void doGet(HttpServletRequest req, HttpServletResponse
        resp) throws IOException {
        String s1 = req.getParameter(NAME);
        String s2 = id(s1);
        String s3 = id("abc");
        PrintWriter writer = resp.getWriter();
        writer.println(s2);  /* BAD */
        writer.println(s3); /* OK */
    }

    private String id(String string) {
        return string;
    }
}
```

2.5.1 Rules

Sparse value flow analysis uses a group of rules based on assignment statements in the form `lhs = rhs`, as shown in Table 2.6, which includes these representations: `s` = statement, `s'` = any earlier statement, `o` = object, `pt(v)` = points-to of v, `cs` = call statement, `rs` = return statement, and `is` = initial statement of a method.

Name	**Case**	**Constraint**
Copy	$s : p = q$	$q@s' \rightarrow p@s$
Load	$s : p = *q$	$o@s' \rightarrow p@s, \forall\, o \in pt(q)$
Store	$s : *p = q$	***Strong***
		$q@s' \rightarrow o@s, \forall\, o \in pt(p)$
		Weak
		$pt(o)@s' \rightarrow o@s, \forall\, o \in pt(p)$
Call	$cs : r = f(..., p, ...)$	***Direct***
	$is : f(..., q, ...)$	$p@s' \rightarrow q@is$
		Indirect
		$o@s' \rightarrow q@is, \forall\, o \in pt(p)$
Return	$cs : r = f(..., p, ...)$	***Direct***
	$is : f(..., q, ...)$	$x@s' \rightarrow x@rs \rightarrow r@cs$
	...	***Indirect***
	$rs : return\ x$	$o@s' \rightarrow x@rs, \forall\, o \in pt(x) \rightarrow r@cs$

Table 2.6: Sparse Value-Flow rules

Copy Rule: This rule involves assigning values from a non-trivial expression (i.e., contains a reference to a variable, for instance) on the right to the variable on the left.

Load Rule: It calculates all possible allocations of objects that a pointer, on the right side, may reference and allocates each object to the pointer on the left.

Store Rule: It identifies the allocation sites to which a pointer on the left may refer and assigns the value on the right to each of these sites. Store operations can be categorized into two types: *weak* and *strong.*

Call Rule: It addresses interprocedural scenarios and handles the information flow when variables are passed as arguments in a method call. Call operations can be categorized into two types: *direct* and *indirect.*

Return Rule: It illustrates the flow of a variable in the presence of a return statement, resulting in two edges: one from the definition of the return variable to the return statement, and another from the return statement to the point where the method was invoked. Return operations can be categorized into two types: *direct* and *indirect.*

2.5.2 Graph Representation

This section shows the flow of data within a method (*intraprocedural*) and between methods (*interprocedural*), as well as how Sparse Value Flow analyses can represent these flows as a graph.

Intraprocedural SVFA Graph Figure 2.3 illustrates two flows converging at the variable d at Line 5 on the left side. Variable a at Line 2 flows to variable c at Line 4 and subsequently to variable d at Line 5. Additionally, variable b at Line 3 also flows to variable d at Line 5. The right side of Figure 2.3 illustrates the flow of these values within a function body.

```
1 function main() {
2     a = 1
3     b = 2
4     c = a
5     d = c + b
6     print d
7 }
```

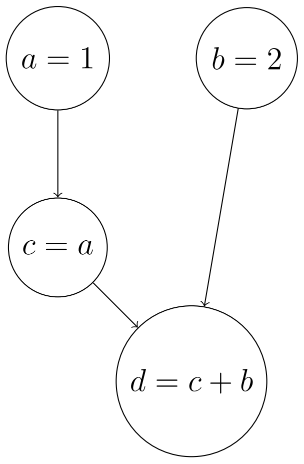


Figure 2.3: Intraprocedural SVFA

Interprocedural SVFA Graph Figure 2.4 illustrates the flow of values within a function body and the call of another function, detailing the processes of value passing and returning. The process initiates when `a` is passed as an argument to the function `id()`, where it is assigned to the variable `x`. At Line 2, `x` is returned and assigned to `c` at Line 8. Subsequently, at Line 9, the addition operation assigns the value of `c` to variable `d`, which is then leaked at Line 10.

```
function id(x) {
    return x
}

function main() {
    a = 1
    b = 2
    c = id(a)
    d = c + b
    print d
}
```

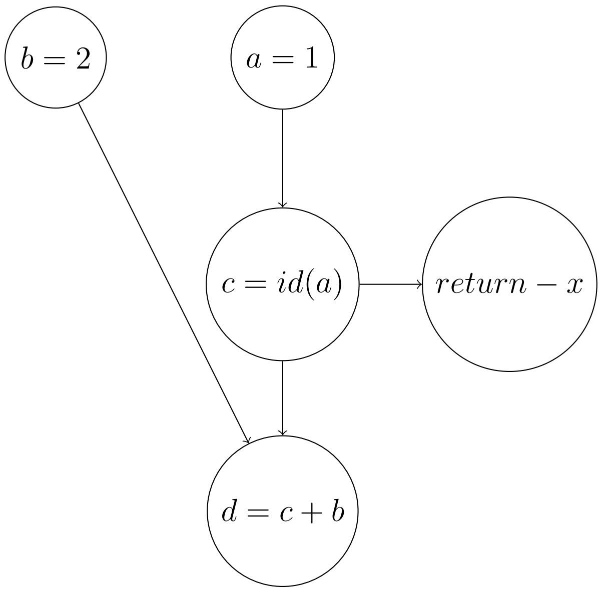


Figure 2.4: Interprocedural SVFA

Chapter 3

Research Method

This research seeks to empirically assess the completeness of JSVFA-v0.3.0 and improve its design and implementation. This study aims to investigate the following research questions:

RQ1 What is the comparison of JSVFA-v0.3.0 with state-of-the-art tools like FlowDroid and Joana regarding standard completeness metrics?

RQ2 What are the primary limitations of JSVFA-v0.3.0 that diminish its completeness in comparison to FlowDroid and Joana?

RQ3 Following the identification and fixing of the primary limitations of JSVFA-v0.3.0, how does JSVFA-v0.6.0 compare to its previous version JSVFA-v0.3.0 and to the tools FlowDroid and Joana?

RQ4 What is the completeness of JSVFA-v0.6.0 within the Android context?

To address the research questions **RQ1**, **RQ3**, and **RQ4**, we leverage the metrics *Pass Rate*, *Precision*, *Recall*, *F-score*, and others, which allow for the assessment, comparison, and evaluation of the completeness of JSVFA framework. The first metric highlights the proportion of executed tests that have successfully passed, while the subsequent metrics concentrate on the comparison between predicted values and actual outcomes. The specified metrics are calculated as follows:

Pass Rate: Calculate the proportion of test cases that have yielded a successful result.

*Pass Rate = (Passed Test Cases / Total Executed Test Cases) * 100*

E.g., The total number of passed test cases using JSVFA-v0.3.0 is 63. Considering that SecuriBench contains 122 tests, the *pass rate* is 55.34%.

Precision: Calculate the proportion of true positive values in the results. The calculation involves dividing the number of true positives by the total number of positive results, which includes both true and false positives.

Precision = TP / (TP + FP)

E.g., After running JSVFA-v0.3.0 using SecuriBench, it reports 80 true positives (TP) and 11 false positive (FP), resulting in a *precision* of 0.88%.

Recall: Calculate the proportion of values from the results that were accurately identified as positives. The calculation involves dividing the number of true positives by the total of true positives and false negatives.

Recall = TP / (TP + FN)

E.g., Upon executing JSVFA-v0.3.0 with SecuriBench, the results indicate 80 true positives (TP) and 50 false positives (FP), yielding a recall of 0.62%.

F-score: This can be understood as the harmonic mean of precision and recall.

*F-score = (2 * (Precision * Recall)) / (Precision + Recall)*

E.g., By using the values from the last two metrics obtained while running JSVFA-v0.3.0 with SecuriBench, we achieved a *precision* of 0.88% and a recall of 0.62%, resulting in an *f-score* of 0.72%.

The second research question, **RQ2**, is addressed through a qualitative analysis that identifies the limitations of JSVFA-v0.3.0 and suggests possible improvements in its design and implementation. This assessment applies software engineering techniques, including unit testing and debugging, to clarify the limitations of JSVFA-v0.3.0 and how they are related to static analysis taxonomies as described in the literature [19].

This research consists of two primary studies. The first is exploratory research, aimed at identifying the limitations of JSVFA-v0.3.0, while the second is a quantitative research approach that evaluates the completeness of both the initial and newer versions of JSVFA framework in comparison to other taint analysis tools through the use of specific benchmarks: SecuriBench and TaintBench.

3.1 Benchmark 01: SecuriBench

This benchmark evaluates the effectiveness of taint-analysis tools, with FlowDroid serving as one of the first to systematically customize and utilize its collection of test cases for benchmark evaluation. SecuriBench comprises 123 Java test cases, categorized into 12 distinct categories: Aliasing (6), Arrays (10), Basic (42), Collections (15), Data Structures (6), Factory (3), Inter (14), Pred (9), Reflection (4), Sanitizer (6), Session (3), Strong Update (5). The tests stress static-analysis tools by exploring field sensitiveness, object sensitiveness, and trade-offs in access-path lengths, among others [20]. Listing 3.1 presents a test from the *Basic* category, which contains a Cross-Site Scripting (XSS) vulnerability in the main method (`doGet`). Furthermore, the example includes two auxiliary methods that provide the test description and the count of vulnerabilities that need to be identified.

Listing 3.1: Basic1.java

```java
public class Basic1 extends BasicTestCase implements MicroTestCase {
    protected void doGet(HttpServletRequest req, HttpServletResponse
        resp) throws IOException {
        String str = req.getParameter("name");
        PrintWriter writer = resp.getWriter();
        writer.println(str);
    }

    public String getDescription() {
        return "very simple XSS";
    }

    public int getVulnerabilityCount() {
        return 1;
    }
}
```

Note: previous research [4, 11] has incorporated SecuriBench into tools such as FlowDroid and Joana; however, none of these studies have clearly outlined the necessary steps for reproducing their results. Consequently, we opted to implement and document the integration of SecuriBench into the mentioned tools.

The tests from the SecuriBench benchmark have been integrated by implementing a generic architecture that facilitates the incorporation of external tools. This architecture supports multiple test suites via concrete classes that must override abstract methods, enabling features such as dynamic test discovery, tool analysis, and reporting.

Architectural Components. This architecture relies on an abstract class with a multi-layered inheritance structure that follows the Template Method pattern. There are two abstract methods in the abstract class, which require concrete implementation: *basePackage()* to set the Java package containing test cases, and *entryPointMethod()* to set the name of the method to be analyzed. The architectural components implement a three-phase process pipeline (Figure 3.1) for data collection.

- Dynamic Test Discovery: conducts a runtime search using Java reflection to identify test classes, including those in nested packages. This logic can be tailored to filter by specific concrete classes, e.g., MicroTestCase.
- Tool Analysis: involves the execution of concrete implementations that run the static analysis from tools such as FlowDroid or JSVFA framework on each available test case.
- Reporting: outputs the results of the analysis in a consistent format that summarizes vulnerability detection and other metrics.

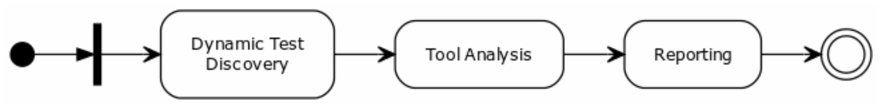


Figure 3.1: Generic workflow for data collection

JSVFA framework integration with SecuriBench. The JSVFA framework integration utilizes `SecuribenchRuntimeTest` as its primary abstract class. Figure 3.2 presents an overview of the class hierarchy. This integration relies on the class `SecuribenchTest`, whose primary purpose is to implement the logic for executing the analysis. Consequently, it depends on the `SecuribenchSpec` class, which defines the list of source and sink statements, and the abstract class `JSVFATest`, which establishes a uniform JSVFA framework configuration and fills some abstract methods declared in the base abstract class `JSVFA`. This class provides the foundational implementation of JSVFA framework features.

Listing 3.2 illustrates the integration algorithm and emphasizes three key steps. First, the statement `svfa = new SecuribenchTest(...)` instantiates a concrete implementation of the SVFA analysis. Next, the core execution of the algorithm is triggered with the call `svfa.buildSparseValueFlowGraph()`. Finally, vulnerability detection and the report of leakages are performed through the statement `conflicts = svfa.reportConflictsSVG()`.

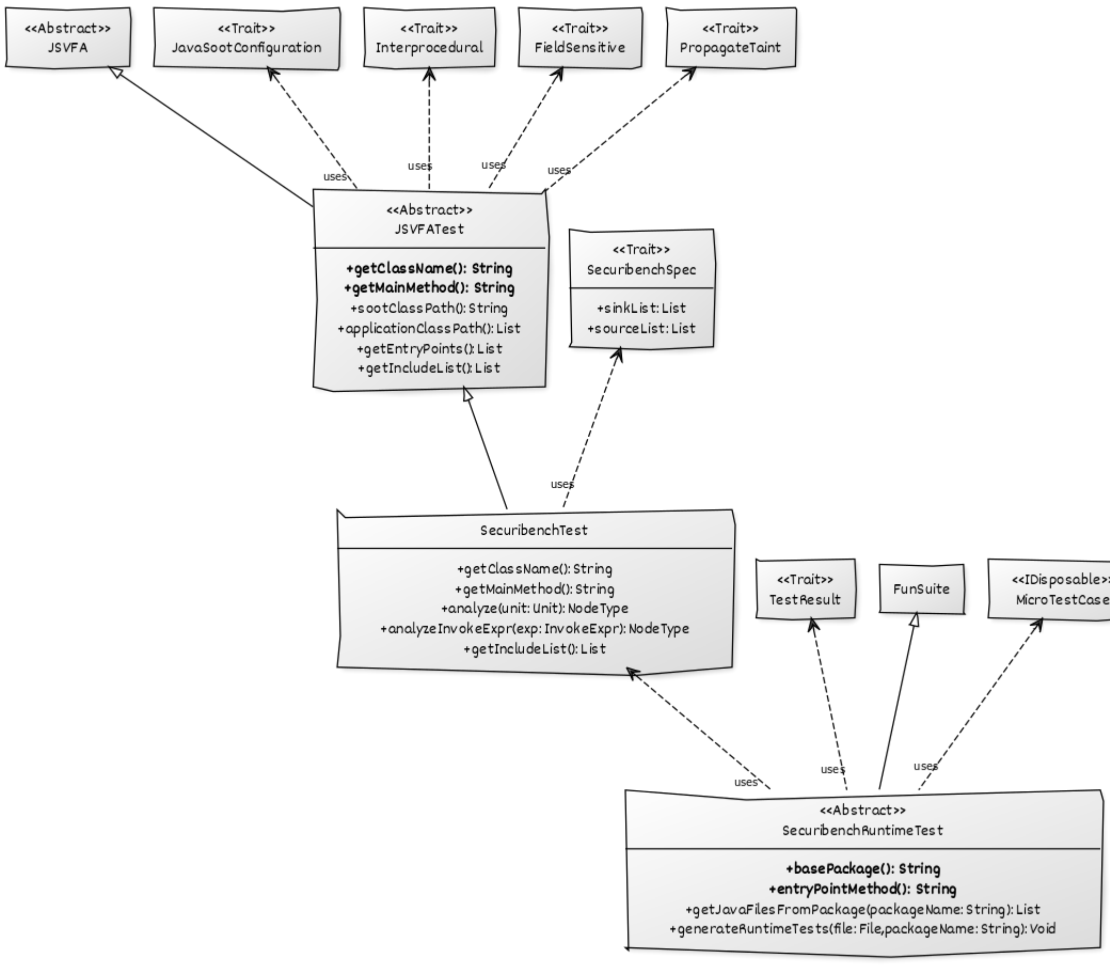


Figure 3.2: Class hierarchy of the JSVFA integration with the SecuriBench

Listing 3.2: SecuribenchRuntimeTest.scala

```scala
def generateRuntimeTests(file: AnyRef, packageName: String): Unit = {
  val svfa = new SecuribenchTest(className, entryPointMethod())
  svfa.buildSparseValueFlowGraph()
  val conflicts = svfa.reportConflictsSVG()
}
```

All concrete classes derive from the primary abstract class `SecuribenchRuntimeTest` and provide implementations for its two abstract methods. Listing 3.3 presents a concrete example for the *Basic* category of test cases in SecuriBench. In this implementation, the base package is configured as `securibench.micro.basic`, and the entry point method is specified as `doGet`.

Listing 3.3: SecuribenchBasicTest

```scala
class SecuribenchBasicTest extends SecuribenchRuntimeTest {
  def basePackage(): String = "securibench.micro.basic"
  def entryPointMethod(): String = "doGet"
}
```

FlowDroid integration with SecuriBench. The FlowDroid integration is fundamentally based on the `SecuriBenchTestCase` abstract class. Figure 3.3 provides an overview of the class hierarchy. This integration relies on the abstract class `JUnitTests`, which encompasses the majority of the implementation logic, including the definitions of source and sink lists, the implementation of test utilities for checking analysis results, the configuration of application and library paths, and the initialization and configuration of the taint analysis engine via `Infoflow`.

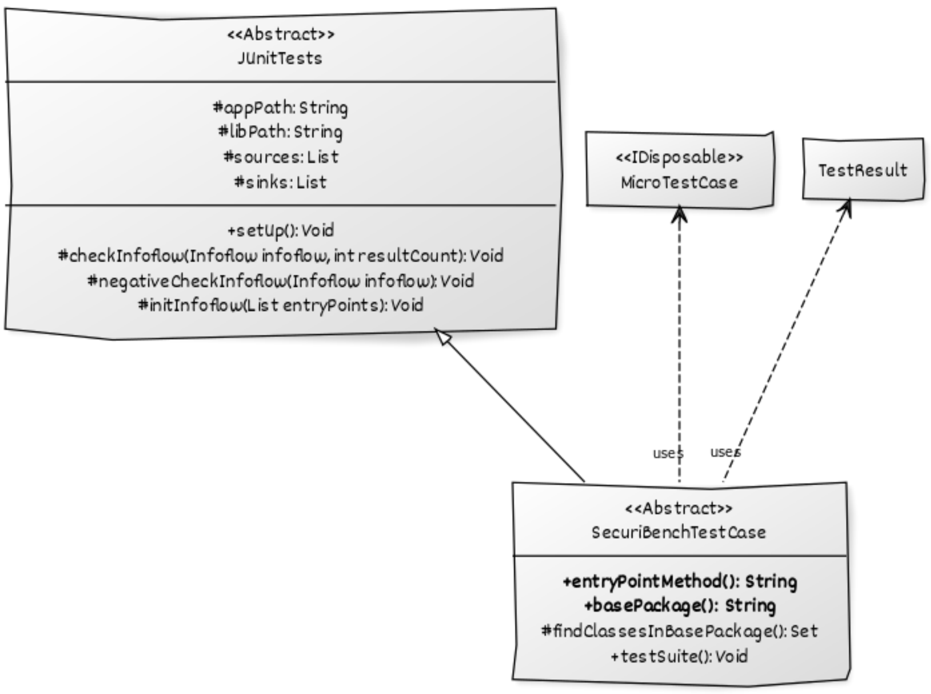


Figure 3.3: Class hierarchy for the SecuriBench integration in FlowDroid

Listing 3.6 provides an overview of the integration that utilizes FlowDroid (Soot-based) for tracking taint propagation within the application. The result is achieved by establishing the entry point method through `initInfoflow(epoints)`, followed by executing the analysis via the `computeInfoflow()` method call, which relies on computing data flow from sources and sinks. Overall, it produces results for identifying potential vulnerabilities.

Listing 3.4: SecuriBenchTestCase.java

```java
public final void testSuite() throws Exception {
    // ... class name processing
    List<String> epoints = new ArrayList<String>();
    epoints.add("<" + testName + ": " + entryPointMethod() + ">");
    Infoflow infoflow = initInfoflow(epoints);
    infoflow.computeInfoflow(appPath, libPath, entryPointCreator, sources,
        sinks);
    found = infoflow.getResults().size();
    // ... metrics computation
}
```

All concrete classes inherit from the abstract class `SecuriBenchTestCase` and meet its requirements established via abstract methods. Listing 3.5 presents an example that highlights the implementation of the test cases in the *Aliasing* category. The process involves configuring the package containing the test cases to identify (`securibench.micro.aliasing`) and employing the method signature to start the analysis (`doGet()`).

Listing 3.5: AliasingSuiteTest

```java
public class AliasingSuiteTest extends SecuriBenchTestCase {
    @Override
    public String basePackage() {
        return "securibench.micro.aliasing";
    }

    @Override
    public String entryPointMethod() {
        return "void doGet(
          javax.servlet.http.HttpServletRequest,
          javax.servlet.http.HttpServletResponse
        )";
    }
}
```

This integration employs a stable version of FlowDroid (v2.13) and can be accessed at `https://github.com/PAMunb/flowdroid-taint-analysis`.

Joana integration with SecuriBench. The Joana integration uses as its main abstract class the `SecuriBenchTestCase` class. Figure 3.4 illustrates the class hierarchy, which originates from `JoanaTestCase` and implements a template method pattern for executing benchmark tests. An instantiated object from class `Driver` facilitates access to the Joana engine for IFC analysis. In contrast, method `loadConfiguration()` instantiates another object from class `Config` that functions as a YAML configuration manager to define the *sources* and *sinks* methods (vulnerability points).

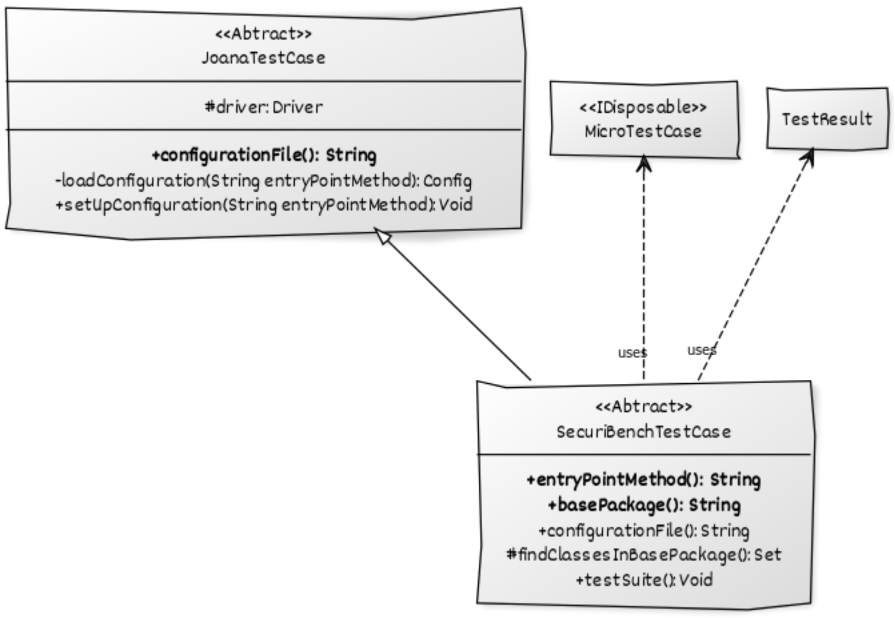

<<Abtract>>
JoanaTestCase
#driver: Driver
+configurationFile(): String
-loadConfiguration(String entryPointMethod): Config
+setUpConfiguration(String entryPointMethod): Void
<<IDisposable>>
MicroTestCase
TestResult
uses
uses
<<Abtract>>
SecuriBenchTestCase
+entryPointMethod(): String
+basePackage(): String
+configurationFile(): String
#findClassesInBasePackage(): Set
+testSuite(): Void

Figure 3.4: Class hierarchy for the SecuriBench integration in Joana

Listing 3.6 presents an overview of the integration, emphasizing two principal aspects: the configuration that is initially established by the method `setUpConfiguration` using the test name and entry point and the analysis that is then executed, yielding the number of vulnerabilities detected.

Listing 3.6: SecuriBenchTestCase.java

```
def generateRuntimeTests(file: AnyRef, packageName: String): Unit = {
  // ... class name processing
    setUpConfiguration(testName + "." + entryPointMethod());
    found = driver.execute().size();
  // ... metrics computation
}
```

All concrete classes derive from the primary abstract class `SecuriBenchTestCase` and implement its abstract method constraints. Listing 3.7 provides a practical example of the test case for the *Collection* category. It involves configuring the package from the test group (`securibench.micro.collections`) and utilizing the entry point method (`doGet`).

Listing 3.7: CollectionTestSuite.java

```
public class CollectionTestSuite extends SecuriBenchTestCase {
    @Override
    public String basePackage() {
        return "securibench.micro.collections";
    }

    @Override
    public String entryPointMethod() {
        return "doGet(
            Ljavax/servlet/http/HttpServletRequest;
            Ljavax/servlet/http/HttpServletResponse;
        )V";
    }
}
```

This integration uses a stable version of Joana and its source code is available at `https://github.com/PAMunb/joanaTaintAnalysis.git`.

3.2 Benchmark 02: TaintBench

This benchmark assesses the efficacy of taint analysis tools in the context of Android applications. The dataset includes a realistic and diverse collection of applications featuring known data-flow vulnerabilities, comprising 39 malicious APKs. This benchmark is compared with other benchmarks, such as DroidBench [20], which primarily utilize small, made-up test cases; however, TaintBench employs real Android applications to facilitate equitable and reproducible evaluations of taint-analysis tools [21]. Many of those applications gather sensitive data from mobile devices and send it to remote servers. Each application includes source–sink pairs that delineate the flow of sensitive information, such as GPS location and internet connection. TaintBench was introduced to the academic community in the paper by Luo et al. [12], which presented six types of experiments utilizing two tools: FlowDroid and Amandroid [22].

This section exclusively addresses the integration of the TaintBench benchmark into the JSVFA framework tool.

Note that, to integrate JSVFA framework with TaintBench, we extended its implementation to implement the analysis of Android applications. This modification represents an additional contribution of this research, as JSVFA framework originally supported only Java bytecode. Figure 3.5 depicts the class hierarchy resulting from this integration. We implemented the concrete class `AndroidTaintBenchSuiteExperimentTest` to maintain the list of tests. This class follows a multi-composition pattern, inheriting from the base class `FunSuite` (provided by the ScalaTest library). It also interacts with `AndroidTaintBenchTest`, the primary test orchestrator responsible for coordinating test execution and managing outcomes. This orchestrator leverages several components:

- `Analysis:` implemented on the abstract class `JSVFA` to perform Sparse Value Flow Analysis;
- `Configuration:` applies the `AndroidSootConfiguration` trait to configure the Soot framework for Android APK analysis, in combination with the traits `PropagateTaint`, `Interprocedural`, and `FieldSensitive` to refine analysis capabilities;
- `Vulnerability Definition:` employs the `TaintBenchSpec` trait to specify a comprehensive set of sources and sinks.

An examination of the functionalities of `AndroidTaintBenchTest`, the primary class in this integration that functions as the analysis orchestrator, highlights the implementation of methods including `getApkPath()`, `apk()`, and `platform()` for the resolution of APK paths and platform configuration. Additionally, the method `readProperty(key: String)` ensures that required properties are established, while the methods `analyze(unit: soot.Unit)` and `analyzeInvokeStmt(exp: InvokeExpr)` facilitate multi-statement support, method call detection, and node type classification (i.e., source and sink nodes).

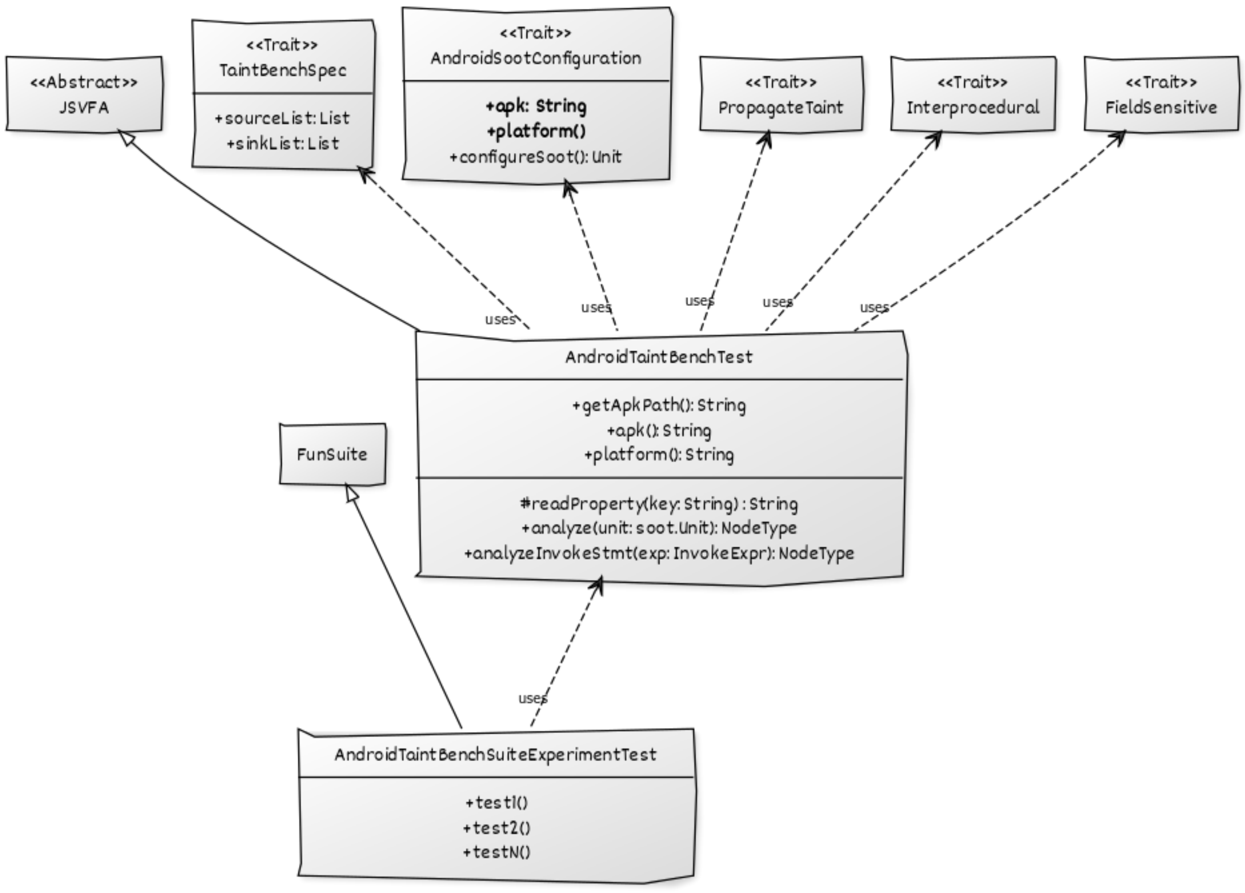


Figure 3.5: Dependency Chain: TaintBench integration in JSVFA framework

The test cases from the TaintBench benchmark have been integrated using a simpler architecture, with each test written separately. In contrast, the architecture for the SecuriBench benchmark loads and executes tests dynamically (Section 3.1). Listing 3.8 shows that each test adheres to a uniform pattern: 1) Define the anticipated outcome and identify the APK for testing; 2) construct an analysis graph and evaluate actual results; and 3) verify correctness.

Listing 3.8: TaintBench test template

```
test("APK name - expected flows") {
  val expected = X
  val nameAPK = "apk_name"

  val svfa = new AndroidTaintBenchTest(nameAPK)
  svfa.buildSparseValueFlowGraph()
  val actual = svfa.reportConflictsSVG().size

  assert(actual == expected)
}
```

3.3 Data Collection

This section outlines the computation of key metric values, including precision, recall, f-score, and pass rate, as well as other metrics such as true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), test outcomes (passed/failed), and both the actual and expected number of findings.

In the context of SecuriBench integration, we have developed a common architecture that enables the collection, storage, and reporting of test-related metrics. This implementation relies on the `TestResult` class, which contains an internal data structure that associates each test, identified by its name, with the corresponding metric values pertinent to that test. Figure 3.6 presents the pipeline for data collection through its primary method, `compute(expected, found)`, which takes the expected and found amounts for a test as parameters. The pipeline systematically computes the metrics for a specific test, including `reportTruePositives()` and `reportPassedTest()`, among other methods. Additionally, a list of methods exists to compute standard evaluation metrics, either for a specific test or aggregated across all tests, including `precision()` and `recall()`, and others. The implementation provides an option to generate a comprehensive metric report, encompassing TP, FP, FN, TN, precision, recall, f-score, test outcomes, and pass rate.

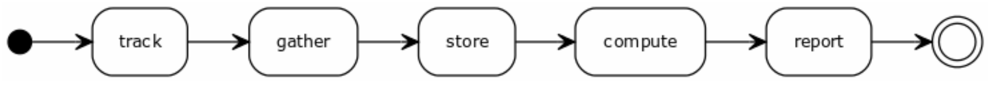


Figure 3.6: Metrics flow

As output, we produce a markdown-style summary table encompassing all tests and overall metrics. The markdown table consists of 10 columns that include *Test* (identifier for the test case), *Found* (count of detected vulnerabilities), *Expected* (anticipated number of vulnerabilities), *Status* (indicator of test results), *TP* (true positives), *FP* (false positives), *FN* (false negatives), *Precision* (metric for precision), *Recall* (metric for recall), and *F-score* (metric for F1-score). This format provides a detailed and uniform report of the assessment, facilitating a comparison of results among test categories and highlighting their strengths and weaknesses. Table 3.1 presents an example of the output metric report for the test category *aliasing*.

With respect to the integrations with JSVFA framework, FlowDroid, and Joana, we have used the already explained data collection architecture. For example, in the JSVFA

`AliasingTest` - failed: 5, passed: 1 of 6 tests - *(16.67%)*.

`TEST`	`FOUND`	`EXPECTED`	`STATUS`	`TP`	`FP`	`FN`	`PRECISION`	`RECALL`	`F-SCORE`
Aliasing1	1	1	YES	1	0	0	1.00	1.00	1.00
Aliasing2	0	1	NO	0	0	1	0.00	0.00	0.00
Aliasing3	0	1	NO	0	0	1	0.00	0.00	0.00
Aliasing4	2	1	NO	0	1	0	0.00	0.00	0.00
Aliasing5	0	1	NO	0	0	1	0.00	0.00	0.00
Aliasing6	1	7	NO	0	0	6	0.00	0.00	0.00
`TOTAL`	**4**	**12**	**1/6**	**1**	**1**	**9**	**0.50**	**0.10**	**0.17**

Table 3.1: JSVFA framework metric report for Aliasing tests

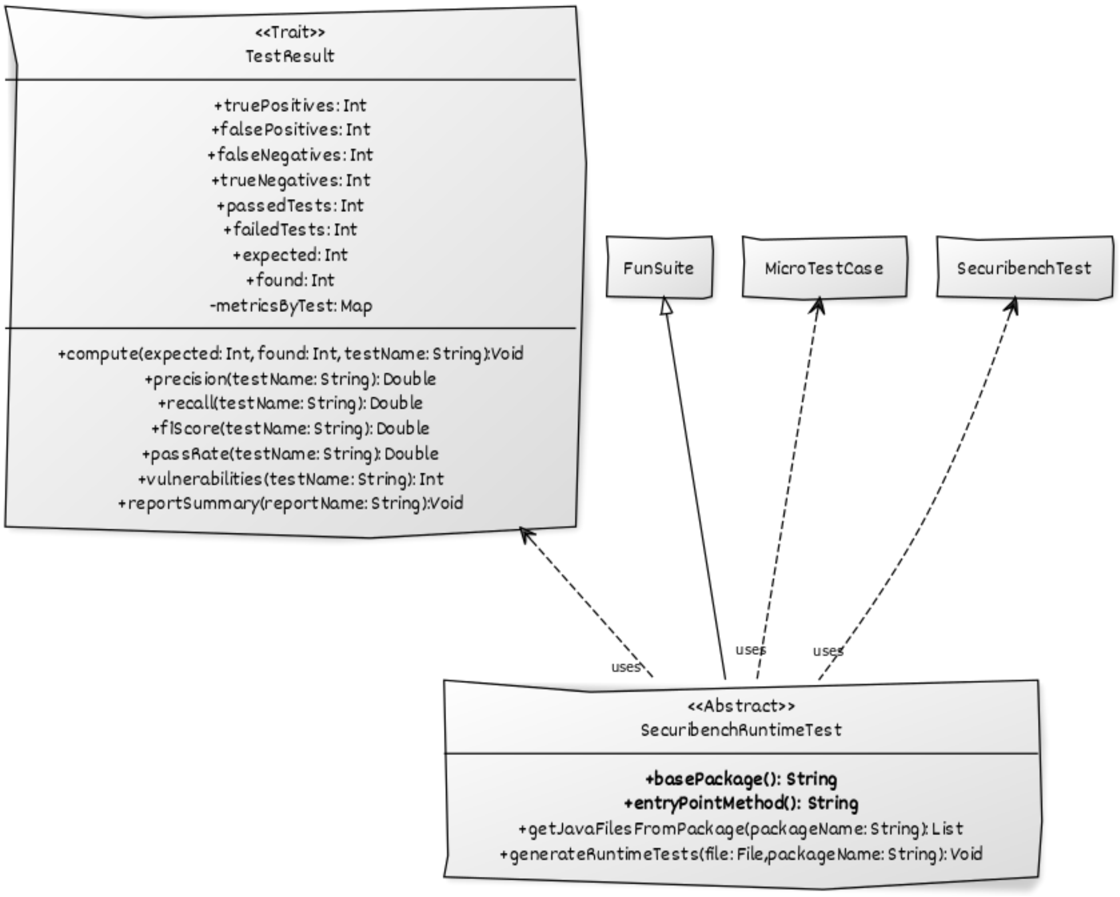
framework, we have implemented a trait, called `TestResult`, which is integrated into `SecuribenchRuntimeTest` to evaluate the tool. This component is intended for incorporation into the SecuriBench test suite for the purpose of monitoring and reporting metrics. Figure 3.7 illustrates the class hierarchy, featuring a "case class" named `TestResult`, which facilitates the construction of structures for the storage of collected information. The data collection architecture for FlowDroid and Joana is instantiated using a similar design.

Conversely, regarding TaintBench integration, we changed the approach to computing metrics, as in SecuriBench. This integration compares the matches between the set of leaks reported by each APK execution in the paper [12], termed "expected," and the number of leaks identified by JSVFA framework, referred to as "actual." A "match" occurs when the source and sink are identical in both the actual and expected sets.

The analysis is performed for each APK from TaintBench, with results presented in a table comprising 4 columns: *APK* (the name of the Android application package analyzed), *actual-findings* (the count of detected taint flows/vulnerabilities), *expected-findings* (the anticipated number of taint flows to be detected—ground truth), and *matches* (the count of findings that accurately correspond to the expected results). This format offers a comprehensive and standardized report of the assessment, enabling the monitoring of tool effectiveness and the identification of areas for improvement in the analysis. Table 3.2 displays a partial result of Experiment-I, which compares the actual results of the taint analysis tool JSVFA framework with the expected ground truth.

3.4 Data Analysis

We use "descriptive statistics" to perform the data interpretation. This technique focuses on summarizing and describing the main features of a dataset, which makes complex data understandable. [23]. Its key aspects include metrics of central tendency (mean, median, mode), metrics of variability (range, variance, standard deviation), and data



<<Trait>>
TestResult
+truePositives: Int
+falsePositives: Int
+falseNegatives: Int
+trueNegatives: Int
+passedTests: Int
+failedTests: Int
+expected: Int
+found: Int
-metricsByTest: Map
+compute(expected: Int, found: Int, testName: String):Void
+precision(testName: String): Double
+recall(testName: String): Double
+f1Score(testName: String): Double
+passRate(testName: String): Double
+vulnerabilities(testName: String): Int
+reportSummary(reportName: String):Void
FunSuite
MicroTestCase
SecuribenchTest
uses
uses
uses
<<Abstract>>
SecuribenchRuntimeTest
+basePackage(): String
+entryPointMethod(): String
+getJavaFilesFromPackage(packageName: String): List
+generateRuntimeTests(file: File, packageName: String): Void

Figure 3.7: Dependency Chain: SecuriBench integration in JSVFA framework

APK	actual	expected	matches
jollyserv	1	1	0
phospy	1	4	1
cajino_baidu	8	14	1
fakedaum	3	2	0
roidsec	3	4	0
...	...	...	...

Table 3.2: TaintBench metric report for Experiment-I

visualization through charts and graphs. [24]. Descriptive statistics form a basis for all quantitative analysis and are a precursor for inferential statistics, which use properties of a data set to make inferences and predictions beyond the data. Depending on the scope of a project, descriptive statistics alone may be sufficient, or they can be used to inform further statistical analyses. [25]

This research employs two components of descriptive statistics: metrics of central tendency, which utilize metrics such as precision and recall, as outlined in the introduction to this section, and data visualization, including tables and bar charts, to illustrate the patterns within the data. This technique is employed to provide a summary of the primary characteristics of tool completeness. Descriptive methods, as outlined, do not test hypotheses; instead, they offer a foundational overview.

Chapter 4

Study Result

This chapter presents the study results, followed by a discussion of their enhancements and limitations. Initially, we conduct *study-I*, which evaluates and analyzes the preliminary completeness of JSVFA framework utilizing the SecuriBench benchmark. Subsequently, we proceed with *study-II*, in which we identify and rectify various bugs to release an updated version of JSVFA framework. Next, we compare this new version to the old one to see if there are any improvements. Additionally, we conduct *study-III* to investigate the completeness of the new version of JSVFA framework within mobile application domains, utilizing a benchmark referred to as TaintBench.

4.1 Study I

This study has the goal to answer **RQ1** and **RQ2**. We aim to evaluate the completeness of the initial version of the JSVFA-v0.3.0 framework using the SecuriBench benchmark. It collects metrics, including precision, recall, f-score, and pass rate, to obtain insights into potential issues. Furthermore, we evaluate the JSVFA-v0.3.0 results in comparison to two established static analyzers: FlowDroid and Joana. We incorporated all test categories from SecuriBench to evaluate the positioning of JSVFA framework in relation to the other analyzers based on their metrics. This study's findings reveal that, of the 122 test cases executed, JSVFA-v0.3.0 failed in 63, yielding a pass rate of 51.64%. SecuriBench is insufficiently addressed by JSVFA-v0.3.0 in scenarios from different test case categories.

Figure 4.1 presents a comparison of the static analysis tools in the SecuriBench, indicating that JSVFA-v0.3.0 demonstrates lower precision levels (88%) compared to FlowDroid (90%). Nonetheless, the minimal difference suggests that JSVFA-v0.3.0 addresses a low rate of false positives. Furthermore, FlowDroid exhibits a slightly enhanced capacity to mitigate false negatives, achieving a recall rate of 68%. On the other hand, JSVFA-v0.3.0 missed 59 leaks, which led to a recall rate of 62%. Concerning the f-score,

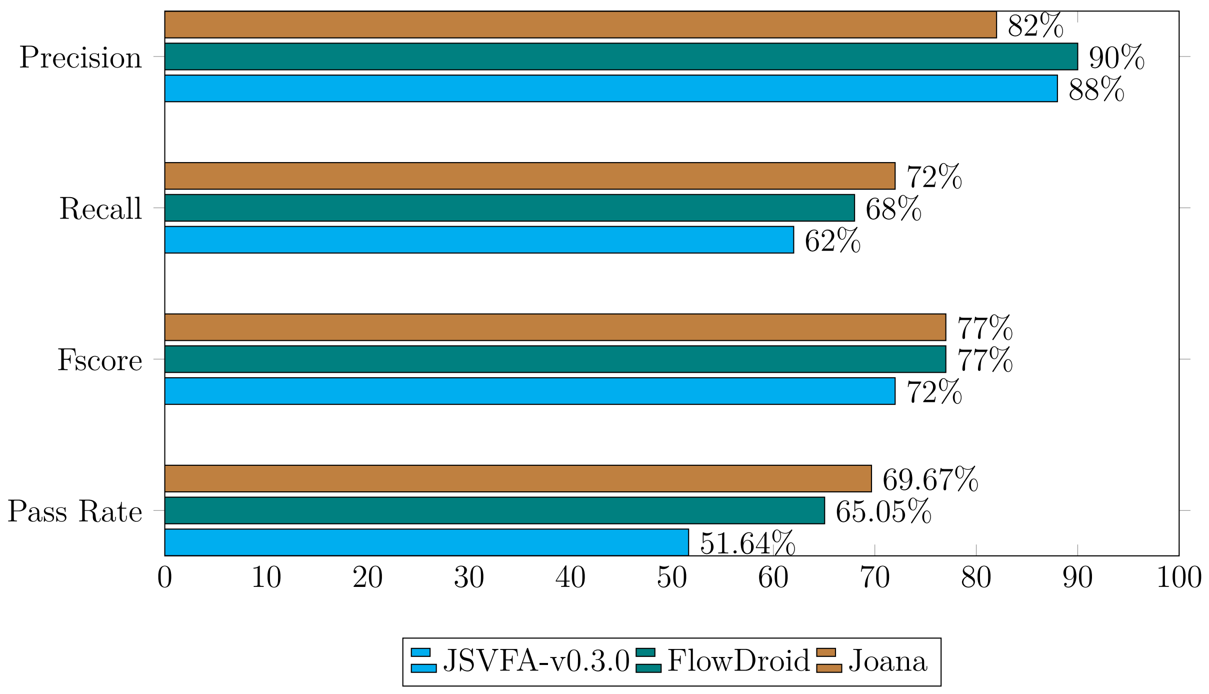


Figure 4.1: JSVFA-v0.3.0 comparison against FlowDroid and Joana

FlowDroid exhibits a 5% higher value compared to JSVFA-v0.3.0. The pass rate metric demonstrates that FlowDroid execution achieves a higher percentage of passing test cases at 65.05%, in contrast to 51.64% for JSVFA-v0.3.0.

In contrast to Joana, JSVFA-v0.3.0 exhibits 88% precision, which means it is 6% superior to Joana and indicates a higher likelihood of classifying correctly a result as positive. Joana outperforms JSVFA-v0.3.0 in recall, achieving 72% compared to 62%, by predicting a lower number of false negatives. Furthermore, JSVFA-v0.3.0 yields an f-score of 72%, while Joana yields 77%. In terms of the number of tests completed, Joana surpasses JSVFA-v0.3.0, achieving a pass rate of 69.67%, whereas JSVFA-v0.3.0 attains only 51.64% of this metric.

The second research question, **RQ2**, addresses the potential limitations of JSVFA framework that have adversely affected its correctness, specifically an inadequate pass rate of 51.64%. We categorize and describe these limitations based on their specific categories.

The category *Aliasing* includes six test cases, with two (*Aliasing1* and *Aliasing6*) successfully handled by JSVFA-v0.3.0. The remaining cases do not succeed, as demonstrated in the test case `Aliasing5` (Listing 4.1), which depicts an interprocedural argument aliasing scenario with a unique leak. However, JSVFA-v0.3.0 fails to report this leakage, leading to a false negative.

Listing 4.1 presents an interprocedural parameter aliasing case, which leads to an XSS vulnerability. At `line:3`, the same object (*buf*) is passed twice to the `foo` method—once as *buf* and once as *buf2*. This means both parameters refer to the same mutable object. Then, at `line:8`, the a variable (*source*) is appended to *buf*, and due to *buf* and *buf2* referencing the same underlying `StringBuffer`, the modification done through *buf* is also visible through *buf2*. Finally, at `line:10`, a method that may lead to a leak (*sink*) is invoked and receives the content of variable *buf2* as a parameter.

Listing 4.1: Aliasing5.java

```
1  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
       throws IOException {
2      StringBuffer buf = new StringBuffer("abc");
3      foo(buf, buf, resp, req);
4  }
5
6  void foo(StringBuffer buf, StringBuffer buf2, ServletResponse resp,
       ServletRequest req) throws IOException {
7      String name = "req.getParameter("name");
8      buf.append(name);
9      PrintWriter writer = resp.getWriter();
10     writer.println(buf2.toString());
11 }
```

The category *Array* comprises 10 test cases, of which only one, *ArrayTest5*, is successfully managed by JSVFA-v0.3.0. In contrast, the remaining cases fail, as illustrated in the test case `ArrayTest1` (Figure 4.2), which presents a simple array manipulation scenario featuring a single leak (path). However, JSVFA-v0.3.0 does not report this leakage, resulting in a false negative.

In Figure 4.2, at `line:2`, a value (*source*) is assigned to the variable *n*. At `line:4`, this variable is assigned to the first index of the array (`array[0]`). Finally, at `line:6`, a method that may lead to a leak (*sink*) is invoked and receives the first value of the array as a parameter. Thus, a path exists between the "source" and the "sink." The right side of Figure 4.2 presents the SVG graph produced by the JSVFA-v0.3.0 framework, in which each node corresponds to a statement, and it indicates the absence of a path between the source (*s2*) and the sink (*s6*).

Category *Basic* comprises 42 test cases, with 36 handled effectively by JSVFA-v0.3.0 and 6 not managed successfully. Listing 4.2 illustrates a scenario involving a basic heap-allocated data structure that expects one memory leak; however, JSVFA-v0.3.0 identifies two, resulting in a false positive outcome.

```
function main() {
    String n = "req.getParameter("name");
    String[] array = new String[10];
    array[0] = n;
    PrintWriter writer = resp.getWriter();
    writer.println(array[0]);
}
```

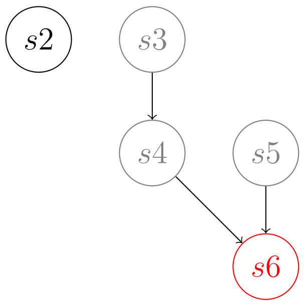


Figure 4.2: Case: ArrayTest1

Listing 4.2 provides a clearer view of the context scenario in which two objects of the same type assign values to themselves; one value is tainted (`line:4`) while the other is not (`line:7`). Subsequently, both getter methods are called, resulting in the output of their respective object content. JSVFA-v0.3.0 cannot distinguish between the method calls and categorizes both as tainted.

Listing 4.2: Basic17.java

```
1  protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
2      String s = req.getParameter("name");
3      Widget w1 = new Widget();
4      w1.setContents(s);
5  
6      Widget w2 = new Widget();
7      w2.setContents("abc");
8  
9      PrintWriter writer = resp.getWriter();
10     writer.println(w1.getContents());
11     writer.println(w2.getContents());
12 }
```

The category *Collections* contains 14 test cases, of which only one was successfully managed by JSVFA-v0.3.0, while the others resulted in failure. Listing 4.3 illustrates a failing test of iterators that involves looping structures, which expects one leak. However, JSVFA-v0.3.0 shows no leakages, resulting in a false negative outcome.

Listing 4.3 presents a test of iterators; it reads untrusted input at `line:2`, stores this value in a `LinkedList` at `line:4`, and subsequently iterates over the list at `line:6`. Within the loop, the list element is accessed using `next` at `line:8` and then printed at `line:9`, resulting in a vulnerability.

Listing 4.3: Collections4.java

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    String s = req.getParameter("name");
    LinkedList ll = new LinkedList();
    ll.addLast(name);

    for(Iterator iter = ll.iterator(); iter.hasNext();) {
        PrintWriter writer = resp.getWriter();
        Object o = iter.next();
        writer.println(o);
    }
}
```

The category *Datastructures* comprises six test cases, with four successfully handled by JSVFA-v0.3.0 and two not managed effectively. Listing 4.4 presents a failing test involving a straightforward case of nested data, which expects no leaks. Nonetheless, JSVFA-v0.3.0 indicates one leakage, leading to a false positive outcome.

Listing 4.4 presents simple nested data (false positive). At `line:2`, it retrieves a request parameter (*name*), which is stored inside *c1* via the `setData` method at `line:4`. Then, at `line:7`, a constant string (*abc*) is stored in a second object (*c2*). Consequently, object *c1* establishes a link to the new object at *line:8*. Finally, a value is extracted at `line:9`, with *c1* navigating to the next object (*c2*) and reading the field *str*, which contains *abc* and is printed at `line:13`. Due to *str* originating from a constant value, not from user input, this line is safe.

Listing 4.4: Datastructures4.java

```java
 1  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
 2     String n = req.getParameter("name");
 3     C c1 = new C();
 4     c1.setData(name);
 5
 6     C c2 = new C();
 7     c2.setData("abc");
 8     c1.setNext(c2);
 9
10     String str = c1.next.str;
11
12     PrintWriter writer = resp.getWriter();
13    writer.println(str);
14  }
```

The category *Factories* includes three test cases, of which two were successfully addressed by JSVFA-v0.3.0, while one wasn't handled successfully. Listing 4.5 illustrates a failing test related to a factory problem involving a string wrapper, which expects one leak. However, JSVFA-v0.3.0 reveals two leakages, resulting in a false positive.

Listing 4.5 exhibits the factory problem using a string wrapper. At `line:4`, *w1* wraps the tainted string from the user (*s1*), and similarly at `line:4`, *w2* wraps a constant. Then, both wrappers print their value content. At `line:9`, the output includes the tainted value (leak), whereas at `line:10`, the output is a hardcoded value, making it safe (no leak).

Listing 4.5: Factories3.java

```
1   protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
2       String s1 = req.getParameter("name");
3
4       StringWrapper w1 = new StringWrapper(s1);
5       StringWrapper w2 = new StringWrapper("abc");
6
7       PrintWriter writer = resp.getWriter();
8
9       writer.println(w1.toString());
10      writer.println(w2.toString());
11  }
```

The category *Inter* includes 14 test cases; JSVFA-v0.3.0 successfully manages seven, while the remaining seven are not managed. Listing 4.5 presents a test concerning object sensitivity that expects two leakages; however, the analyzer identifies only one, resulting in a false negative scenario.

Listing 4.5 illustrates a scenario involving calls to two distinct methods. A literal value (`line:4`) and a tainted value (`line:5`) are passed as parameters to a method that performs a standard string manipulation. Both values are subsequently printed at `line:8` and `line:9`, respectively. Subsequently, a similar process is employed utilizing another method that achieves similar results. In this second part, JSVFA-v0.3.0 fails to recognize that one of the printed values could lead to a leakage.

Listing 4.6: Inter9.java

```
1   protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
2       String s1 = req.getParameter("name");
3
4      String s2 = foo(s1);
5      String s3 = foo("abc");
6
7      PrintWriter writer = resp.getWriter();
8      writer.println(s2);
9
10     writer.println(s3);
11
12     String s4 = bar(s1);
13     String s5 = bar("abc");
14
15     writer.println(s4);
16     writer.println(s5);
17 }
18
19 private String foo(String s1) {
20    return s1.toLowerCase();
21 }
22
23 private String bar(String s1) {
24     return s1.toLowerCase(Locale.ENGLISH);
25 }
```

The category *Session* includes three test cases, all of which fail, showing that JSVFA-v0.3.0 is unable to handle any situation in this category. Listing 4.7 illustrates a simple session test case, which expects one leak. However, JSVFA-v0.3.0 does not report any, resulting in a false negative.

Listing 4.7 presents a case where an input is read from the request (source) at `line:2` to then be saved as a session attribute at `line:4`. After that, at `line:5`, the input is retrieved from the session and printed at `line:8` (sink), resulting in a leakage.

Listing 4.7: Session1.java

```
1 protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws IOException {
2    String name = req.getParameter("name");
3    HttpSession session = req.getSession();
4    session.setAttribute("name", name);
5    String s2 = (String) session.getAttribute("name");
6
7    PrintWriter writer = resp.getWriter();
```

```
    writer.println(s2);
}
```

The category *Strong Updates* includes five test cases, of which four are successfully addressed by JSVFA-v0.3.0, while one is not handled. Listing 4.8 illustrates the failing test, which is considered tricky because we can't assume a strong update with multiple variables that are not thread-local. JSVFA-v0.3.0 is unable to detect the expected leakage, resulting in a false negative.

Listing 4.8 shows a case when the sink is labeled as a leakage even though the runtime behavior is safe. At `line:4`, the field *this.name* becomes tainted (source), but then at `line:5`, the field is overwritten with a constant literal (in real execution, this operation erases all previous values). Finally, at `line:8`, the value of the field *this.name* is printed (sink?). It is considered a leakage because object fields may have aliases, so it is expected to report one vulnerability, even though the code is safe at runtime.

Listing 4.8: StrongUpdates4.java

```
1 private String name;
2
3 protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws IOException {
4     String name = req.getParameter("name");
5     name = "abc";
6
7     PrintWriter writer = resp.getWriter();
8     writer.println(name);
9 }
```

The category *Pred* comprises nine test cases, with six successfully addressed by JSVFA-v0.3.0, while the remaining three are not managed. Listing 4.9 demonstrates a simple correlated test where the two branches are mutually exclusive, and these conditions cannot both be true, thus preventing any leakage. JSVFA-v0.3.0 identifies a single leakage, leading to a false positive result.

Listing 4.9 shows a case where the variable *name* starts as a safe constant at `line:3`, and the variable *name* randomly becomes true or false at `line:2`. If `choice == true`, variable *name* is overwritten with untrusted input at `line:6`, but if `choice == false`, *name* remains untouched. Finally, the sink executes only when `choice == false` at `line:9`, which means that there is no leakage because the sink only runs in execution paths where *name* still contains the constant.

Listing 4.9: Pred3.java

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    boolean choice = new Random().nextBoolean();
    String name = "abc";

    if(choice) {
        name = req.getParameter("name");
    }

    if(! choice) {
        PrintWriter writer = resp.getWriter();
        writer.println(name); // nothing bad gets here
    }
}
```

The category *Reflection* encompasses four test cases, all of which fail, indicating that JSVFA-v0.3.0 is incapable of addressing any scenario within this category. Listing 4.10 demonstrates a reflective invocation of a method that expects a single leak. However, JSVFA-v0.3.0 does not yield any results, leading to a false negative.

Listing 4.10: Refl1.java

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    String s1 = req.getParameter("name");
    PrintWriter writer = resp.getWriter();

    Method idMethod = null;
    try {
        Class clazz = Class.forName("securibench.micro.reflection.Refl1");
        Method methods[] = clazz.getMethods();
        for(int i = 0; i < methods.length; i++) {
            Method method = methods[i];
            if(method.getName().equals("id")) {
                idMethod = method;
                break;
            }
        }
        // a fancy way to call id(s1, writer)
        Object o = idMethod.invoke(this, new Object[] {s1, writer});
        String s2 = (String) o;
        writer.println(s2);
    } catch( ClassNotFoundException e ) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

public String id(String string, PrintWriter writer) {
    return string;
}
}
```

The category *Sanitizers* includes six test cases, none of which were successfully addressed by JSVFA-v0.3.0. When the analyzer is executed, an exception is thrown, avoiding the computation of its metrics. Listing 4.9 illustrates a basic sanitization examination that employs an intermediary method to eliminate the taint from a value (`line:3`). It anticipates one leakage; however, JSVFA-v0.3.0 fails to identify any, resulting in a false negative outcome.

Listing 4.11: Sanitizers1.java

```java
1  protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
2      String s1 = req.getParameter("name");
3      String clean = clean(name);
4  
5      writer = resp.getWriter();
6      resp.setContentType("text/html");
7  
8      writer.println("<html>");
9      writer.println(’<b>’ + name + ’</b>’); /* BAD */
10     writer.println("<b>" + clean + "</b>");
11     writer.println("</html>");
12 
13 }
```

In addition, a general issue about *strings*, specifically the class `StringBuilder`, which is designed for efficient string manipulation such as appending, deleting, or replacing strings or characters. JSVFA-v0.3.0 is able to compute that there is a path between the source and sink in a straightforward scenario, such as that depicted in Figure 4.3, which performs an `append` operation at `line:4`.

```
1 function main(String args[]) {
2     String s = source();
3     StringBuilder buffer = new StringBuilder();
4     buffer.append(s);
5     sink(buffer.toString());
6 }
```

Figure 4.3: Pass Case: StringBuilder

In contrast, Figure 4.4 illustrates a scenario in which JSVFA-v0.3.0 fails to detect an explicit leak between source and sink. This issue occurs due to the invocation of the `StringBuilder` operation `append`; it is called three times in a nested manner at `line:5`. JSVFA-v0.3.0 is unable to process this kind of sequence call when an `append` is contained within another.

```
1 function main(String args[]) {
2     String s = source();
3     String name = "pepito";
4     StringBuilder buffer = new StringBuilder();
5     buffer.append('pepe').append(name).append(s);
6     sink(buffer.toString());
7 }
```

Figure 4.4: Fail Case: StringBuilder

4.2 Study II

This study aims to address **RQ3** by evaluating the completeness of the enhanced framework JSVFA-v0.6.0 using the SecuriBench benchmark. It gathers metrics, including *precision*, *recall*, *f-score*, and *pass rate*, to identify potential issues. We once again compare the results with the static analyzers FlowDroid and Joana to evaluate the positioning of the JSVFA framework in relation to them based on their metrics. Similar to the initial study (Section 4.1).

The first part of this study compares the metrics between the initial and the newer version of JSVFA framework (Table 4.1). JSVFA-v0.6.0 reveals that out of 122 executed test cases, it detects the expected leaks in 75 of them, yielding a pass rate of 61.48%. This result indicates an improvement from the previous pass rate of 51.64% for JSVFA-v0.3.0. The new JSVFA-v0.6.0 achieved 87% precision, representing a 1% decrease compared to its predecessor JSVFA-v0.3.0, indicating a tiny decline in its ability to classify results as positive, which could be caused by some collateral effects introduced by the new changes. However, JSVFA-v0.6.0 outperforms JSVFA-v0.3.0 in recall, achieving 73% compared to its previous 62%, by detecting fewer false negatives. Therefore, the trade-off between the last two metrics, based on f-score, shows that JSVFA-v0.6.0 outperforms JSVFA-v0.3.0 with a score of 79% compared to 72%. In conclusion, JSVFA-v0.6.0 has shown enhancements in *recall*, *f-score*, and *pass rate* metrics; however, there was a slight decline in *precision*.

Frameworks	**Precision**	**Recall**	**F-score**	**Pass Rate**
JSVFA-v0.3.0	0.88	0.62	0.72	51.64%
JSVFA-v0.6.0	0.87	0.73	0.79	61.48%

Table 4.1: Metrics: JSVFA-v0.3.0 vs JSVFA-v0.6.0

In terms of pass rate, JSVFA-v0.6.0 shows a substantial improvement from 51.64% to 61.48%. Figure 4.5 presents a comparison of the pass rate metric for each test category against its previous versions, indicating that some metrics have been improved or at least maintained; however, just one of them has experienced a decline in its value. This new version demonstrates improved results in the test categories *Array*, *Basic*, *Collection*, *Inter*, and *Sanitizers*, particularly highlighting an increase in the pass rate metric in the *Array* category from 10% to 50%. Furthermore, the categories *Aliasing*, *DataStructure*, *Factory*, *Session*, and *Sanitizers* exhibit no variation in their metric values, because we failed to address issues pertaining to them. In addition, the test category *Strong update* exhibits a lower pass rate compared to its predecessor, attributed to collateral effects resulting from recent enhancements.

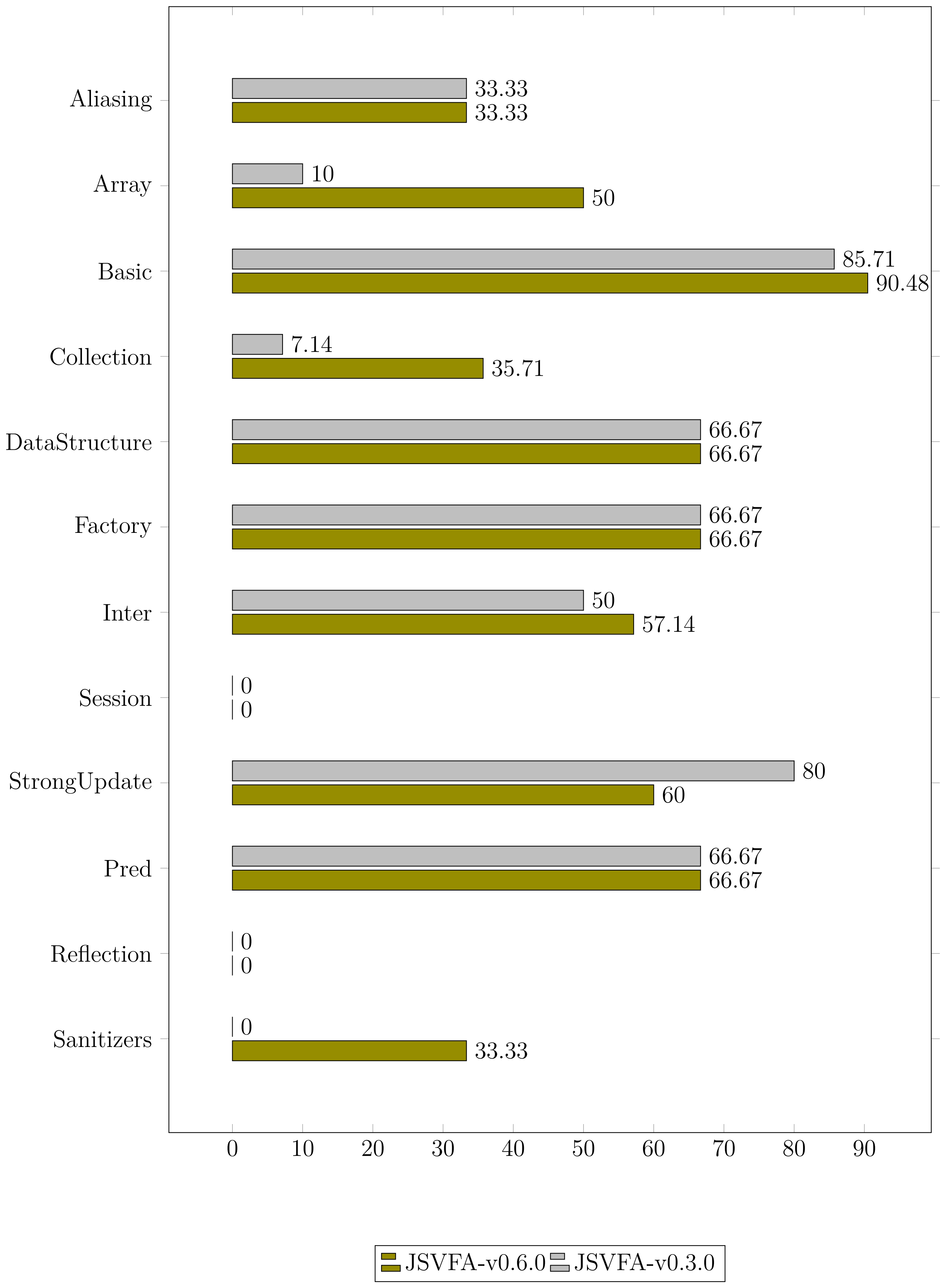


Figure 4.5: Test categories: JSVFA-v0.3.0 vs JSVFA-v0.6.0

The category *Aliasing* maintains a pass rate of 33.33%. We examine the case of *Aliasing5* 4.1, which demonstrates a vulnerability caused by aliasing. This situation arises when two variables, provided as parameters to a method, point to the same mutable object, meaning that a modification made through one alias affects the other. This test example displays a leakage that the new version (JSVFA-v0.6.0) fails to identify due to *the actual implementation's inability to manage interprocedural argument aliasing*. Such behavior makes analysis harder because the vulnerability arises through alias-induced side effects.

The category *Arrays* has improved its pass rate from 10% to 50%. We have introduced a significant enhancement in the methods that handle array logic; this improvement has addressed issues in four additional tests (`ArrayTest1`, `ArrayTest3`, `ArrayTest4`, and `ArrayTest6`), which have passed successfully with the most recent modifications. We performed changes in two array-related methods: (1) the `loadArrayRule` and (2) the `storeArrayRule`.

The method `loadArrayRule` facilitates array load operations by incorporating edges into the value-flow graph. This evaluates expressions of the form $p = q[i]$, indicating the assignment of the array index value $q[i]$ (on the right) to the local variable p (on the left). Listing 4.12 presents the newly added code lines to this method, which addresses special cases, specifically when the right-hand side of the array index references other local variables, resulting in edges from those variables to the source statement. To handle this behavior, we utilize the global variable *arrayStores*, which is populated in the `storeArrayRule` method.

Listing 4.12: Method loadArrayRule

```scala
def loadArrayRule(
     targetStmt: soot.Unit,
     ref: ArrayRef,
     method: SootMethod,
     defs: SimpleLocalDefs
  ): Unit = {
  ...
    val stmt = Statement.convert(sourceStmt)
    stmt match {
        case AssignStmt(base) => {
            val rightOp = AssignStmt(base).stmt.getRightOp
            if (rightOp.isInstanceOf[Local]) {
                arrayStores
                  .getOrElseUpdate(rightOp.asInstanceOf[Local], List())
                  .foreach(storeStmt => {
                    val source = createNode(method, storeStmt)
                    val target = createNode(method, sourceStmt)
                    updateGraph(source, target)
                })
            }
        }
        case _ =>
    }
  ...
}
```

In contrast, the method `storeArrayRule` controls the storage of values in arrays and incorporates edges into the value-flow graph when values are assigned to array indexes. It additionally maintains a global mapping called `arrayStores`, which records all instances of modifications to each array variable for further analysis. This assesses expressions structured as $p[i] = q$, indicating the assignment of the local variable q (to the right) to the array index location $p[i]$ (to the left). Listing 4.13 illustrates the latest incorporated code lines in this method, which instances connections from all definitions of that variable (right-hand side) to the current statement.

Listing 4.13: Method storeArrayRule

```
def storeArrayRule(
    assignStmt: AssignStmt,
    method: SootMethod,
    defs: SimpleLocalDefs
): Unit = {
...
  if (right.isInstanceOf[Local]) {
      val rightLocal = right.asInstanceOf[Local]
      defs
          .getDefsOfAt(rightLocal, assignStmt.stmt)
          .forEach(sourceStmt => {
              val source = createNode(method, sourceStmt)
              val target = createNode(method, assignStmt.stmt)
              svg.addEdge(
                  source,
                  target
              )
          })
    }
  ...
}
```

The category *Basic* has achieved a pass rate of 90.48%. The primary issue regarding context has been addressed; however, the updated version of JSVFA framework continues to underreport the anticipated number of leaks from four tests. This issue persists due to specific edge cases related to context, including nested method calls and complex loop statements like "for" or "while". These factors impact not only this category but also others, such as *collections*, *factories*, and *inter*.

The category *Collections* has achieved a pass rate of 35.71%. While certain issues related to repetitive structure have been resolved, as demonstrated in Listing 4.3 with the updated version (JSVFA-v0.6.0), there remain nine tests that continue to fail. These tests involve complex scenarios that integrate loop statements and context cases. These tests involve edge cases that we have not yet fixed.

The category *Datastructures* has consistently maintained a pass rate of 66.67%. We analyze *Datastructures4* (Listing 4.4), which illustrates a vulnerability arising from nested data. It occurs when two objects *(c1, c2)* are linked in a list, followed by the retrieval of the field from the second object, as shown in the statement `String str = c1.next.str`. This test example does not exhibit any leakage; however, the JSVFA framework identifies one false positive. This behavior arises from the inability of the underlying logic to manage context in nested scenarios. A potential solution to this issue involves modifying and adding additional constraints to the logic responsible for generating the *Open/Close* labels in the file `br/unb/cic/soot/svfa/jimple/JSVFA.scala`.

The category *Factories* has maintained a pass rate of 66.67%. We examined the failing test *Factories3* (Listing 4.5), which demonstrates a factory issue involving a value wrapped in a string object, as shown in the statement `StringWrapper w1 = new StringWrapper(s1)`. This case aims to identify a single leakage; however, the updated version of `jsvfa` fails to do so, detecting two leakages instead, one of which is a false positive. This behavior results from a context issue that the existing logic fails to address, particularly concerning the manner in which parameters are passed to the constructor.

The category *Inter* has experienced a slight increase in its pass rate to 57.14%; however, there remain instances of failure, as illustrated in Listing 4.6. Similar to other categories, complex scenarios related to context persist, and JSVFA-v0.6.0 has yet to address these challenges. These shortcomings will result in the reporting of false negatives in areas where leakage is expected.

The category *session* has exhibited no advancement compared to its predecessor. None of the three test cases have passed successfully. We analyzed the failing test *Session1*

(Listing 4.7), which illustrates a straightforward session scenario in which unsafe data is transmitted through an indirect container (the session). The expectation was for one leakage; however, the new version of JSVFA framework continues to fail in detection, resulting in a false negative. This behavior may be attributed to the manner in which the SecuriBench benchmark simulates the session methods within the *HttpServletRequest* context.

The category *Strong Updates* indicates a decline in the pass rate metric. The initial report utilizing JSVFA-v0.3.0 recorded one failure out of five tests (80%); however, the execution of JSVFA-v0.6.0 resulted in two failures (60%), reflecting a reduction in the pass rate percentage. We analyze one of the failing tests, `StrongUpdates5` (Listing 4.14), which reads a tainted parameter (source) and subsequently overwrites it with the string "abc" prior to printing (sink). This test is designed as a safe case anticipated to yield no leaks, thus presenting no XSS risk; however, JSVFA-v0.6.0 indicates one false positive. This workflow appears to be a straightforward case that should be addressed by the JSVFA framework; however, the issue may stem from the use of `synchronized`, which locks on the object referenced by `this.name`.

The analysis result in both tests does not align with the expected number of leaks, potentially due to limitations or bugs in the JSVFA framework implementation, including incomplete alias analysis, imprecise value flow tracking, or inadequate propagation of taint through specific program constructs.

Listing 4.14: StrongUpdates5

```java
protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    synchronized (this.name) {
        name = req.getParameter(FIELD_NAME);
        name = "abc";

        PrintWriter writer = resp.getWriter();
        writer.println(name);
    }
}
```

The category *Pred* maintained a pass rate of 66.67%. We analyzed the failing test *Pred3* (Listing 4.9), which illustrates a scenario where conditions are mutually exclusive, thereby ensuring that no untrusted data reaches the leakage. The updated version of `jsvfa` does not achieve the intended outcome and generates a false negative, as the existing logic consistently evaluates both paths of a conditional, regardless of the condition being True or False.

The category *Reflection* has shown no progress relative to its predecessor. None of the three or four cases have been successfully completed due to the fact that we have not worked on them. This decision was made as the reflection feature is not within the scope of this research; however, it will be considered in future work.

The category *Sanitizers* has attained a pass rate of 33.33%, an improvement from zero in the prior version. Nonetheless, the updated version of JSVFA-v0.6.0 continues to yield false negatives in the tests that are still failing, as we have not fully implemented the logic required to handle the intermediary method employed by the sanitizer.

We categorized the 47 remaining failing tests into eight groups, as detailed in Table 4.2. Future research will focus on addressing these issues.

Array Indexes (i): The actual implementation is unable to recognize tainted in specific indexes from an array. Currently, it marks all the array as tainted. We have mapped six cases *(12.77%)*.

Support Class Missing (ii): Some tests use methods from securibench that are not mocked. We have mapped seven cases *(14.89%)*.

Missing Context (iii): The logic for handling context is not entirely flawless, resulting in certain edge cases that lead to bugs such as (a) nested structures as HashMap, LinkedList, and others; (b) complex loop statements as "for" or "while"; and (c) parameters passed in the constructor. We have mapped 16 cases. *(34.04%)*.

Reflection (iv): The current implementation does not address the reflection feature. We have mapped five cases *(10.64%)*.

Global variables references (v): There are unaddressed edge cases regarding the handling of the definition of global variables. We have mapped two cases *(4.26%)*.

Path for conditional (vi): The current logic always evaluates two paths for a conditional, regardless of whether the condition is set to True or False. We have mapped three cases *(6.38%)*.

Sanitizer method (vii): The current implementation fails to deal with the intermediary method utilized by the sanitizer. We have mapped three cases *(6.38%)*.

Others (viii): We have mapped five case: *(10.63%)*.

The second part of this study performs a comparison with the state-of-the-art tools. Figure 4.6 shows that JSVFA-v0.6.0 has a precision level that is 3% lower than FlowDroid, which is 90%. This means that JSVFA-v0.6.0 has a slightly higher rate of false positives. However, JSVFA-v0.6.0 exhibits a better ability to address fewer false negatives, yielding a recall of 73%; conversely, FlowDroid misidentified 4 leakages, leading to a recall rate of 68%. Concerning the F-score, JSVFA-v0.6.0 surpasses FlowDroid by 2% at least. The pass rate metric reveals that FlowDroid executions exhibited a superior value, achieving a 65.05% pass rate for test cases, in contrast to 61.48% for JSVFA-v0.6.0. In summary,

Issue	Tests
i	Aliasing3, Arrays2, Arrays5, Arrays8, Arrays9, and Arrays10
ii	Basic31, Basic36, Basic38, Session1, Session2, Session3, and Sanitizers5
iii	Aliasing5, Basic42, Collections3, and Collections5, Collections6, Collections7, Collections8, Collections9, Collections10, Collections12, Collections13, Datastructures4, Datastructures5, Factories3, Inter9, and Inter12
iv	Inter6, Refl1, Refl2, Refl3, and Refl4
v	StrongUpdates, and StrongUpdates5
vi	Pred3, Pred6, and Pred7
vii	Sanitizers2, Sanitizers4, and Sanitizers6
viii	Aliasing2, Aliasing4, Inter4, Inter5, and Inter11

Table 4.2: Common Issues

although the precision from the new version of JSVFA framework has decreased, the recall has increased, so the trade-off represented by the f-score has improved, outperforming FlowDroid. JSVFA-v0.6.0 has not yet surpassed FlowDroid regarding the pass rate metric; however, it has reduced the margin percentage from 18%, in the initial version, to 5% currently.

In comparison to Joana, JSVFA-v0.6.0 exhibits superior precision, achieving 87% versus 82%, respectively, suggesting an increased probability of accurately classifying a result as positive. Regarding recall, JSVFA-v0.6.0 surpasses Joana, attaining 73% in contrast to 68%, by recognizing fewer false negatives. Consequently, JSVFA-v0.6.0 outperforms Joana with respect to the f-score metric. However, Joana surpasses JSVFA-v0.6.0, achieving a 69.67% pass rate in tests completed, while JSVFA-v0.6.0 attains only 61.48%. In conclusion, the latest iteration of JSVFA framework demonstrates superior metrics (*precision*, *recall*, and *f-score*) compared to Joana. Although JSVFA-v0.6.0 does not overcome Joana in terms of the pass rate metric, it has decreased the percentage margin between them by over 5%.

4.3 Study III

This study addresses the **RQ4** and the Android application domain by assessing the latest version of the framework JSVFA-v0.6.0 through a benchmark referred to as TaintBench, which consists of a compilation of 38 legacy malware Android Apks [12]. This evaluation aims to collect metrics to improve the understanding of JSVFA-v0.6.0 completeness in this novel domain. We have organized it into two experiments with different approaches,

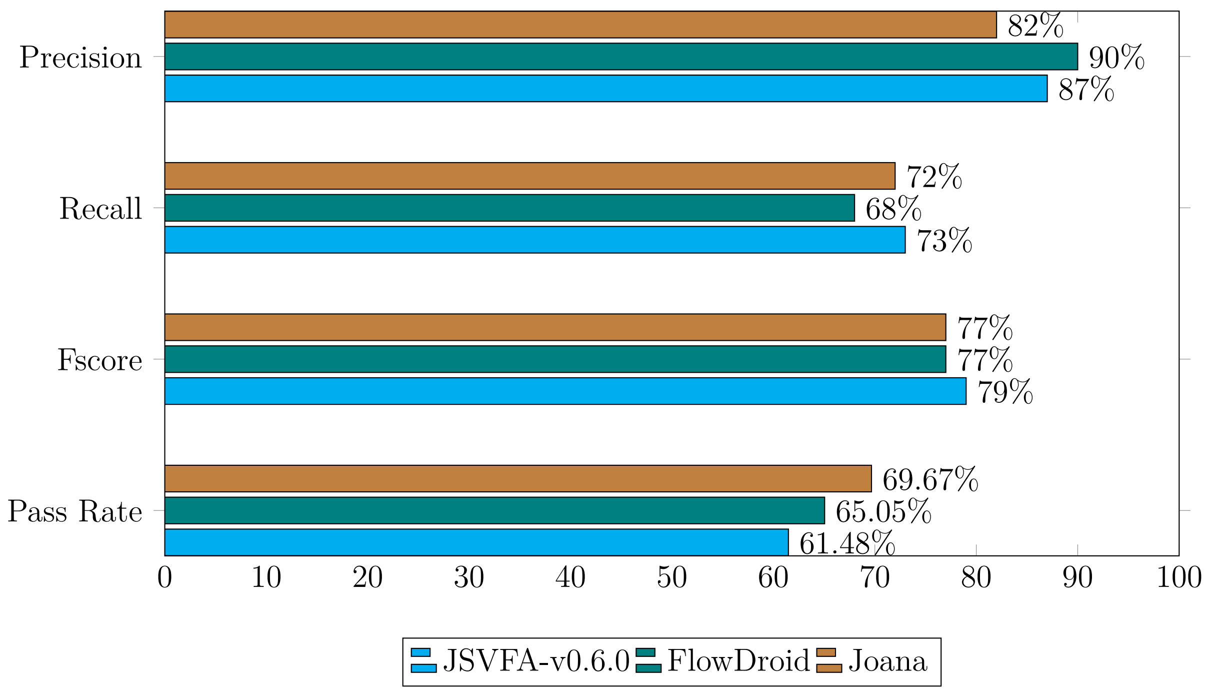


Figure 4.6: JSVFA-v0.6.0 comparison against FlowDroid and Joana

and both compare the matches between the set of leaks reported by each APK execution in the paper [12], referred to as "expected," against the number of leaks identified by JSVFA framework, which is termed "actual." A "match" is when the source and sink are the same in both the actual and expected sets. The comparison is conducted for each APK from TaintBench, with the results displayed in a tabular format (Table 4.3 4.4).

The first experiment replicates `Experiment 2 - TB2` from paper [12], which states, *All tools are configured with sources and sinks defined in the benchmark suite.* This set of sources and sinks is located in the repository at `https://github.com/TaintBench/TaintBench/blob/main/TB_SourcesAndSinks.txt`. The findings of the present experiment are displayed in Table 4.3. Out of 39 APKs executed by JSVFA framework, no leaks were detected in 21 of them. Among the remaining 18, leaks have been detected; however, they hardly correspond to those identified in the experiment presented in the TaintBench paper.

We identified 10 pairing cases in which the sources and sinks are identical in both instances. Furthermore, we examined the additional leaks that are not presented in the TaintBench paper and determined that they are also valid leaks. This example examines the APK named *roidsec*, in which JSVFA framework identified three leaks. We have selected one leak, with its source being *<java.io.File: java.io.File[] listFiles()>* and its sink *<java.io.File: boolean delete()>*, from the list of potential leaks. Listing 4.15 illus-

APK	**actual**	**expected**	**matches**
jollyserv	1	1	0
phospy	1	4	1
cajino_baidu	8	14	1
fakedaum	3	2	0
roidsec	3	4	0
repane	1	1	0
backflash	1	12	0
smssilience_fake_vertu	1	4	1
proxy_samp	13	19	3
sms_send_locker_qqmagic	2	4	2
sms_google	1	4	0
save_me	9	26	0
threatjapan_uracto	1	2	0
chat_hook	8	11	2
death_ring_materialflow	18	1	0
hummingbad_android_samp	10	2	0
dsencrypt_samp	3	1	0
scipiex	2	3	0
faketaobao	0	2	0
samsapo	0	5	0
fakeappstore	0	3	0
godwon_samp	0	5	0
overlay_android_samp	0	6	0
exprespam	0	2	0
overlaylocker2_android_samp	0	19	0
fakemart	0	1	0
smssend_packageInstaller	0	5	0
tetus	0	2	0
smsstealer_kysn_assassincreed_android_samp	0	4	0
remote_control_smack	0	17	0
the_interview_movieshow	0	1	0
fakeplay	0	2	0
beita_com_beita_contact	0	3	0
xbot_android_samp	0	3	0
stels_flashplayer_android_update	0	3	0
vibleaker_android_samp	0	4	0
slocker_android_samp	0	5	0
chulia	0	4	0
fakebank_android_samp	0	5	0
`Total`	**86**	**216**	**10**

Table 4.3: TaintBench- Experiment I

trates that this finding constitutes a true positive leak, as there exists a potential path in method `onDestroy()` from class `PhoneSyncService` when the dataflow enters the nested structure (*if*) followed by the loop (*for*).

The absence of this leak from the findings list for this APK, which serves as the project's ground truth, was unexpected. Initially, we considered categorizing this taint flow as a false positive. However, we opted to consult Dr. Linghui Luo, a primary developer of the TaintBench project, to address this issue. She responded via email, stating that *"the findings in TaintBench are baseline findings"*, indicating that the TaintBench findings report may not encompass all taint flows.

Finally, `experiment-i` revealed that JSVFA-v0.6.0 was unable to identify leaks in over half of the analyzed APKs from TaintBench (53.85%). This failure may be due to the reliance on a unique set of sources and sinks, as well as the absence of specific data flow constraints. Conversely, JSVFA-v0.6.0 has identified undocumented leaks in the TaintBench paper, uncovering 76 new flows that converge at these leaks, with 10 corresponding to the ground truth reports.

Listing 4.15: New leak found in Roidsec

```java
public void onDestroy() {
    File[] files = getCacheDir().listFiles();
    if (files) {
        for (File f : files) {
            f.delete();
        }
    }
    releaseWakeLock();
    Intent localIntent = new Intent();
    localIntent.setClass(this, PhoneSyncService.class);
    startService(localIntent);
    Log.i(TAG, "service destroy");
    super.onDestroy();
}
```

In addition, we performed another experiment, which emulates `Experiment 3 - TB3` from paper [12]. It follows the statement that *For each benchmark app, a list of sources and sinks defined in this app is used to configure all tools. Each tool analyzes each benchmark app with the associated list of sources and sinks.* The individual list of sources and sinks can be found in each application repository at `https://taintbench.github.io/taintbenchSuite`. The results of the current experiment are presented in Table 4.4. Of the 39 APKs executed by JSVFA framework, 9 exhibited no leaks. Among the remaining

30, leaks have been detected; however, they do not significantly correspond to those found during the experiment described in the TaintBench paper.

To put it concisely, `experiment-ii` indicated that employing a more targeted selection of sources and sinks enhances JSVFA-v0.6.0's likelihood of detecting leaks. The current failure rate for detecting any leaks in the target APKs is 23.07%. This issue may stem from the non-implementation of specific dataflow rules in the source code, particularly because **jsvfa does not accurately model calls to native methods in the Android source code.** Furthermore, JSVFA-v0.6.0 has identified 645 new leaks, suggesting a considerable number of undocumented flows in the TaintBench paper; however, only 10 of these have been confirmed to be aligned with the ground truth reports, indicating the same value observed in `experiment-i`.

jollyserv	9	1	0
phospy	3	4	1
cajino_baidu	167	14	0
fakedaum	15	2	0
roidsec	10	4	0
repane	1	1	0
backflash	17	12	0
overlay_android_samp	8	6	0
smssilience_fake_vertu	5	4	1
proxy_samp	60	19	3
overlaylocker2_android_samp	35	19	0
sms_send_locker_qqmagic	12	4	2
sms_google	12	4	0
smssend_packageInstaller	48	5	0
tetus	27	2	0
smsstealer_kysn_assassincreed_android_samp	1	4	0
remote_control_smack	7	17	0
the_interview_movieshow	1	1	0
fakeplay	15	2	0
save_me	32	26	0
beita_com_beita_contact	14	3	0
stels_flashplayer_android_update	28	3	0
vibleaker_android_samp	9	4	0
threatjapan_uracto	9	2	0
chat_hook	19	11	2
death_ring_materialflow	40	1	1
fakebank_android_samp	8	5	0
hummingbad_android_samp	31	2	0
dsencrypt_samp	2	1	0
scipiex	10	3	0
faketaobao	0	2	0
samsapo	0	5	0
fakeappstore	0	3	0
godwon_samp	0	5	0
exprespam	0	2	0
fakemart	0	1	0
xbot_android_samp	0	3	0
slocker_android_samp	0	5	0
chulia	0	4	0
TOTAL	**655**	**216**	**10**

Table 4.4: TaintBench- Experiment II

Chapter 5

Final Remarks

In this final chapter, we will discuss the findings of this research.

5.1 Answer the research questions

We have run the initial version of JSVFA framework, referred to as JSVFA-v0.3.0, using SecuriBench. It successfully addressed 63 out of 122 tests, resulting in a precision of 88%, a recall of 62%, an f-score of 72%, and a pass rate of 51.64%. Compared to FlowDroid metrics, the latter demonstrates superior values in all metrics. In the Joana comparison, JSVFA framework only demonstrates superior precision, yet exhibits lower values in recall, f-score and pass rate.

RQ1: The trade-off between precision and recall reveals that JSVFA-v0.3.0 achieves an f-score that is inferior to the one for the two known analyzers. In addition, in terms of the pass rate metric, JSVFA-v0.3.0 displays the lowest value in comparison to the other alternatives.

Furthermore, we analyzed the prior results and utilized the test categories into which SecuriBench is divided as a foundation for a more in-depth analysis. This examination revealed lower metrics in certain categories, including *Array*, *Collection*, *Session*, *Reflection*, and *Sanitizers*. Limitations were identified in the core logic, particularly regarding the analysis's inability to effectively manage array manipulation, the distinctiveness of context in method calls, and the handling of native string methods such as *append*.

RQ2: Certain scenarios exist where the core logic of the analysis fails to correspond with what is expected in terms of leaks. This misalignment arises from limitations such as incomplete implementation of constraints in the analysis rules, imprecise tracking of value flow, and insufficient propagation of taint through the graph construction.

Following the implementation of fixes in the source code and the release of a new version of JSVFA framework, designated JSVFA-v0.6.0, we recalculated the metric using the SecuriBench benchmark. The results indicated that 75 out of 122 tests successfully passed, translating to 87% precision, 73% recall, 79% f-score, and a pass rate of 61.48%. In comparison to the other tools, JSVFA framework has shown improvements in terms of recall and f-score metrics; however, it ranks in the middle for precision, with FlowDroid leading in this metric. Conversely, JSVFA framework exhibits the lowest pass rate value.

RQ3: The most recent version of JSVFA framework exhibits improvements across the majority of its metrics. Nonetheless, the recent modifications in the source code have resulted in collateral effects, leading to a slight decrease in the precision of this new version.

On the other hand, we conducted an experimental study to investigate the Android ecosystem, utilizing the benchmark known as TaintBench. In this study, we have modified the approach to computing metrics, as in the previous ones. This experiment quantifies the number of analogous leak paths between JSVFA framework results and those documented in the ground truth. Following the experiments, we identified a few matching cases; however, we also uncovered a considerable number of new valid leaks that were not recorded in the official benchmark reports.

RQ4: The results suggest that these findings offer promise for employing the JSVFA framework in the detection of data leaks within APKs. Further exploration of the Android context is essential to identify, define, and implement new mobile data flow rules.

5.2 Future Work

Future work will focus on further enhancements of JSVFA framework. We have mapped a list of possible issues, not all of which have been resolved, and they may be the cause that certain SecuriBench test categories still exhibit low pass rate metrics, including *Session*, *Strong Updated*, and *Reflection*. Additionally, the categories *Aliasing*, *Data Structure*, *Factory*, and *Pred* report no changes between the initial and latest versions of JSVFA framework, indicating potential for improvement.

We aim to investigate the Android field; the experimental study conducted has provided intriguing insights regarding this area. The Android lifecycle differs from Java due to the structure of mobile applications, which comprise one or more screens or activities, and each activity undergoes multiple stages, which are organized by activity stacks. JSVFA framework does not handle the mentioned logic, so it requires further research to delineate the new dataflow rules.

A future direction involves upgrading the base framework Soot, which JSVFA framework relies on, to its new and improved version termed "sootUp." It represents a comprehensive re-implementation and redesign of the framework, intended to resolve the limitations and complexities encountered in Soot. This updated version presents a more modular, maintainable, and extensible architecture [26]. We must conduct multiple concept proofs to ascertain whether all specific features required by JSVFA framework are comprehensively implemented in SootUp, including certain analyses, such as point-to.

Furthermore, we intend to integrate additional benchmarks that include test cases for Java and Android. This approach will mitigate bias by utilizing only the actual benchmark (SecuriBench and TaintBench). Furthermore, these new benchmarks can help identify additional edge cases that the current ones might not detect.

5.3 Contributions

1. Identification of the main limitations of JSVFA framework,

   The most important contribution is identifying the significant limitations of JSVFA framework, based on the empirical studies that we conducted. Consequently, this contribution identifies and discusses the problems within the framework and provides viable solutions, making it feasible to perform further research in this domain. A list of potential issues impacting the correct identification of leakages has been documented.

2. An up-to-date implementation of SVFA for Java and Android,

   Another principal contribution is the release of a novel version of JSVFA framework, which brings new enhancements including bug fixes, project modularization, pipeline construction, and the implementation of a new workflow to handle the analysis of APKs.

3. A reliable repository incorporating the SecuriBench benchmark within state-of-the-art tools.

   The source code for the integration of the SecuriBench benchmark with FlowDroid and Joana tools represents another significant contribution. Due to the absence of information regarding the reproduction of studies assessing the completeness of the specified tools, we opted to implement and host them in a GitHub repository. This repository includes a comprehensive "readme" file that provides detailed, step-by-step instructions for reproducing the computations.

5.4 Reproducibility and Code Availability

Reproducibility constitutes a core principle in scientific research. To facilitate validation and extend our contributions, we provide the source code from the updated version of the JSVFA framework. This project is licensed under the MIT License and is permitted for non-commercial, academic purposes, specifically for researchers seeking to replicate or extend the findings of this study.

We meticulously document and incorporate detailed accounts of the benchmark's execution, including version details and reproducibility instructions, within the source code. Researchers seeking to access the implementation are invited to request access to the repository from PAM (Program Analysis and Manipulation Group at University of Brasília) at `https://github.com/PAMunb`. We are dedicated to promoting transparency and collaboration within the academic community and will strive to enhance reproducibility in accordance with research practices.

For the API usage, it is necessary to implement a class that extends the `JSVFA` class and provides the required implementations for:

- `getEntryPoints()` - Set up the "main" methods (returns a list of Soot methods),
- `sootClassPath()` - Set up the soot classpath (returns a String),
- `analyze(unit)` - Identify node types (source, sink, simple node) in the graph.

Listing 5.1 presents the structure of the source code, which contains the required methods. An example of implementation is available in the code located at `JSVFATest`.

Listing 5.1: MyAnalysis

```
import br.unb.cic.soot.JSVFATest

class MyAnalysis extends JSVFATest {
  override def getEntryPoints(): List[SootMethod] = {
    // Define your entry points
  }

  override def sootClassPath(): String = {
    // Return your classpath
```

```scala
  }

  override def analyze(unit: Unit): NodeType = {
    // Analyze statements and return Source, Sink, or SimpleNode
  }
}
```

The framework employs a flexible approach as a trait for defining the set of entry points, including both source and sink. The method `analyze(unit)` uses the information from this trait.

Listing 5.2: SecuribenchSpec

```scala
trait SecuribenchSpec {
  val sinkList: Seq[String] = List()

  val sourceList: Seq[String] = List()
```

More detailed information is in the `readme.md` file from this project.

5.5 Conclusion

This study successfully achieved its objective of conducting an empirical analysis of JSVFA framework through source code examination, enhancements, metric computation, and comparative evaluation with other tools. The exploration of the core algorithm of the analyzer has revealed the origins of certain issues, resulting in the resolution of these limitations and the release of a new version of JSVFA framework, which has shown a significant improvement over its predecessor. The updated version surpasses the initial iteration across nearly all metrics: *recall*, *f-score*, and *pass rate*; however, this improvement entails a minor decrease in *precision*. The comparison of the new version of the framework with FlowDroid and Joana reveals a trade-off among *precision*, *recall*, and *f-score*, positioning JSVFA framework in a median position relative to both state-of-the-art tools. The findings indicate that our tool may serve as a viable alternative in the academic domain, subject to the specific approach and selected metric.

Furthermore, two experimental studies were conducted to evaluate the completeness of JSVFA framework within the Android environment utilizing the TaintBench benchmark. The experiments revealed two notable findings: first, JSVFA framework successfully identified a significant number of new leaks (76 and 645, respectively, for each experiment) that were not included in the ground truth reports but subsequently were confirmed as “true positives” by the author of TaintBench. The second issue is that JSVFA framework significantly failed to identify the paths between sources and sinks outlined in the ground truth report. The results indicate that these new findings provide optimism for utilizing JSVFA framework to identify data leaks in APKs. However, further exploration of the Android lifecycle is necessary to identify, delineate, and implement potential new rules in mobile data flow.

References

[1] Lhoták, Ondrej: *Spark: A flexible points-to analysis framework for java.* 2003. x, 12, 13

[2] Møller, Anders and Michael I Schwartzbach: *Static program analysis.* Notes. Feb, 2012. 1, 10, 11

[3] Sui, Yulei and Jingling Xue: *Svf: interprocedural static value-flow analysis in llvm.* In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016. 1, 2, 15

[4] Arzt, Steven, 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. 2, 3, 13, 14, 22

[5] Qu, Binbin, Beihai Liang, Sheng Jiang, and Chutian Ye: *Design of automatic vulnerability detection system for web application program.* In *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pages 89–92. IEEE, 2013. 2

[6] Tripp, Omer, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman: *Taj: effective taint analysis of web applications.* ACM Sigplan Notices, 44(6):87–97, 2009. 2

[7] Arzt, Steven: *Soot - a framework for analyzing and transforming java and android applications*, 1997. `https://soot-oss.github.io/soot/`, visited on 2023-07-09. 2, 12

[8] rbonifacio: *Sparse value flow analysis implementation based on soot*, 2019. `https://github.com/rbonifacio/svfa-scala`, visited on 2023-07-09. 2

[9] Lausanne, Ecole Polytechnique Federale de: *Scala 2.x*, 2017. `https://www.scala-lang.org/download/2.13.11.html`, visited on 2023-07-09. 2

[10] Jesus, Galileu Santos de, Paulo Borba, Rodrigo Bonifácio, and Matheus Barbosa de Oliveira: *Detecting semantic conflicts using static analysis.* CoRR, abs/2310.04269, 2023. `https://doi.org/10.48550/arXiv.2310.04269`. 2

[11] Graf, Jürgen, Martin Hecker, and Martin Mohr: *Using joana for information flow control in java programs - a practical guide.* In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215, pages 123–138. Springer Berlin / Heidelberg, February 2013. 2, 22

[12] Luo, Linghui, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci: *Taintbench: Automatic real-world malware benchmarking of android taint analyses.* Empirical Software Engineering, 27:1–41, 2022. 3, 14, 28, 32, 53, 54, 56

[13] Engström, Emelie, Margaret-Anne D. Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre: *How software engineering research aligns with design science: a review.* Empir. Softw. Eng., 25(4):2630–2660, 2020. `https://doi.org/10.1007/s10664-020-09818-7`. 4

[14] Nielson, Flemming, Hanne R Nielson, and Chris Hankin: *Principles of program analysis.* Springer, 2015. 6, 7, 8, 10

[15] Lhoták, Ondřej and Laurie Hendren: *Scaling java points-to analysis using spark.* In *International conference on compiler construction*, pages 153–169. Springer, 2003. 12

[16] Sui, Yulei, Ding Ye, and Jingling Xue: *Static memory leak detection using full-sparse value-flow analysis.* In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264, 2012. 15

[17] Cai, Yuandao, Peisen Yao, and Charles Zhang: *Canary: practical static detection of inter-thread value-flow bugs.* In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1126–1140, 2021. 15

[18] too4words: *securibench micro.* https://github.com/too4words/securibench-micro, 2014. 15

[19] Li, Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon: *Static analysis of android apps: A systematic literature review.* Information and Software Technology, 88:67–95, 2017. 21

[20] StevenArzt: *Droidbench 3.0*, 2013. `https://github.com/secure-software-engineering/DroidBench/tree/9898d71ca9053fc0c918a0aac7a768746d6738b`, visited on 2023-11-01. 22, 28

[21] Luo, Linghui: *Taintbench framework*, 2019. `https://taintbench.github.io/taintbenchFramework/`, visited on 2023-11-14. 28

[22] Wei, Fengguo, Sankardas Roy, Xinming Ou, and Robby: *Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps.* ACM Transactions on Privacy and Security (TOPS), 21(3):1–32, 2018. 28

[23] Everitt, Brian S. and Torsten Hothorn: *A Handbook of Statistical Analyses Using R, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2009, ISBN 1420079336. 32

[24] Hair, Joseph F: *Multivariate data analysis.* 2009. 33

[25] Green, Jennifer L., Sarah E. Manski, Timothy A. Hansen, and Jennifer E. Broatch: *Descriptive statistics.* In Tierney, Robert J, Fazal Rizvi, and Kadriye Ercikan (editors): *International Encyclopedia of Education (Fourth Edition)*, pages 723–733. Elsevier, Oxford, fourth edition edition, 2023, ISBN 978-0-12-818629-9. `https://www.sciencedirect.com/science/article/pii/B9780128186305100831`. 33

[26] Arzt, Steven: *Sootup*, 2023. `https://soot-oss.github.io/SootUp/`, visited on 2023-07-09. 61

Appendix A

JSVFA-v0.3.0 metrics

JSVFA initial metrics

SUMMARY (*computed in June 2023.*)

- **securibench.micro** - failed: 46, passed: 57 of 103 tests - (55.34%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score	Pass Rate
aliasing	10	12	2/6	8	1	3	0.89	0.73	0.80	33.33
arrays	0	9	1/10	0	0	9	0.00	0.00	0.00	10
basic	60	60	36/42	52	3	3	0.95	0.95	0.95	85.71
collections	3	15	1/14	1	1	13	0.50	0.07	0.12	7.14
datastructures	7	5	4/6	4	2	0	0.67	1.00	0.80	66.67
factories	4	3	2/3	2	1	0	0.67	1.00	0.80	66.67
inter	10	18	7/14	8	0	8	1.00	0.50	0.67	50
session	0	3	0/3	0	0	3	0.00	0.00	0.00	0
strong_updates	0	1	4/5	0	0	1	0.00	0.00	0.00	80
TOTAL	94	126	57/103	75	8	40	0.90	0.65	0.75	55.34

Details (*computed in June, 2025.*)

- **securibench.micro.aliasing** - failed: 4, passed: 2 of 6 tests - (33.33%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Aliasing1	1	1		1	0	0	1.00	1.00	1.00
Aliasing2	0	1		0	0	1	0.00	0.00	0.00
Aliasing3	0	1		0	0	1	0.00	0.00	0.00
Aliasing4	2	1		0	1	0	0.00	0.00	0.00
Aliasing5	0	1		0	0	1	0.00	0.00	0.00
Aliasing6	7	7		7	0	0	1.00	1.00	1.00
TOTAL	10	12	2/6	8	1	3	0.89	0.73	0.80

- **securibench.micro.arrays** - failed: 9, passed: 1 of 10 tests - (10%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Arrays1	0	1		0	0	1	0.00	0.00	0.00
Arrays2	0	1		0	0	1	0.00	0.00	0.00
Arrays3	0	1		0	0	1	0.00	0.00	0.00
Arrays4	0	1		0	0	1	0.00	0.00	0.00
Arrays5	0	0		0	0	0	0.00	0.00	0.00
Arrays6	0	1		0	0	1	0.00	0.00	0.00
Arrays7	0	1		0	0	1	0.00	0.00	0.00
Arrays8	0	1		0	0	1	0.00	0.00	0.00
Arrays9	0	1		0	0	1	0.00	0.00	0.00
Arrays10	0	1		0	0	1	0.00	0.00	0.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
TOTAL	0	9	1/10	0	0	9	0.00	0.00	0.00

- **securibench.micro.basic** - failed: 6, passed: 36 of 42 tests - (85.71%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Basic0	1	1		1	0	0	1.00	1.00	1.00
Basic1	1	1		1	0	0	1.00	1.00	1.00
Basic2	1	1		1	0	0	1.00	1.00	1.00
Basic3	1	1		1	0	0	1.00	1.00	1.00
Basic4	1	1		1	0	0	1.00	1.00	1.00
Basic5	3	3		3	0	0	1.00	1.00	1.00
Basic6	1	1		1	0	0	1.00	1.00	1.00
Basic7	1	1		1	0	0	1.00	1.00	1.00
Basic8	1	1		1	0	0	1.00	1.00	1.00
Basic9	1	1		1	0	0	1.00	1.00	1.00
Basic10	1	1		1	0	0	1.00	1.00	1.00
Basic11	2	2		2	0	0	1.00	1.00	1.00
Basic12	2	2		2	0	0	1.00	1.00	1.00
Basic13	1	1		1	0	0	1.00	1.00	1.00
Basic14	1	1		1	0	0	1.00	1.00	1.00
Basic15	1	1		1	0	0	1.00	1.00	1.00
Basic16	1	1		1	0	0	1.00	1.00	1.00
Basic17	2	1		0	1	0	0.00	0.00	0.00
Basic18	1	1		1	0	0	1.00	1.00	1.00
Basic19	1	1		1	0	0	1.00	1.00	1.00
Basic20	1	1		1	0	0	1.00	1.00	1.00
Basic21	4	4		4	0	0	1.00	1.00	1.00
Basic22	1	1		1	0	0	1.00	1.00	1.00
Basic23	3	3		3	0	0	1.00	1.00	1.00
Basic24	1	1		1	0	0	1.00	1.00	1.00
Basic25	1	1		1	0	0	1.00	1.00	1.00
Basic26	1	1		1	0	0	1.00	1.00	1.00
Basic27	1	1		1	0	0	1.00	1.00	1.00
Basic28	1	2		0	0	1	0.00	0.00	0.00
Basic29	2	2		2	0	0	1.00	1.00	1.00
Basic30	1	1		1	0	0	1.00	1.00	1.00
Basic31	3	2		0	1	0	0.00	0.00	0.00
Basic32	1	1		1	0	0	1.00	1.00	1.00
Basic33	1	1		1	0	0	1.00	1.00	1.00
Basic34	2	2		2	0	0	1.00	1.00	1.00
Basic35	6	6		6	0	0	1.00	1.00	1.00
Basic36	0	1		0	0	1	0.00	0.00	0.00
Basic37	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Basic38	2	1		0	1	0	0.00	0.00	0.00
Basic39	1	1		1	0	0	1.00	1.00	1.00
Basic41	1	1		1	0	0	1.00	1.00	1.00
Basic42	0	1		0	0	1	0.00	0.00	0.00
TOTAL	60	60	36/42	52	3	3	0.95	0.95	0.95

- **securibench.micro.collections** - failed: 13, passed: 1 of 14 tests - (7.14%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Collections1	1	1		1	0	0	1.00	1.00	1.00
Collections2	2	1		0	1	0	0.00	0.00	0.00
Collections3	0	2		0	0	2	0.00	0.00	0.00
Collections4	0	1		0	0	1	0.00	0.00	0.00
Collections5	0	1		0	0	1	0.00	0.00	0.00
Collections6	0	1		0	0	1	0.00	0.00	0.00
Collections7	0	1		0	0	1	0.00	0.00	0.00
Collections8	0	1		0	0	1	0.00	0.00	0.00
Collections9	0	1		0	0	1	0.00	0.00	0.00
Collections10	0	1		0	0	1	0.00	0.00	0.00
Collections11	0	1		0	0	1	0.00	0.00	0.00
Collections12	0	1		0	0	1	0.00	0.00	0.00
Collections13	0	1		0	0	1	0.00	0.00	0.00
Collections14	0	1		0	0	1	0.00	0.00	0.00
TOTAL	3	15	1/14	1	1	13	0.50	0.07	0.12

- **securibench.micro.datastructures** - failed: 2, passed: 4 of 6 tests - (66.67%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Datastructures1	1	1		1	0	0	1.00	1.00	1.00
Datastructures2	2	1		0	1	0	0.00	0.00	0.00
Datastructures3	1	1		1	0	0	1.00	1.00	1.00
Datastructures4	1	0		0	1	0	0.00	0.00	0.00
Datastructures5	1	1		1	0	0	1.00	1.00	1.00
Datastructures6	1	1		1	0	0	1.00	1.00	1.00
TOTAL	7	5	4/6	4	2	0	0.67	1.00	0.80

- **securibench.micro.factories** - failed: 1, passed: 2 of 3 tests - (66.67%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Factories1	1	1		1	0	0	1.00	1.00	1.00
Factories2	1	1		1	0	0	1.00	1.00	1.00
Factories3	2	1		0	1	0	0.00	0.00	0.00
TOTAL	4	3	2/3	2	1	0	0.67	1.00	0.80

- **securibench.micro.inter** - failed: 7, passed: 7 of 14 tests - (50%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Inter1	1	1		1	0	0	1.00	1.00	1.00
Inter2	2	2		2	0	0	1.00	1.00	1.00
Inter3	1	1		1	0	0	1.00	1.00	1.00
Inter4	0	2		0	0	2	0.00	0.00	0.00
Inter5	1	2		0	0	1	0.00	0.00	0.00
Inter6	0	1		0	0	1	0.00	0.00	0.00
Inter7	0	1		0	0	1	0.00	0.00	0.00
Inter8	1	1		1	0	0	1.00	1.00	1.00
Inter9	1	2		0	0	1	0.00	0.00	0.00
Inter10	1	1		1	0	0	1.00	1.00	1.00
Inter11	0	1		0	0	1	0.00	0.00	0.00
Inter12	0	1		0	0	1	0.00	0.00	0.00
Inter13	1	1		1	0	0	1.00	1.00	1.00
Inter14	1	1		1	0	0	1.00	1.00	1.00
TOTAL	10	18	7/14	8	0	8	1.00	0.50	0.67

- **securibench.micro.session** - failed: 3, passed: 0 of 3 tests - (0%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Session1	0	1		0	0	1	0.00	0.00	0.00
Session2	0	1		0	0	1	0.00	0.00	0.00
Session3	0	1		0	0	1	0.00	0.00	0.00
TOTAL	0	3	0/3	0	0	3	0.00	0.00	0.00

- **securibench.micro.strong_updates** - failed: 1, passed: 4 of 5 tests - (80%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
StrongUpdates1	0	0		0	0	0	0.00	0.00	0.00
StrongUpdates2	0	0		0	0	0	0.00	0.00	0.00
StrongUpdates3	0	0		0	0	0	0.00	0.00	0.00
StrongUpdates4	0	1		0	0	1	0.00	0.00	0.00
StrongUpdates5	0	0		0	0	0	0.00	0.00	0.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
TOTAL	0	1	4/5	0	0	1	0.00	0.00	0.00

OBSERVATIONS

- Flowdroid is not taking in count the TP expected in StrongUpdate4;
- Test Basic40 is commented in the test suite so the amount of TP differs from the original run by Flowdroid;
- There are two flaky tests: Basic6 and Inter11.

  Extra Test These tests are not executed by Flowdroid

- **securibench.micro.pred** - failed: 3, passed: 6 of 9 tests - (66.67%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Pred1	0	0		0	0	0	0.00	0.00	0.00
Pred2	1	1		1	0	0	1.00	1.00	1.00
Pred3	1	0		0	1	0	0.00	0.00	0.00
Pred4	1	1		1	0	0	1.00	1.00	1.00
Pred5	1	1		1	0	0	1.00	1.00	1.00
Pred6	1	0		0	1	0	0.00	0.00	0.00
Pred7	1	0		0	1	0	0.00	0.00	0.00
Pred8	1	1		1	0	0	1.00	1.00	1.00
Pred9	1	1		1	0	0	1.00	1.00	1.00
TOTAL	8	5	6/9	5	3	0	0.63	1.00	0.77

- **securibench.micro.reflection** - failed: 4, passed: 0 of 4 tests - (0.0%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Refl1	0	1		0	0	1	0.00	0.00	0.00
Refl2	0	1		0	0	1	0.00	0.00	0.00
Refl3	0	1		0	0	1	0.00	0.00	0.00
Refl4	0	1		0	0	1	0.00	0.00	0.00
TOTAL	0	4	0/4	0	0	4	0.00	0.00	0.00

- **securibench.micro.sanitizers** - failed: 6, passed: 0 of 6 tests - (0.0%) An exception is thrown when the tests run so it was not possible to compute metrics.

Appendix B

JSVFA-v0.6.0 metrics

JSVFA metrics

SUMMARY (*computed in September 2025.*)

- **securibench.micro** - failed: 38, passed: 65 of 103 tests - (63.11%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score	Pass Rate
Aliasing	4	12	1/6	1	1	9	0.50	0.10	0.17	16.67%
Arrays	11	9	5/10	5	4	2	0.56	0.71	0.63	50%
Basic	59	60	37/42	53	2	3	0.96	0.95	0.95	88.1%
Collections	8	15	5/14	5	1	8	0.83	0.38	0.52	35.71%
Datastructures	5	5	4/6	4	1	1	0.80	0.80	0.80	66.67%
Factories	4	3	2/3	2	1	0	0.67	1.00	0.80	66.67%
Inter	12	18	8/14	9	0	6	1.00	0.60	0.75	57.14%
Session	0	3	0/3	0	0	3	0.00	0.00	0.00	8%
StrongUpdates	3	1	3/5	1	2	0	0.33	1.00	0.50	60%
TOTAL	106	126	65/103	80	12	32	0.87	0.71	0.78	63.11%

Details

- **AliasingTest** - failed: 5, passed: 1 of 6 tests - (16.67%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Aliasing1	1	1		1	0	0	1.00	1.00	1.00
Aliasing2	0	1		0	0	1	0.00	0.00	0.00
Aliasing3	0	1		0	0	1	0.00	0.00	0.00
Aliasing4	2	1		0	1	0	0.00	0.00	0.00
Aliasing5	0	1		0	0	1	0.00	0.00	0.00
Aliasing6	1	7		0	0	6	0.00	0.00	0.00
TOTAL	4	12	1/6	1	1	9	0.50	0.10	0.17

- **ArraysTest** - failed: 5, passed: 5 of 10 tests - (50.0%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Arrays1	1	1		1	0	0	1.00	1.00	1.00
Arrays2	3	1		0	2	0	0.00	0.00	0.00
Arrays3	1	1		1	0	0	1.00	1.00	1.00
Arrays4	1	1		1	0	0	1.00	1.00	1.00
Arrays5	1	0		0	1	0	0.00	0.00	0.00
Arrays6	1	1		1	0	0	1.00	1.00	1.00
Arrays7	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Arrays8	2	1		0	1	0	0.00	0.00	0.00
Arrays9	0	1		0	0	1	0.00	0.00	0.00
Arrays10	0	1		0	0	1	0.00	0.00	0.00
TOTAL	11	9	5/10	5	4	2	0.56	0.71	0.63

- **BasicTest** - failed: 5, passed: 37 of 42 tests - (88.1%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Basic0	1	1		1	0	0	1.00	1.00	1.00
Basic1	1	1		1	0	0	1.00	1.00	1.00
Basic2	1	1		1	0	0	1.00	1.00	1.00
Basic3	1	1		1	0	0	1.00	1.00	1.00
Basic4	1	1		1	0	0	1.00	1.00	1.00
Basic5	3	3		3	0	0	1.00	1.00	1.00
Basic6	1	1		1	0	0	1.00	1.00	1.00
Basic7	1	1		1	0	0	1.00	1.00	1.00
Basic8	1	1		1	0	0	1.00	1.00	1.00
Basic9	1	1		1	0	0	1.00	1.00	1.00
Basic10	1	1		1	0	0	1.00	1.00	1.00
Basic11	2	2		2	0	0	1.00	1.00	1.00
Basic12	2	2		2	0	0	1.00	1.00	1.00
Basic13	1	1		1	0	0	1.00	1.00	1.00
Basic14	1	1		1	0	0	1.00	1.00	1.00
Basic15	1	1		1	0	0	1.00	1.00	1.00
Basic16	1	1		1	0	0	1.00	1.00	1.00
Basic17	1	1		1	0	0	1.00	1.00	1.00
Basic18	1	1		1	0	0	1.00	1.00	1.00
Basic19	1	1		1	0	0	1.00	1.00	1.00
Basic20	1	1		1	0	0	1.00	1.00	1.00
Basic21	4	4		4	0	0	1.00	1.00	1.00
Basic22	1	1		1	0	0	1.00	1.00	1.00
Basic23	3	3		3	0	0	1.00	1.00	1.00
Basic24	1	1		1	0	0	1.00	1.00	1.00
Basic25	1	1		1	0	0	1.00	1.00	1.00
Basic26	1	1		1	0	0	1.00	1.00	1.00
Basic27	1	1		1	0	0	1.00	1.00	1.00
Basic28	1	2		0	0	1	0.00	0.00	0.00
Basic29	2	2		2	0	0	1.00	1.00	1.00
Basic30	1	1		1	0	0	1.00	1.00	1.00
Basic31	3	2		0	1	0	0.00	0.00	0.00
Basic32	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Basic33	1	1		1	0	0	1.00	1.00	1.00
Basic34	2	2		2	0	0	1.00	1.00	1.00
Basic35	6	6		6	0	0	1.00	1.00	1.00
Basic36	0	1		0	0	1	0.00	0.00	0.00
Basic37	1	1		1	0	0	1.00	1.00	1.00
Basic38	2	1		0	1	0	0.00	0.00	0.00
Basic39	1	1		1	0	0	1.00	1.00	1.00
Basic41	1	1		1	0	0	1.00	1.00	1.00
Basic42	0	1		0	0	1	0.00	0.00	0.00
TOTAL	59	60	37/42	53	2	3	0.96	0.95	0.95

- **CollectionTest** - failed: 9, passed: 5 of 14 tests - (35.71%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Collections1	1	1		1	0	0	1.00	1.00	1.00
Collections2	1	1		1	0	0	1.00	1.00	1.00
Collections3	1	2		0	0	1	0.00	0.00	0.00
Collections4	1	1		1	0	0	1.00	1.00	1.00
Collections5	0	1		0	0	1	0.00	0.00	0.00
Collections6	0	1		0	0	1	0.00	0.00	0.00
Collections7	0	1		0	0	1	0.00	0.00	0.00
Collections8	0	1		0	0	1	0.00	0.00	0.00
Collections9	0	1		0	0	1	0.00	0.00	0.00
Collections10	2	1		0	1	0	0.00	0.00	0.00
Collections11	1	1		1	0	0	1.00	1.00	1.00
Collections12	0	1		0	0	1	0.00	0.00	0.00
Collections13	0	1		0	0	1	0.00	0.00	0.00
Collections14	1	1		1	0	0	1.00	1.00	1.00
TOTAL	8	15	5/14	5	1	8	0.83	0.38	0.52

- **DataStructureTest** - failed: 2, passed: 4 of 6 tests - (66.67%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Datastructures1	1	1		1	0	0	1.00	1.00	1.00
Datastructures2	1	1		1	0	0	1.00	1.00	1.00
Datastructures3	1	1		1	0	0	1.00	1.00	1.00
Datastructures4	1	0		0	1	0	0.00	0.00	0.00
Datastructures5	0	1		0	0	1	0.00	0.00	0.00
Datastructures6	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
TOTAL	5	5	4/6	4	1	1	0.80	0.80	0.80

- **FactoryTest** - failed: 1, passed: 2 of 3 tests - (66.67%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Factories1	1	1		1	0	0	1.00	1.00	1.00
Factories2	1	1		1	0	0	1.00	1.00	1.00
Factories3	2	1		0	1	0	0.00	0.00	0.00
TOTAL	4	3	2/3	2	1	0	0.67	1.00	0.80

- **InterTest** - failed: 6, passed: 8 of 14 tests - (57.14%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Inter1	1	1		1	0	0	1.00	1.00	1.00
Inter2	2	2		2	0	0	1.00	1.00	1.00
Inter3	1	1		1	0	0	1.00	1.00	1.00
Inter4	1	2		0	0	1	0.00	0.00	0.00
Inter5	1	2		0	0	1	0.00	0.00	0.00
Inter6	0	1		0	0	1	0.00	0.00	0.00
Inter7	1	1		1	0	0	1.00	1.00	1.00
Inter8	1	1		1	0	0	1.00	1.00	1.00
Inter9	1	2		0	0	1	0.00	0.00	0.00
Inter10	1	1		1	0	0	1.00	1.00	1.00
Inter11	0	1		0	0	1	0.00	0.00	0.00
Inter12	0	1		0	0	1	0.00	0.00	0.00
Inter13	1	1		1	0	0	1.00	1.00	1.00
Inter14	1	1		1	0	0	1.00	1.00	1.00
TOTAL	12	18	8/14	9	0	6	1.00	0.60	0.75

- **SessionTest** - failed: 3, passed: 0 of 3 tests - (0.0%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Session1	0	1		0	0	1	0.00	0.00	0.00
Session2	0	1		0	0	1	0.00	0.00	0.00
Session3	0	1		0	0	1	0.00	0.00	0.00
TOTAL	0	3	0/3	0	0	3	0.00	0.00	0.00

- **StrongUpdateTest** - failed: 2, passed: 3 of 5 tests - (60.0%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
StrongUpdates1	0	0		0	0	0	0.00	0.00	0.00
StrongUpdates2	0	0		0	0	0	0.00	0.00	0.00
StrongUpdates3	1	0		0	1	0	0.00	0.00	0.00
StrongUpdates4	1	1		1	0	0	1.00	1.00	1.00
StrongUpdates5	1	0		0	1	0	0.00	0.00	0.00
TOTAL	3	1	3/5	1	2	0	0.33	1.00	0.50

Extra Test

These tests are not executed by Flowdroid

- **securibench.micro.pred** - failed: 3, passed: 6 of 9 tests - (66.67%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Pred1	0	0		0	0	0	0.00	0.00	0.00
Pred2	1	1		1	0	0	1.00	1.00	1.00
Pred3	1	0		0	1	0	0.00	0.00	0.00
Pred4	1	1		1	0	0	1.00	1.00	1.00
Pred5	1	1		1	0	0	1.00	1.00	1.00
Pred6	1	0		0	1	0	0.00	0.00	0.00
Pred7	1	0		0	1	0	0.00	0.00	0.00
Pred8	1	1		1	0	0	1.00	1.00	1.00
Pred9	1	1		1	0	0	1.00	1.00	1.00
TOTAL	8	5	6/9	5	3	0	0.63	1.00	0.77

- **securibench.micro.reflection** - failed: 4, passed: 0 of 4 tests - (0.0%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Refl1	0	1		0	0	1	0.00	0.00	0.00
Refl2	0	1		0	0	1	0.00	0.00	0.00
Refl3	0	1		0	0	1	0.00	0.00	0.00
Refl4	0	1		0	0	1	0.00	0.00	0.00
TOTAL	0	4	0/4	0	0	4	0.00	0.00	0.00

- **securibench.micro.sanitizers** - failed: 4, passed: 2 of 6 tests - (33.33%)

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Sanitizers1	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F-score
Sanitizers2	0	1		0	0	1	0.00	0.00	0.00
Sanitizers3	0	0		0	0	0	0.00	0.00	0.00
Sanitizers4	1	2		0	0	1	0.00	0.00	0.00
Sanitizers5	0	1		0	0	1	0.00	0.00	0.00
Sanitizers6	0	1		0	0	1	0.00	0.00	0.00
TOTAL	2	6	2/6	1	0	4	1.00	0.20	0.33

Appendix C

FlowDroid metrics

FLOWDROID metrics

SUMMARY (*computed in November 2025.*)

failed: 36, passed: 67 of 103 tests. `(65.05%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1	Pass Rate
Aliasing	11	11	4/6	9	1	1	0.90	0.90	0.90	66.67%
Arrays	14	9	6/10	6	5	0	0.55	1.00	0.71	60%
Basic	38	61	26/42	33	1	24	0.97	0.58	0.73	61.90%
Collections	14	15	11/14	12	1	2	0.92	0.86	0.89	78.57%
Datastructures	5	5	4/6	3	1	1	0.75	0.75	0.75	66.67%
Factories	1	3	1/3	1	0	2	1.00	0.33	0.50	33.33%
Inter	15	18	11/14	13	0	3	1.00	0.81	0.90	78.57%
Session	0	3	0/3	0	0	3	0.00	0.00	0.00	0%
StrongUpdates	0	1	4/5	0	0	1	0.00	0.00	0.00	80%
Pred	-	-	-	-	-	-	-	-	-	-
Reflection	-	-	-	-	-	-	-	-	-	-
Sanitizers	-	-	-	-	-	-	-	-	-	-
TOTAL	98	126	67/103	77	9	37	0.90	0.68	0.77	65.05%

According to Flowdroid Paper (https://www.bodden.de/pubs/far+14flowdroid.pdf), they computed the next values.

failed: 0, passed: 0, ignored: 0 of 103 tests.

Test	TP	FP
Aliasing	11/11	0
Array	9/9	6
Basic	58/60	0
Collection	14/14	3
DataStructure	5/5	0
Factory	3/3	0
Inter	14/16	0
~~Pred~~	-	-
~~Reflection~~	-	-
~~Sanitizers~~	-	-
Session	3/3	0
StrongUpdate	0/0	0
TOTAL	117/121	9

DETAILS

- : PASSED; : FAIL

- **AliasingTest** - failed: 2, passed: 4 of 6 tests. `(66.67%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Aliasing1	1	1		1	0	0	1.00	1.00	1.00
Aliasing2	0	0		0	0	0	0.00	0.00	0.00
Aliasing3	1	0		0	1	0	0.00	0.00	0.00
Aliasing4	1	2		0	0	1	0.00	0.00	0.00
Aliasing5	1	1		1	0	0	1.00	1.00	1.00
Aliasing6	7	7		7	0	0	1.00	1.00	1.00
TOTAL	11	11	4/6	9	1	1	0.90	0.90	0.90

- **ArraysTest** - failed: 4, passed: 6 of 10 tests. `(60%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Arrays1	1	1		1	0	0	1.00	1.00	1.00
Arrays2	3	1		0	2	0	0.00	0.00	0.00
Arrays3	1	1		1	0	0	1.00	1.00	1.00
Arrays4	1	1		1	0	0	1.00	1.00	1.00
Arrays5	1	0		0	1	0	0.00	0.00	0.00
Arrays6	1	1		1	0	0	1.00	1.00	1.00
Arrays7	1	1		1	0	0	1.00	1.00	1.00
Arrays8	2	1		0	1	0	0.00	0.00	0.00
Arrays9	1	1		1	0	0	1.00	1.00	1.00
Arrays10	2	1		0	1	0	0.00	0.00	0.00
TOTAL	14	9	6/10	6	5	0	0.55	1.00	0.71

- **BasicTest** - failed: 16, passed: 26 of 42 tests. `(61.90%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Basic1	1	1		1	0	0	1.00	1.00	1.00
Basic2	1	1		1	0	0	1.00	1.00	1.00
Basic3	0	1		0	0	1	0.00	0.00	0.00
Basic4	1	1		1	0	0	1.00	1.00	1.00
Basic5	0	3		0	0	3	0.00	0.00	0.00
Basic6	0	1		0	0	1	0.00	0.00	0.00
Basic7	0	1		0	0	1	0.00	0.00	0.00
Basic8	1	1		1	0	0	1.00	1.00	1.00
Basic9	1	1		1	0	0	1.00	1.00	1.00
Basic10	1	1		1	0	0	1.00	1.00	1.00
Basic11	1	2		0	0	1	0.00	0.00	0.00
Basic12	2	2		2	0	0	1.00	1.00	1.00
Basic13	1	1		1	0	0	1.00	1.00	1.00
Basic14	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Basic15	1	1		1	0	0	1.00	1.00	1.00
Basic16	1	1		1	0	0	1.00	1.00	1.00
Basic17	1	1		1	0	0	1.00	1.00	1.00
Basic18	1	1		1	0	0	1.00	1.00	1.00
Basic19	1	1		1	0	0	1.00	1.00	1.00
Basic20	1	1		1	0	0	1.00	1.00	1.00
Basic21	0	4		0	0	4	0.00	0.00	0.00
Basic22	0	1		0	0	1	0.00	0.00	0.00
Basic23	0	3		0	0	3	0.00	0.00	0.00
Basic24	0	1		0	0	1	0.00	0.00	0.00
Basic25	0	1		0	0	1	0.00	0.00	0.00
Basic26	0	1		0	0	1	0.00	0.00	0.00
Basic27	1	1		1	0	0	1.00	1.00	1.00
Basic28	2	2		2	0	0	1.00	1.00	1.00
Basic29	3	2		0	1	0	0.00	0.00	0.00
Basic30	1	1		1	0	0	1.00	1.00	1.00
Basic31	0	3		0	0	3	0.00	0.00	0.00
Basic32	1	1		1	0	0	1.00	1.00	1.00
Basic33	0	1		0	0	1	0.00	0.00	0.00
Basic34	1	2		0	0	1	0.00	0.00	0.00
Basic35	6	6		6	0	0	1.00	1.00	1.00
Basic36	0	1		0	0	1	0.00	0.00	0.00
Basic37	1	1		1	0	0	1.00	1.00	1.00
Basic38	1	1		1	0	0	1.00	1.00	1.00
Basic39	1	1		1	0	0	1.00	1.00	1.00
Basic40	1	1		1	0	0	1.00	1.00	1.00
Basic41	1	1		1	0	0	1.00	1.00	1.00
Basic42	1	1		1	0	0	1.00	1.00	1.00
TOTAL	38	61	26/42	33	1	24	0.97	0.58	0.73

- **CollectionTest** - failed: 3, passed: 11 of 14 tests. `(78.57%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Collections1	1	1		1	0	0	1.00	1.00	1.00
Collections2	1	1		1	0	0	1.00	1.00	1.00
Collections3	2	2		2	0	0	1.00	1.00	1.00
Collections4	1	1		1	0	0	1.00	1.00	1.00
Collections5	1	1		1	0	0	1.00	1.00	1.00
Collections6	2	1		0	1	0	0.00	0.00	0.00
Collections7	1	1		1	0	0	1.00	1.00	1.00
Collections8	0	1		0	0	1	0.00	0.00	0.00
Collections9	0	1		0	0	1	0.00	0.00	0.00
Collections10	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Collections11	1	1		1	0	0	1.00	1.00	1.00
Collections12	1	1		1	0	0	1.00	1.00	1.00
Collections13	1	1		1	0	0	1.00	1.00	1.00
Collections14	1	1		1	0	0	1.00	1.00	1.00
TOTAL	14	15	11/14	12	1	2	0.92	0.86	0.89

- **DataStructureTest** - failed: 2, passed: 4 of 6 tests. `(66.67%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Datastructures1	2	1		0	1	0	0.00	0.00	0.00
Datastructures2	1	1		1	0	0	1.00	1.00	1.00
Datastructures3	1	1		1	0	0	1.00	1.00	1.00
Datastructures4	0	0		0	0	0	0.00	0.00	0.00
Datastructures5	0	1		0	0	1	0.00	0.00	0.00
Datastructures6	1	1		1	0	0	1.00	1.00	1.00
TOTAL	5	5	4/6	3	1	1	0.75	0.75	0.75

- **FactoryTest** - failed: 2, passed: 1 of 3 tests. `(33.33%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Factories1	0	1		0	0	1	0.00	0.00	0.00
Factories2	0	1		0	0	1	0.00	0.00	0.00
Factories3	1	1		1	0	0	1.00	1.00	1.00
TOTAL	1	3	1/3	1	0	2	1.00	0.33	0.50

- **InterTest** - failed: 3, passed: 11 of 14 tests. `(78.57%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Inter1	1	1		1	0	0	1.00	1.00	1.00
Inter2	2	2		2	0	0	1.00	1.00	1.00
Inter3	1	1		1	0	0	1.00	1.00	1.00
Inter4	1	2		0	0	1	0.00	0.00	0.00
Inter5	1	2		0	0	1	0.00	0.00	0.00
Inter6	0	1		0	0	1	0.00	0.00	0.00
Inter7	1	1		1	0	0	1.00	1.00	1.00
Inter8	1	1		1	0	0	1.00	1.00	1.00
Inter9	2	2		2	0	0	1.00	1.00	1.00
Inter10	1	1		1	0	0	1.00	1.00	1.00
Inter11	1	1		1	0	0	1.00	1.00	1.00
Inter12	1	1		1	0	0	1.00	1.00	1.00
Inter13	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Inter14	1	1		1	0	0	1.00	1.00	1.00
TOTAL	15	18	11/14	13	0	3	1.00	0.81	0.90

- **SessionTest** - failed: 3, passed: 0 of 3 tests. `(0.00%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Session1	0	1		0	0	1	0.00	0.00	0.00
Session2	0	1		0	0	1	0.00	0.00	0.00
Session3	0	1		0	0	1	0.00	0.00	0.00
TOTAL	0	3	0/3	0	0	3	0.00	0.00	0.00

- **StrongUpdateTest** - failed: 1, passed: 4 of 5 tests. `(80%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
StrongUpdate01	0	0		0	0	0	0.00	0.00	0.00
StrongUpdate02	0	0		0	0	0	0.00	0.00	0.00
StrongUpdate03	0	0		0	0	0	0.00	0.00	0.00
StrongUpdate04	0	1		0	0	1	0.00	0.00	0.00
StrongUpdate05	0	0		0	0	0	0.00	0.00	0.00
TOTAL	0	1	4/5	0	0	1	0.00	0.00	0.00

Appendix D

Joana metrics

JOANA Metrics

SUMMARY (*computed in November 2025.*)

- **Securibench** - failed: 37, passed: 85 of 122 tests. `(69.67%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1	Pass rate
Aliasing	6	11	2/6	2	2	7	0.50	0.22	0.31	33.33%
Arrays	10	9	9/10	9	1	0	0.90	1.00	0.95	90%
Basic	45	61	25/42	26	6	22	0.81	0.54	0.65	59.52%
Collections	15	14	13/14	14	1	0	0.93	1.00	0.96	92.86%
Datastructures	6	5	5/6	5	1	0	0.83	1.00	0.91	83.33%
Factories	3	3	3/3	3	0	0	1.00	1.00	1.00	100%
Inter	13	16	11/14	11	0	3	1.00	0.79	0.88	78.57%
Session	3	3	3/3	3	0	0	1.00	1.00	1.00	100%
StrongUpdates	5	1	1/5	1	4	0	0.20	1.00	0.33	20%
Pred	8	5	6/9	5	3	0	0.63	1.00	0.77	66.67%
Reflection	3	4	3/4	3	0	1	1.00	0.75	0.86	75%
Sanitizer	6	6	4/6	4	1	1	0.80	0.80	0.80	66.67%
TOTAL	123	138	85/122	86	19	34	0.82	0.72	0.77	69.67%

DETAILS

- **AliasingTest** - failed: 4, passed: 2 of 6 tests. `(33.33%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Aliasing1	1	1	PASS	1	0	0	1.00	1.00	1.00
Aliasing2	1	0	FAIL	0	1	0	0.00	0.00	0.00
Aliasing3	1	0	FAIL	0	1	0	0.00	0.00	0.00
Aliasing4	1	2	FAIL	0	0	1	0.00	0.00	0.00
Aliasing5	1	1	PASS	1	0	0	1.00	1.00	1.00
Aliasing6	1	7	FAIL	0	0	6	0.00	0.00	0.00
TOTAL	6	11	2/6	2	2	7	0.50	0.22	0.31

- **ArraysTest** - failed: 1, passed: 9 of 10 tests. `(90%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Arrays1	1	1		1	0	0	1.00	1.00	1.00
Arrays2	1	1		1	0	0	1.00	1.00	1.00
Arrays3	1	1		1	0	0	1.00	1.00	1.00
Arrays4	1	1		1	0	0	1.00	1.00	1.00
Arrays5	1	0		0	1	0	0.00	0.00	0.00
Arrays6	1	1		1	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Arrays7	1	1		1	0	0	1.00	1.00	1.00
Arrays8	1	1		1	0	0	1.00	1.00	1.00
Arrays9	1	1		1	0	0	1.00	1.00	1.00
Arrays10	1	1		1	0	0	1.00	1.00	1.00
TOTAL	10	9	9/10	9	1	0	0.90	1.00	0.95

- **BasicTest** - failed: 17, passed: 25 of 42 tests. `(59.52%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Basic1	1	1		1	0	0	1.00	1.00	1.00
Basic2	1	1		1	0	0	1.00	1.00	1.00
Basic3	1	1		1	0	0	1.00	1.00	1.00
Basic4	1	1		1	0	0	1.00	1.00	1.00
Basic5	1	3		0	0	2	0.00	0.00	0.00
Basic6	1	1		1	0	0	1.00	1.00	1.00
Basic7	1	1		1	0	0	1.00	1.00	1.00
Basic8	1	1		1	0	0	1.00	1.00	1.00
Basic9	1	1		1	0	0	1.00	1.00	1.00
Basic10	1	1		1	0	0	1.00	1.00	1.00
Basic11	1	2		0	0	1	0.00	0.00	0.00
Basic12	1	2		0	0	1	0.00	0.00	0.00
Basic13	0	1		0	0	1	0.00	0.00	0.00
Basic14	0	1		0	0	1	0.00	0.00	0.00
Basic15	1	1		1	0	0	1.00	1.00	1.00
Basic16	1	1		1	0	0	1.00	1.00	1.00
Basic17	1	1		1	0	0	1.00	1.00	1.00
Basic18	1	1		1	0	0	1.00	1.00	1.00
Basic19	0	1		0	0	1	0.00	0.00	0.00
Basic20	0	1		0	0	1	0.00	0.00	0.00
Basic21	0	4		0	0	4	0.00	0.00	0.00
Basic22	1	1		1	0	0	1.00	1.00	1.00
Basic23	2	3		0	0	1	0.00	0.00	0.00
Basic24	1	1		1	0	0	1.00	1.00	1.00
Basic25	1	1		1	0	0	1.00	1.00	1.00
Basic26	1	1		1	0	0	1.00	1.00	1.00
Basic27	1	1		1	0	0	1.00	1.00	1.00
Basic28	1	2		0	0	1	0.00	0.00	0.00
Basic29	1	2		0	0	1	0.00	0.00	0.00
Basic30	1	1		1	0	0	1.00	1.00	1.00
Basic31	0	3		0	0	3	0.00	0.00	0.00
Basic32	1	1		1	0	0	1.00	1.00	1.00
Basic33	0	1		0	0	1	0.00	0.00	0.00
Basic34	2	2		2	0	0	1.00	1.00	1.00

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Basic35	12	6		0	6	0	0.00	0.00	0.00
Basic36	1	1		1	0	0	1.00	1.00	1.00
Basic37	1	1		1	0	0	1.00	1.00	1.00
Basic38	1	1		1	0	0	1.00	1.00	1.00
Basic39	1	1		1	0	0	1.00	1.00	1.00
Basic40	0	1		0	0	1	0.00	0.00	0.00
Basic41	0	1		0	0	1	0.00	0.00	0.00
Basic42	0	1		0	0	1	0.00	0.00	0.00
TOTAL	45	61	25/42	26	6	22	0.81	0.54	0.65

- **CollectionTest** - failed: 1, passed: 13 of 14 tests. `(92.86%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Collections1	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections2	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections3	2	2	PASS	2	0	0	1.00	1.00	1.00
Collections4	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections5	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections6	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections7	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections8	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections9	1	0	FAIL	0	1	0	0.00	0.00	0.00
Collections10	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections11	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections12	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections13	1	1	PASS	1	0	0	1.00	1.00	1.00
Collections14	1	1	PASS	1	0	0	1.00	1.00	1.00
TOTAL	15	14	13/14	14	1	0	0.93	1.00	0.96

- **DataStructureTest** - failed: 1, passed: 5 of 6 tests. `(83.33%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Datastructures1	1	1		1	0	0	1.00	1.00	1.00
Datastructures2	1	1		1	0	0	1.00	1.00	1.00
Datastructures3	1	1		1	0	0	1.00	1.00	1.00
Datastructures4	1	0		0	1	0	0.00	0.00	0.00
Datastructures5	1	1		1	0	0	1.00	1.00	1.00
Datastructures6	1	1		1	0	0	1.00	1.00	1.00
TOTAL	6	5	5/6	5	1	0	0.83	1.00	0.91

- **FactoryTest** - failed: 0, passed: 3 of 3 tests. `(100%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Factories1	1	1		1	0	0	1.00	1.00	1.00
Factories2	1	1		1	0	0	1.00	1.00	1.00
Factories3	1	1		1	0	0	1.00	1.00	1.00
TOTAL	3	3	3/3	3	0	0	1.00	1.00	1.00

- **InterTest** - failed: 3, passed: 11 of 14 tests. `(78.57%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Inter1	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter2	1	2	FAIL	0	0	1	0.00	0.00	0.00
Inter3	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter4	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter5	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter6	0	1	FAIL	0	0	1	0.00	0.00	0.00
Inter7	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter8	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter9	1	2	FAIL	0	0	1	0.00	0.00	0.00
Inter10	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter11	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter12	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter13	1	1	PASS	1	0	0	1.00	1.00	1.00
Inter14	1	1	PASS	1	0	0	1.00	1.00	1.00
TOTAL	13	16	11/14	11	0	3	1.00	0.79	0.88

- **SessionTest** - failed: 0, passed: 3 of 3 tests. `(100%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Session1	1	1		1	0	0	1.00	1.00	1.00
Session2	1	1		1	0	0	1.00	1.00	1.00
Session3	1	1		1	0	0	1.00	1.00	1.00
TOTAL	3	3	3/3	3	0	0	1.00	1.00	1.00

- **StrongUpdateTest** - failed: 4, passed: 1 of 5 tests. `(20%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
StrongUpdate1	1	0		0	1	0	0.00	0.00	0.00
StrongUpdate2	1	0		0	1	0	0.00	0.00	0.00
StrongUpdate3	1	0		0	1	0	0.00	0.00	0.00
StrongUpdate4	1	1		1	0	0	1.00	1.00	1.00
StrongUpdate5	1	0		0	1	0	0.00	0.00	0.00
TOTAL	5	1	1/5	1	4	0	0.20	1.00	0.33

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1

Extras

- **PredTest** - failed: 3, passed: 6 of 9 tests. `(66.67%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Pred1	0	0	PASS	0	0	0	0.00	0.00	0.00
Pred2	1	1	PASS	1	0	0	1.00	1.00	1.00
Pred3	1	0	FAIL	0	1	0	0.00	0.00	0.00
Pred4	1	1	PASS	1	0	0	1.00	1.00	1.00
Pred5	1	1	PASS	1	0	0	1.00	1.00	1.00
Pred6	1	0	FAIL	0	1	0	0.00	0.00	0.00
Pred7	1	0	FAIL	0	1	0	0.00	0.00	0.00
Pred8	1	1	PASS	1	0	0	1.00	1.00	1.00
Pred9	1	1	PASS	1	0	0	1.00	1.00	1.00
TOTAL	8	5	6/9	5	3	0	0.63	1.00	0.77

- **ReflectionTest** - failed: 1, passed: 3 of 4 tests. `(75%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Refl1	1	1	PASS	1	0	0	1.00	1.00	1.00
Refl2	1	1	PASS	1	0	0	1.00	1.00	1.00
Refl3	1	1	PASS	1	0	0	1.00	1.00	1.00
Refl4	0	1	FAIL	0	0	1	0.00	0.00	0.00
TOTAL	3	4	3/4	3	0	1	1.00	0.75	0.86

- **SanitizersTest** - failed: 2, passed: 4 of 6 tests. `(66.67%)`

Test	Found	Expected	Status	TP	FP	FN	Precision	Recall	F1
Sanitizers1	1	1	PASS	1	0	0	1.00	1.00	1.00
Sanitizers2	1	1	PASS	1	0	0	1.00	1.00	1.00
Sanitizers3	1	0	FAIL	0	1	0	0.00	0.00	0.00
Sanitizers4	1	2	FAIL	0	0	1	0.00	0.00	0.00
Sanitizers5	1	1	PASS	1	0	0	1.00	1.00	1.00
Sanitizers6	1	1	PASS	1	0	0	1.00	1.00	1.00
TOTAL	6	6	4/6	4	1	1	0.80	0.80	0.80