TU WIEN Informatics

Differentielles Testen von Statischen Taint Analyse Tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Florian Tesarek, BSc
Matrikelnummer 01616261

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in sc. Maria Christakis
Mitwirkung: Univ.Ass. Dipl.-Ing. Samuel Pilz
Valentin Wüstholz, PhD

Wien, 26. November 2024

Florian Tesarek   Maria Christakis

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Differential Testing of Static Taint Analysis Tools

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering und Internet Computing

by

Florian Tesarek, BSc
Registration Number 01616261

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in sc. Maria Christakis
Assistance: Univ.Ass. Dipl.-Ing. Samuel Pilz
Valentin Wüstholz, PhD

Vienna, November 26, 2024

Florian Tesarek | Maria Christakis

Erklärung zur Verfassung der Arbeit

Florian Tesarek, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 26. November 2024

Florian Tesarek

Danksagung

Mein größter Dank gilt meiner Betreuerin Maria Christakis für die tolle Unterstützung während dem kompletten Arbeitsprozess von der ersten Ideenfindung bis zur Unterschrift zwei Seiten weiter vorne. Besonders hervorheben möchte ich sowohl die stets freundliche und angenehme Atmosphäre in unseren regelmäßigen Meetings als auch die teils beängstigend schnellen Rückmeldungen auf sämtliche meiner Fragen und Versionen dieser Arbeit.

Neben Maria Christakis bedanke ich mich auch bei Samuel Pilz und Valentin Wüstholz für die laufende Unterstützung in Form von Ratschlägen, Kritik und Diskussionen über den ganzen Arbeitsprozess hinweg.

Abschließend will ich diese Seite auch noch nutzen, um mich bei all jenen Leuten zu bedanken, die mich abseits der Universität bei meinem Studium unterstützt haben. Dieser Dank gilt besonders meinen Eltern und meiner Partnerin Michelle Köberl, die sich mit mir nicht nur über jeden gefundenen Bug gefreut, sondern mir auch beim orthographischen Korrekturlesen dieser Arbeit eine große Hilfe war.

Acknowledgements

My deepest gratitude goes towards my advisor, Maria Christakis, for her outstanding support throughout the entire work process, from the initial brainstorming to the final signature three pages earlier. I want to especially highlight the consistently friendly and pleasant atmosphere in our regular meetings as well as the almost frighteningly quick feedback she provided on all my questions and drafts of this thesis.

In addition to Maria Christakis, I would also like to thank Samuel Pilz and Valentin Wüstholz for their ongoing support in the form of advice, constructive criticism, and discussions throughout the entire process.

Finally, I want to use this page to thank everyone outside of university who supported me in my studies. My gratitude especially goes to my parents and my partner, Michelle Köberl, who not only celebrated every discovered bug with me but also provided invaluable help in proofreading this thesis.

Kurzfassung

Statische Taint-Analyse-Werkzeuge spielen eine wichtige Rolle bei der automatisierten Identifikation von potenziellen Sicherheitslücken in Software und unterstützen die rechtzeitige Behebung dieser Schwachstellen vor der Veröffentlichung. Diese Werkzeuge überprüfen, ob ein Programm Informationsflüsse von bestimmten Datenquellen zu als sensibel eingestuften Funktionen enthält. Wie jede Software können jedoch auch Statische Taint-Analyse-Werkzeuge selbst Bugs enthalten, welche möglicherweise zu fehlerhaften Analyseergebnissen führen. In dieser Arbeit präsentieren wir unser Tool FlowFuzz, das zufällige Testdaten generiert und diese zum differentiellen Testen von Statischen Taint-Analyse-Werkzeugen verwendet. Da bisher kein ähnlicher Ansatz in diesem Bereich bekannt ist, stellt FlowFuzz den Hauptbeitrag dieser Arbeit dar. Unsere experimentellen Auswertungen bestätigen die Effektivität unserer Herangehensweise, da FlowFuzz bereits Probleme in drei verschiedenen Statischen Taint-Analyse-Werkzeuge finden konnte.

Abstract

Static taint analysis tools are essential for the automatized identification of potential security vulnerabilities and enabling their remediation before software is released. They can be used to check if the information flow in a program contains flows from specific data sources to functions deemed problematic. But as any software, these tools themselves might contain bugs that could probably lead to incorrect analysis results. In this thesis we present our tool FlowFuzz that generates random test cases which are automatically applied to static taint analysis tools using differential testing. As this has not been done before, the tool represents the main contribution of this thesis. Our experimental evaluations confirm the effectiveness of our approach as FlowFuzz already uncovered issues in three different static taint analysis tools.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

As digitalization advances and starts affecting increasing parts of our lives, the security of software also becomes more vital. One method to check code for potential security risks is the use of taint analysis tools. These tools play a crucial role in identifying potential security vulnerabilities, such as leaked passwords or the inadvertent execution of user-controlled code.

At their core, taint analysis tools examine the path of information within an application, checking whether data coming from a set of predefined sources ultimately arrives at a set of predetermined sinks. This analysis is imperative for identifying and mitigating data leaks that could compromise the integrity and confidentiality of sensitive information.

Despite their essential role, it is also important to know that even popular taint analysis tools are not immune to the common problems of software development. Like any software, these tools may contain soundness bugs, introducing the possible risk of missing certain leaks.

This challenge highlights the need for efforts to minimize the amount of soundness bugs in taint analysis tools. One effective way to achieve this goal is to employ differential testing by comparing the results of multiple tools when given the same input.

1.1.1 Our approach

In this thesis we present a general differential testing technique for detecting bugs in static taint analysis tools. To the best of our knowledge this has not been done before, therefore our approach provides a new possibility of improving the robustness of those tools. As part of this work, we implemented a tool capable of carrying out differential testing on a variety of different static taint analysis tools.

1.1.2 Contributions

This thesis makes the following contributions:

- We implement FlowFuzz, the first differential testing tool for testing static taint analysis tools.
- Integrated in FlowFuzz we implement a tool for minimizing found test cases.
- We evaluate the performance of our implementation.
- We submit bugs found during our research and already improve the quality of currently available static taint analysis tools.

1.1.3 Outline

The following chapter 2 gives the needed background knowledge on static taint analysis tools and differential testing. Chapter 3 briefly explains our approach and showcases the functionality of our tool using an example issue. Chapter 4 explains the functionality of our tool and its components in depth. Chapter 5 highlights the experimental evaluations we made of our tool and presents all bug candidates found during our research. Finally chapter 6 gives an overview of related work and chapter 7 concludes the thesis.

CHAPTER 2

Background

In this section we give the necessary background knowledge about static taint analysis tools and differential testing.

2.1 Introduction to Static Taint Analysis Tools

Generally there exist two different types of data flow analysis tools: dynamic tools which track the flow of information while the program is running and static tools which analyze a program without executing it [RS10] [ADS19]. In this paper we will only cover static tools.

Static data flow analysis tools are used to examine a program's source code or binary to determine how data moves through different parts of the code. These tools aim to detect potential vulnerabilities and flaws, such as when untrusted input flows into sensitive operations or secret data is leaked to an untrusted output, allowing developers to identify and fix issues before the software is brought into production [LD14].

Static data flow analysis operates by tracking data from sources such as user inputs or external files to sinks such as database queries, console logs, or command executions where mishandled data can lead to security vulnerabilities. The pseudo-code in Alg. 2.1 shows an example of data flowing from such a source into a sink.

Algorithm 2.1: pseudo-code of a program with a taint leak where user input flows directly into a database query without sanitization, posing a risk of an SQL injection

$userInput \leftarrow readUserInput()$
$query \leftarrow$ "SELECT * FROM users WHERE id = " + $userInput$
return $executeQuery(query)$

Unlike dynamic analysis, which requires the program to be running, static analysis happens before compile time (for interpreted languages like Python) or directly after compile time (for compiled languages like Java), making it useful for catching issues early in the development cycle.

A key feature of many static data flow analysis tools is taint analysis, which marks untrusted data as "tainted" and tracks its flow through the program. The analysis checks whether this potentially harmful or confidential data is validated or sanitized before reaching sensitive operations. If tainted data reaches a sink without being sanitized, the analysis flags this as a potential security risk [ARF+14].

Additionally, many tools support flow-sensitive analysis, which takes into account the order in which statements are executed in the program. This allows the analysis to take into consideration whether variables have been properly validated or sanitized before they reach sensitive areas. Using flow-sensitive analysis increases the accuracy of vulnerability detection and reduces the number of false positives. The pseudo-code in Alg. 2.2 shows the program flow from Alg. 2.1 extended with a sanitization method to prevent the risk of SQL injections.

Algorithm 2.2: pseudo-code of a program with sanitization to avoid taint leaks

$userInput \leftarrow readUserInput()$
$userInput \leftarrow sanitize(userInput)$
$query \leftarrow$ “SELECT * FROM users WHERE id = ” + $userInput$
return $executeQuery(query)$

Static taint analysis tools are used for security auditing, detecting vulnerabilities like SQL injections or cross-site scripting, and ensuring that data handling in code follows best practices. Additionally, static data flow analysis tools help improve code quality by identifying improper data usage or logic errors. Their ability to detect issues before a program goes into production makes them a vital part of building robust and secure applications.

2.2 Differential Testing

Differential testing is a software testing technique that compares the outputs of two or more different programs or implementations in response to the same inputs. This approach is particularly effective for identifying discrepancies between systems that are expected to produce similar results, such as different versions of a software application or distinct implementations of an algorithm [McK98].

As differential testing compares the results of two or more programs, it can be used to address the oracle problem [BHM+15]: no ground truth is needed to conduct tests and discover discrepancies between different programs. Therefore it can be used to uncover issues in complex systems that traditional testing approaches might not have detected.

To implement differential testing effectively, it is essential to have a robust mechanism for input generation, as well as a systematic way to compare outputs. If applied correctly, differential testing can serve as a powerful strategy to enhance software quality and reduce potential security issues.

2.2.1 Test case generation

One of the most important steps in differential testing is the automatic generation of test cases which serve as input sets for the tools under test. It can generally be expected that very few generated test cases will result in differences, but as differential testing is capable of generating numerous test cases, even a small percentage of found differences might lead to the detection of relevant bugs [McK98].

In order to increase the chances of finding discrepancies, the test cases need to be carefully designed. Completely random generated strings could serve as inputs for most tools, but these test cases would not bring up significant findings as almost all results would just report "invalid input". Therefore the test cases need to be designed to dive deep into the logic of the tested tools [McK98]. One way to increase the code coverage and thus the efficiency of the testing procedure is by making sure that all generated test cases are at the very least syntactically correct.

2.3 Fuzz Testing

Fuzz testing is a software testing technique that uses random data as inputs to the program under test. This approach is often used to test programs processing structured inputs, whereas the fuzzer typically is configured to produce test cases that do not contain semantical errors and thus can be parsed successfully. Most modern fuzzers start with an initial seed input that is repeatedly mutated in order to find issues in the program under test [KRC+18].

Different fuzzing approaches can be categorized into black box fuzzing using purely random generated inputs and gray box or white box fuzzing where information derived from the program under test is used to guide the test case generation [BDR22].

CHAPTER 3

Overview

3.1 Architectural Overview of FlowFuzz

In this thesis, we introduce FlowFuzz, a new tool that can be used for differential testing of static taint analysis tools.

The high-level structure of FlowFuzz's components and their interactions are illustrated in graphic 3.1.

As an input FlowFuzz takes a configuration file. This configuration determines the cornerstones of the current test run and specifies among others the length of the generated inputs, the number of differential iterations made and if an automatic minimization should be carried out once a difference is found.

Using this configuration, the first component generates a random generic data flow graph.

The second component consists of multiple code translators, one for each used programming language respectively. The translators generate program source code according to the specification given by the generic data flow graph from the previous component and also take care of compiling binaries if needed. As all translators use the same specification we can make sure that all source codes represent the exact same logic.

In the next step, those source code files are used as inputs in the third component, which executes the static taint analysis tools we want to test against each other.

The fourth component parses all analysis results and compares them with one another. If it finds differences in the results (at least one sink found by at least one tool while not being found by at least one other tool), the data flow graph and analysis results are handed over to the minimizer. If there are no differences, a new random flow is generated and the next test round starts.

If called, the fifth component systematically tries to minimize the data flow graph until it represents the minimal set of instruction that still triggers the path miss. Once this

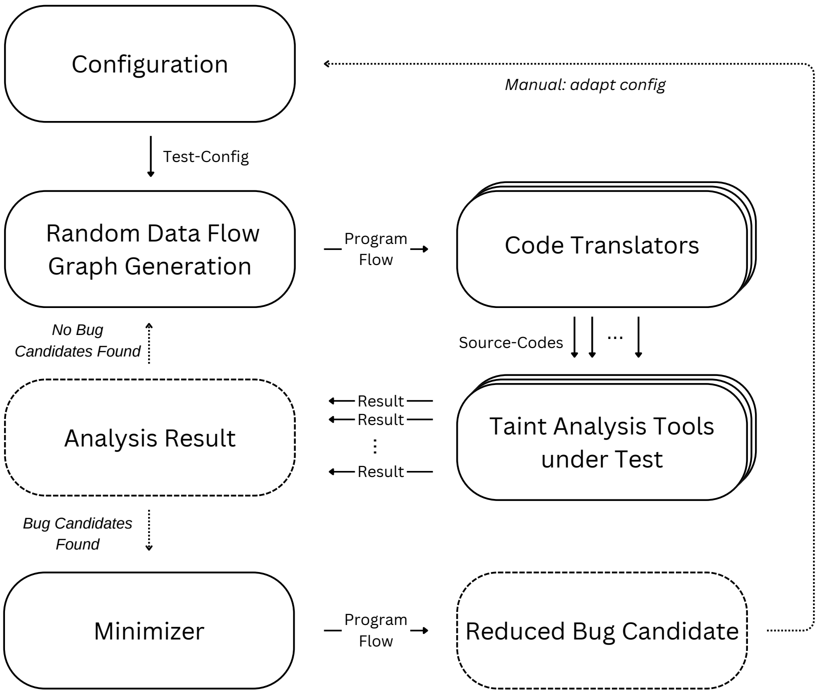

Figure 3.1: Overview of the sequence of steps in a differential test case

minimizer finishes the minimization, the minimized bug candidate is stored for manual inspection.

After manually inspecting the bug candidate and determining if it is indeed a bug, it is reported it to its developers and the test configuration is updated to avoid triggering the same case again.

3.2 Example Issue detected by FlowFuzz

In this section we showcase an example issue that was detected by FlowFuzz in the static taint analyzer FlowDroid (see chapter 5.1.1) while we were running differential testing against the static taint analyzer Pysa (see chapter 5.1.3). This issue was confirmed by one of the FlowDroid developers and partially fixed.

Listing 3.1 shows a small excerpt of the data flow graph generated by FlowFuzz represented in XML format. In total, the generated flow consists of 100 statements. Some of these statements happen to form a taint path which traverses some statements before and after another statement that assigns a negative number to an integer and the declaration of a new list variable.

Listing 3.1: Generated information flow represented in XML

```xml
<functions>
	<function id="1" visibility="public" name="main" return="void">
		<statements>
			<statement id="10" type="variable_declaration" var_name="list1" var_type="list" var_type_list_element="string" />
			<statement id="11" type="list_add">
				<value type="function_call" function_id="2"/>
			</statement>
			<statement id="12" type="variable_declaration" var_name="int1" var_type="int" initial_value="-1" />
			<statement id="13" type="variable_declaration" var_name="list2" var_type="list" var_type_list_element="boolean" />
			<statement id="14" type="function_call" function_id="3" >
				<arguments>
					<argument position="1" type="variable" var_id="10" />
				</arguments>
			</statement>
		...
		</statements>
	</function>
	<function id="2" visibility="public" name="source" return="string" is_source="true"/>
	<function id="3" visibility="public" name="sink" return="void" is_sink="true">
		<arguments>
			<argument var_type="list" var_type_list_element="boolean" />
		</arguments>
	</function>
	...
</functions>
```

Listing 3.2 and listing 3.3 show the example flow from listing 3.1 translated to Java and Python code. This is also only an excerpt of the whole generated files which have around 130 lines of code. After writing these source code files, the code translation component also compiles the Java code to generate a binary that can be analyzed by the tools under test.

After writing and compiling the source codes, the tools under test are executed with those generated test cases as inputs. The outputs generated during the analysis are then parsed and compared. In this case, the analysis results show a difference between the two tools under test: while Pysa indicates 5 leaks, FlowDroid reports only 4.

Since the test case shows a difference, the whole data flow graph is minimized. During minimization most of the statements are removed until the final analysis results show one leak in Pysa and no leaks in FlowDroid. The remaining Java and Python source code files have been reduced to around 25 lines of code and now manly consist of the snippets shown in listing 3.2 and listing 3.3.

During manual inspection, we can now verify that the issue used for this example poses a soundness bug candidate, affecting the recall of FlowDroid. A more detailed description of this issue can be found in the experiments-chapter in section 5.2.2.

Listing 3.2: Translation of the information flow in listing 3.1 to Java code

```java
public void main(){
    List<String> list1 = new ArrayList<String>();
    list1.add(source());

    Integer int1 = -1;
    List<Boolean> list2 = new ArrayList<Boolean>();

    sink(list1);
}

public String source(){
    return "Secret";
}

public void sink(List<String> param){}
...
```

Listing 3.3: Translation of the information flow in listing 3.1 to Python code

```python
def MainFunction(self):
    list1: list[str] = []
    list1.append(self.source())

    int1: int = -1
    list2: list[bool] = []

    self.sink(list1)

def source():
    return "Secret"

def sink(param: list[str]):
    pass
...
```

CHAPTER 4

Approach

4.1 Random Data Flow Graph Generation

Each round of differential testing begins by generating a new input for the tools under test. This input includes a diverse range of statements, capturing various programming language concepts that are arranged randomly.

The random data flow generator produces a specified number of these statements, determined by the parameters set in the configuration. The statements are added one after the other in a context-aware loop which means that, for example, a variable assignment statement can make use of a variable declared in one of the previous statements.

The generator also ensures that the generated data flow is translatable into valid code by avoiding constructs that could cause compilation or interpretation errors, such as accessing a variable before it is declared.

The result of each generation is a valid abstract data flow that can be translated to various programming languages, as long as they support the language concepts used in the generation.

In memory, the generated abstract data flow is stored in custom classes representing all currently supported language concepts. This allows us to easily tailor the needed functionality to satisfy the requirements of each used language concept, like for example statements with multiple blocks of sub-statements as an if/else or a try/catch/finally block.

This flexible architecture not only allows the easy addition of new language concepts but also enables us to easily export the generated data flow as an XML or JSON file without loss of information.

4.2 Translating to Code

The random data flow graph that was generated in the previous step is now handed to multiple code translators which generate language-specific source code according to this specification. Since all translators use the same specification, we can make sure that all source codes represent the same logic which should lead to identical analysis results generated by the tools under test.

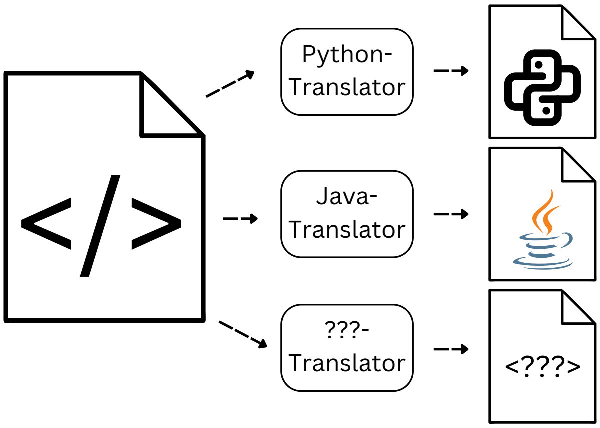

Figure 4.1: Overview of the different code translators, generating program code from the provided generic data flow graph. New translators can be added to support other languages.

After storing the source code files, the translator block also ensures the compilation of all necessary binaries. If the compilation phase raises an error (which might happen during the minimization phase described in chapter 4.4), this information is returned to adapt the input accordingly.

Besides ensuring syntactical correctness while maintaining semantic congruency of the generated source code, the biggest challenge during code translation is to make sure that the generated data flow stays the same even after compilation.

This could be hindered by possible code optimizations the compiler could carry out. Whilst the tools that analyze source code of interpreted languages (like Pysa for Python code in our case) always get exactly the input FlowFuzz generates, the data flow in the input binaries for tools analyzing binaries of compiled languages (like FlowDroid and Mariana Trench for Java) may vary from our generated source code.

The source code of one such case is depicted in listing 4.1.

As we can see, the call to the taint sink $sink1()$ will never be reached during execution. Therefore the compiler might remove this branch all together in order to optimize the code. As other source codes might not receive the same optimization procedure, we could now end up with source files containing different data flow graphs for different tools under test, undermining the very foundation of differential testing.

Listing 4.1: Java code containing a taint path traversing a unreachable location

```
String taint = source();
if(false){
    sink1(taint);
}
```

To circumvent this issue, FlowFuzz needs to make sure to generate code that cannot be further optimized by compilers. Therefore we decided to make all generated conditions runtime-deciding like the improved version of listing 4.1 in listing 4.2 shows.

Listing 4.2: Optimized Java code containing a taint path that can not be declared unreachable at compile-time

```
String taint = source();
if(Math.random() == 1){
    sink1(taint);
}
```

Another example where the compiler could remove part of our generated flow is the usage of exception handling structures where the try-block does not contain any statements that could possibly throw an exception in Java. In this case, the Java compiler also removes the complete catch-block, leading to converging information flows in the multiple generated test inputs. To solve this issue, FlowFuzz simply makes sure to always add one statement that could possibly be throwing exceptions to each generated try-block.

4.3 Bug finding

After preparing all source code files and compiling binaries, this step initiates the execution of each tool under test.

For this execution, it is essential that the source code files and binaries contain identical logic across all tools as well as that the tool configurations are semantically consistent. Another important aspect of the bug finding phase is the comparability of the generated analysis results as they may vary significantly in form and scope depending on the concrete tool under test.

4.3.1 Converging predefined configurations

To achieve consistency in the inputs it is important to closely examine all the tools under test and compare their configuration options in addition to possible predefined settings and rules (as they might vary from tool to tool).

One example illustrating this problem is the propagation of taint through lists. As we can see in listing 4.3, a tainted element is added to a list which consecutively should be marked as tainted itself to correctly analyze this taint flow.

Listing 4.3: Java code showcasing taint propagation via lists

```
List<String> taintList= new ArrayList<String>();
taintList.add(source()); // Add a tainted element to the list
sink(taintList); // Hand the list as argument to a function defined
    as sink
```

Whereas FlowDroid is shipped with configuration that automatically treats such lists as tainted, Mariana Trench does not automatically contain such rules. This convergence in the treatment of lists can lead to many found bug candidates. But this is not a bug as Mariana Trench is also equipped to correctly trace taint flowing through lists once configured accordingly, as the needed propagation model generator depicted in listing 4.4 shows.

Listing 4.4: Propagation Model Generator enabling Mariana Trench to trace taint flowing through lists [Art24b]

```
{
  "find": "methods",
  "where": [
    {
      "constraint": "any_of",
      "inners": [
        {
          "constraint": "signature_pattern",
          "pattern": "Ljava/util/List;\\.add:\\(Ljava/lang/Object;\\)Z"
        },
        {
          "constraint": "signature_pattern",
          "pattern": "Ljava/util/Queue;\\.add:\\(Ljava/lang/Object;\\)Z"
        },
        {
          "constraint": "signature_pattern",
          "pattern": "Ljava/util/Queue;\\.offer:\\(Ljava/lang/Object;\\)Z"
        }
      ]
    }
  ],
  "model": {
    "modes": [
      "skip-analysis",
      "taint-in-taint-this",
      "no-join-virtual-overrides"
    ]
  }
}
```

4.3.2 Mapping sink calls

Another issue during bug finding is the correct classification of found taint sinks. During the generation of random data flow, FlowFuzz may generate multiple calls to the *sink()*

function. Some of these calls might end up propagating taint from calls to the *source()* function to the *sink()* function while others just receive untainted data. The task of the static taint flow analysis tools under test is to figure out which *sink()* calls should be considered problematic as they contain taint from a *source()* call.

To facilitate differential testing, FlowFuzz needs to be able to map each reported sink to a concrete statement in the generated data flow graph. This poses a challenge as tracing the concrete statements across multiple programming language source codes and binaries is not trivial.

To illustrate this problem, we take a look at generated Java code containing a taint path (see listing 4.5) and how the tool FlowDroid reports this path (see listing 4.6).

Listing 4.5: Java code containing a taint path and a benign call of the sink function

```java
public void onCreate(Bundle savedInstanceState){
    ...
    String svar10 = source();
    sink(svar10);
    sink("no_taint");
}
```

Listing 4.6: FlowDroid result for the flow in listing 4.5

```xml
<Result>
    <Sink Statement="virtualinvoke r0.&lt;com.example.testapp.MainActivity: void sink(java.lang.String);(r3)"
        LineNumber="29" Category="no_category"
        Method="&lt;com.example.testapp.MainActivity: void onCreate(android.os.Bundle);"
        MethodSourceSinkDefinition="com.example.testapp.MainActivity: void sink(java.lang.String);"
        SystemCategory="NO_CATEGORY" UserCategory="NO_CATEGORY">
        <AccessPath Value="r3" Type="java.lang.String" TaintSubFields="true"></AccessPath>
    </Sink>
    <Sources>
        <Source Statement="r3 = virtualinvoke r0;com.example.testapp.MainActivity: java.lang.String source();()"
            LineNumber="28" Category="no_category"
            Method=";com.example.testapp.MainActivity: void onCreate(android.os.Bundle);"
            MethodSourceSinkDefinition=";com.example.testapp.MainActivity: java.lang.String source();"
            SystemCategory="NO_CATEGORY" UserCategory="NO_CATEGORY">
            <AccessPath Value="r3" Type="java.lang.String" TaintSubFields="true"></AccessPath>
        </Source>
    </Sources>
</Result>
```

As we can see in the result listing 4.6 the only information FlowDroid returns about the detected taint sink are its signature *com.example.testapp.MainActivity: void sink(java.lang.String)*, the signature of the function it is called from *com.example.testapp.MainActivity: void onCreate(android.os.Bundle)* and the line number of the statement.

The line numbers are referencing the .apk binary, not our source code file and are therefore not useful for tracing back the original statement. As we also have multiple calls to *sink* in our *onCreate()* function (see listing 4.5), FlowFuzz has no way to certainly identify which if these calls has been detected by FlowDroid.

In order to circumvent this problem, FlowFuzz generates a new sink function for each *sink()* call it wants to make. Applying this technique to our code from listing 4.5, we get the new snippet in listing 4.7 where each *sink()* call has a unique name.

Listing 4.7: Java code containing a taint path using multiple sink functions

```
1 public void onCreate(Bundle savedInstanceState){
2     ...
3     String svar10 = source();
4     sink1(svar10);
5     sink2("no_taint");
6 }
```

The resulting analysis result from FlowDroid looks similar to listing 4.6 with the only difference being the signature of the reported sink *com.example.testapp.MainActivity: void sink1(java.lang.String)* thus allowing us to map this result back to the concrete statement in line 4 of our source code listing 4.7.

4.3.3 Ground truth and manual inspection

Once identical inputs (code and configuration) as well as the correct parsing of the outputs are ensured, the tested tools can be run and the obtained analysis results can be compared against each other. As the inputs are randomly generated and therefore no ground truth is available for comparison, the criteria for differentiating inconspicuous results from the ones that yield potential bug candidates are the differences in findings between all tools under test.

If one tool reports a leak that another tool does not report, this could be imprecision in the reporting tool if the detected issue is a false positive. The other option is a soundness issue in the tool which does not report the leak if the found leak is a false negative.

Because there is no ground truth which could help determine which of the tools made a mistake, those test cases containing bug candidates must be retained for manual inspection.

4.4 Minimization

Each found bug candidate needs to be manually examined to check if it exposes a bug in the tool under test. To make this examination as easy as possible, a generated test case that induces different results in the tested tools needs to be minimized to the shortest version that qualifies for examination [McK98].

A typical minimization as shown in Alg. 4.1 consists of applying shortening transformations that remove part of the input and a re-testing phase where the tools under test are run again using the shortened input. If the discrepancy prevails, the test case is replaced by the new shortened version, otherwise the shortening is undone. In both cases, the cycle starts again with the next shortening transformation, until all statements that can be removed from the input set have been tested for their necessity.

Algorithm 4.1: Pseudo-code of a typical minimization phase

```
Input: A program prog, a missed sink sink
Output: The minimized program
for each statement in prog.statements do
    remove statement from prog
    result ← run tools under test
    if result doesn't contain sink then
        add statement to prog
    end
end
return prog
```

This minimization phase is especially important as many automatically generated inputs contain hundreds or thousands of lines of code. That does not only make manual inspection laborious for the team running the differential tests but also hinders the following removal of the bug as developers are less likely to fix or even confirm bug reports if the input files are not easy to grasp. To get the most benefit from differential testing while also conserving time resources, generating short and precise test cases that contain nothing more than the necessary steps for reproducing an issue is essential.

Evidence from the testing of compiler bugs also has shown that most test cases revealing bugs are rather small as 80% of them contain less than 45 lines of code [Bar18] [SLZS16], delivering another argument for carrying out test case minimization.

Therefore, the minimization phase is an important part of differential testing as it facilitates the generation of short input sets that can easily be manually inspected, understood and subsequently reported to the developers of the respective tool.

As our goals for the minimization phase are having a fast minimization as well as removing as many statements as possible while keeping all significant differences, the minimization itself is split into multiple phases.

4.4.1 Pre-Minimization Phase

Before starting with the actual minimization, the minimization component looks at the analysis result to select promising minimization targets.

If there is only one difference between all tested tools (meaning that there is only one sink that is found by some but not all tools under test), the minimization is carried out for this sink.

If there are multiple missed sinks and some of them are unique misses (meaning that all but one tool found this sink) we can assume that all of those misses might pose as potentially interesting bug candidates. In this case, the minimization is carried out multiple times, once for each unique miss.

4.4.2 Fast and Precise Minimization Phase

As it is generally the case in our tests, the most resource-intensive part is running the tools under test. Therefore the minimization starts with the fast minimization phase which aims at increasing efficiency by running the tools under test as infrequent as possible.

In the fast minimization phase the minimizer removes multiple statements at once and runs the tools under test just once after the complete change. If the missed sink is retained, the minimizer continues to remove the next bulk of statements. Only if the missed sink is lost, all statements that were removed in the current step are restored and afterwards removed again one by one in the precise minimization phase.

4.4.3 Recursive Minimization Phase

If during the precise minimization phase the minimizer reaches a statement that contains a body with nested sub-statements (like loops, branching with if/else blocks or an error handling block with try/catch), it can ask two questions:

1. Is the statement necessary as a whole? In order to find out, the minimizer removes the whole statement including all sub-statements and checks if this changes the significant results of the tools under test. If so, the statement is restored and the minimizer starts a recursive minimization of all nested sub-statements in the statement body. This minimization does not differ from the normal minimization procedure described before. The result of this first minimization phase applied to the code in listing 4.8 can be seen in listing 4.9.

2. Do the statements need to be nested or can they be lifted up in the main function body? After the recursive minimization of the statement's body is finished, the minimizer removes the nested statement and adds all statements from inside of its body one level up. Again, if this changes the significant results of the tools under test, this transformation is reversed. The result of this operation applied to our example from before can be seen in listing 4.10.

Listing 4.8: Example Java code snippet before recursive minimization

```java
String taint;
for(int i = 0; i < 10; i++){
    taint = source();
    String not_needed = "test";
}
String not_needed2 = "test2";
sink(taint);
```

Listing 4.9: Example Java code snippet after minimization phase including recursive minimization

```java
String taint;
for(int i = 0; i < 10; i++){
    taint = source();
}
sink(taint);
```

Listing 4.10: Example Java code snippet after moving statements one level up

```java
String taint;
taint = source();
sink(taint);
```

4.4.4 Minimization Order

As we are dealing with a control flow that is used to generate program source code, the minimization always needs to ensure that the minimized flow is still translatable to valid source code. The biggest possible issue in this case would be removing the declaration of a variable that is still used in the subsequent lines of code, as this would lead to compilation/interpretation errors and thus prevent us from successfully running the tools under test again. The simplest approach to circumvent this issue while still being able to remove all declarations of variables that are unnecessary to trigger the current bug candidate is by starting the minimization process bottom-up.

This can be illustrated through the example code presented in listing 4.11. This code consists of three statements that are necessary to reproduce a found bug candidate as well as two additional statements which can be removed as they do not contribute to the current bug candidate.

If the minimization starts top-down , first the variable declarations (line 1 and 2) are removed, but since both removals lead to compilation/interpretation errors because both variables are used in the subsequent lines, those statements are restored.

Listing 4.11: Example Java code snippet before minimization

```
1 String taint;           // Statement needed for reproducing the bug
2 String not_needed;      // Obsolete statement
3 taint = source();       // Statement needed for reproducing the bug
4 not_needed = "text";    // Obsolete statement
5 sink(taint);            // Statement needed for reproducing the bug
```

Next the assignment statements (lines 3 and 4) and the function call (line 5) are removed. The statements in line 3 and 5 contribute to the current bug candidate, therefore the bug is not triggered anymore after removal, so they are restored. The minimization of line 4, however, did not change the results of the tools under test, therefore it stays removed. The minimized program can be seen in listing 4.12.

Listing 4.12: Example Java code snippet after top-down minimization

```
String taint;
String not_needed;      // Removing caused a compilation error,
    therefore this statement was restored
taint = source();
sink(taint);
```

Looking at the minimization starting bottom-up, we can see that, at first, lines 5, 4 and 3 are removed, but because 5 and 3 contribute to the bug candidate, they are restored. Once the minimization reaches the variable declarations (lines 2 and 1), the declaration in line 2 has become obsolete and thus can be removed. Line 1 is restored after minimization, as it is still in use and necessary for the bug candidate. The final minimization result can be seen in listing 4.13.

Listing 4.13: Example Java code snippet after bottom-up minimization

```
String taint;
taint = source();
sink(taint);
```

This shows that better minimization results can be obtained by simply reversing the order of the minimization from top-down to bottom-up without having to deal with interdependencies of the involved statements or iterating over the whole code again.

The complete pseudo-code of our minimizer including the thoughts from the previous sections can be seen in Alg. 4.2.

Algorithm 4.2: Complete pseudo-code of the minimization phase

```
Input: A program prog, a missed sink sink
Output: The minimized program
remove_amount ← multiple
for each function in program.functions do
    for each statement[remove_amount] in reverse(function.statements) do
        remove statement[remove_amount] from function
        result ← run tools under test
        if result doesn't contain sink then
            add statement[remove_amount] to function
            if remove_amount is multiple then
                remove_amount ← single
            end
        else if statement[remove_amount] has body then
            minimize(statement → body)
            move_up(statement → body)
            result ← run tools under test
            if result doesn't contain sink then
                undo_move_up(statement → body)
            end
        end
    end
end
return prog
```

CHAPTER 5

Experimental Evaluations

5.1 The Tools we test

This section gives an overview of the tested static taint analysis tools FlowDroid (FD), Mariana Trench (MT) and Pysa (PY).

5.1.1 FlowDroid

FlowDroid is an open source static data flow analysis tool which was originally developed and is continuously maintained by researchers at the Technical University of Darmstadt [ARF+14]. The implementation of FlowDroid is based on the Java optimization framework Soot [Ac12].

> "FlowDroid statically computes data flows in Android apps and Java programs. Its goal is to provide researchers and practitioners with a tool and library on which they can base their own research projects and product implementations [Ac18]."

The tool can be used through its command-line interface or embedded in a project as a library. It is supported on Windows, Mac OS and Linux [Ac18].

FlowDroid offers a variety of configuration options, including also the possibility to define custom taint sources and sinks using an XML file.

Designed to analyze Android applications and other Java programs, FlowDroid operates directly on binaries built by Gradle and does not need access to the source code to compute its analysis. If configured, the analysis results can be exported as an XML file, detailing all source/sink pairs along with the witness path between them.

5.1.2 Mariana Trench

Similar to FlowDroid, Mariana Trench was also developed to analyze Android applications. It is developed and maintained by Meta Platforms, Inc. and was published to GitHub as part of the Meta Open Source initiative [Sou20a].

> "Mariana Trench is a security focused static analysis platform targeting Android. The tool provides an extensible global taint analysis similar to pre-existing tools like Pysa for Python.
>
> By default Mariana Trench analyzes dalvik bytecode and can work with or without access to the source code [Sou24a]."

Like FlowDroid, Mariana Trench can be used to analyze Android applications as well as vanilla Java projects. It can be installed using the Python package manager pip and is run via its command-line interface.

Mariana Trench is shipped with a set of predefined sources and sinks which can already catch common security vulnerabilities such as remote code execution. Additional custom sources and sinks can be added using a JSON file.

Mariana Trench writes its analysis results as a "set of specifications for each method of the application" [Sou24a]. To convert those specifications into a human readable format, the tool SAPP [Sou20b] can be used. The resulting database includes all found sources and sinks as well as the witness paths between them and can be viewed using a local web application included in SAPP.

5.1.3 Pysa

Pysa, a component of the Pyre type checker, is developed by Meta Platforms, Inc. and published on GitHub as part of the Meta Open Source initiative [Sou17]. As Pyre, Pysa can be used to analyze applications written in Python.

> "[Pysa] can [...] run static analysis, more specifically called Taint Analysis, to identify potential security issues [Sou24b]."

The installation and usage of Pysa is partially similar to Mariana Trench: the installation can be done via the Python package manager pip and the tool is run via a command-line interface.

Pysa also allows the definition of custom sources and sinks via a text file. The analysis results are returned in a JSON file, containing the code positions (file name, column and line) of all found sinks. This file can also be postprocessed and viewed via SAPP.

Bug candidate	Type	Status
FD #779	Imprecision	Confirmed & Fixed
FD #767	Unsoundness	Confirmed & Partially Fixed
MT #174	Unsoundness	Confirmed (Fix planned)
PY #923	Imprecision	Confirmed
MT #172	Imprecision	Confirmed
MT #173	Imprecision	Confirmed [1]
FD #735	Unsoundness	Reported
FD #764	Unsoundness	Reported
FD #781	Unsoundness	Reported
FD #763	Imprecision	Reported
FD #775	Functional	Confirmed & Fixed

Table 5.1: Overview of all found bug candidates in FlowDroid (FD), Mariana Trench (MT) and Pysa (PY). The numbers relate to the issue ID from the respective GitHub issue tracker.

5.2 Found Issues

FlowFuzz already proved its effectiveness by uncovering bugs and bug candidates in the tools under test which can be seen in table 5.1. This section gives an overview of the generated test cases that discovered each of the mentioned bug candidates.

If not presented differently in the listing, all bug candidates were found in an information flow generated in the main function of the main class. They make use of a *source()* function that is defined as a taint source as well as a *sink()* function defined as a taint sink, both of which are implemented as non static methods of the main class.

5.2.1 FlowDroid #779: false positive due to Alias approximation

This bug candidate affects the precision of FlowDroid, where over-approximated aliases can lead to false positives as depicted in listing 5.1. The issue was promptly recognized and confirmed by the FlowDroid developers and is already resolved.

5.2.2 FlowDroid #767: missed Sinks probably related to Issue in Aliasing algorithm

This bug candidate affects the recall of FlowDroid, as can be seen in the source code showing taint flowing from a source to a sink with two uninvolved statements in between, depicted in listing 5.2.

Originally, FlowDroid would not report that taint leak unless the assignment in line 5 was changed to a positive integer or completely removed, or the assignment in line 6

[1]This issue is caused by a false positive by design, see section 5.2.6

Listing 5.1: Java code that leads to a false positive due to problems in alias approximation

```java
List<Boolean> lvar = new ArrayList<Boolean>();
Boolean bvar = true;

lvar.add(source());
lvar.add(bvar);

sink(bvar); // Reported by FlowDroid (false positive)
```

was removed. A fix released by the FlowDroid developers with commit f5d7449 slightly changed but did not solve this issue, whereas now only the deletion of the assignment in line 6 makes FlowDroid correctly report this leak.

Listing 5.2: Java code that triggers a sink miss probably related to the used aliasing algorithm

```java
1 List<String> taint_list = new ArrayList<String>();
2 taint_list.add(source());
3
4 // If one of those two statements is removed (or the first is changed
     to an int >= 0), the leak is detected
5 Integer unused = -1;
6 List<Boolean> unused2 = new ArrayList<Boolean>();
7
8 sink(taint_list); // NOT detected by FlowDroid (false negative)
```

5.2.3 Mariana Trench #174: missed Sink due to Issue in static Initializers

The source code of this bug candidate concerning the recall of Mariana Trench can be seen in listing 5.3. In contrast to the other bug candidates, this uses a static *source()* function.

The issue has been confirmed by the Mariana Trench developers. It was linked to the incorrect handling of static initializers like the one used in this test case and is already scheduled to be fixed.

5.2.4 Pysa #923: missed Sink caused by Aliasing

This bug candidate in Pysa affects the tool's recall as the generated test case depicted in listing 5.4 leads to a missed taint leak.

As can be seen, Pysa does not seem to trace taint across aliasing the names of lists and therefore does not detect taint that is propagated via shallow copies, respectively aliases.

Listing 5.3: Java code that triggers a false negative in Mariana Trench

```
// Main
MyClass myInstance = new MyClass();
String myString = myInstance.myField;
myInstance.myField = "";
sink(myString); // NOT found by Mariana Trench (false negative)

public static String staticSource(){return "Secret";}

class MyClass{
  String myField = MainActivity.staticSource();
}
```

The Pysa developers' answer to this issue report on GitHub states that they are already aware of this issue but "[...] do not have strong indications that it is leading to many missed issues, so [they] are not currently prioritizing fixing this [Art24d]."

Listing 5.4: Python source code causing a missed sink due to aliasing

```
original: list[str] = []
copy: list[str] = []
copy = original
copy.append(source())
sink(original[0])  # NOT found by Pysa (false negative)
sink(copy[0])  # Found by Pysa
```

5.2.5 Mariana Trench #172: false positive due to try/catch Compilation

This bug candidate concerns the precision of Mariana Trench, the generated test case that uncovered it can be seen in listing 5.5.

This bug is related to the way *finally* blocks are compiled in Java. Briefly summarized, the compiler splits the list declaration and initialization and adds another initialization. Those two initializations create two different memory locations, which the *list* variable could point to. As those two locations are treated separately in Mariana Trench, the taint is not always correctly reset, thus leading to the false positive [Art24c].

5.2.6 Mariana Trench #173: false positive with Object tainting

This bug candidate concerns the precision of Mariana Trench. The source code to reproduce this issue can be seen in listing 5.6. The developers of Mariana Trench confirmed this issue as a false positive by design: "[...] without that behavior, we would

Listing 5.5: The compiled version of this Java code triggers a false positive in Mariana Trench

```
for(int i = 0; i < 10; i++){
    try{
        Boolean trigger = false;
    }catch(Exception e){
    }finally{
        List<String> list = new ArrayList<String>();
        sink(list); // Reported as issue (false positive)
        list.add(source());
    }
}
```

have a lot of false negatives. The design choice in our analyzers is to favor false positives (such as this one) rather than false negatives [Art24a]."

Listing 5.6: Java code that triggers a false positive in Mariana Trench

```
// Main
MyClass myInstance = new MyClass();
String myString = myInstance.myField;
myInstance.myField = source();
sink(myString); // Found by Mariana Trench (false positive)

class MyClass{
  String myField = "";
}
```

5.2.7 FlowDroid #735: nondeterministic Behavior

One of the first bug candidates found by FlowFuzz is also the most complex. There seems to be nondeterministic behavior within the implementation of FlowDroid, which sometimes makes it miss some taint sinks that have been found in previous runs given the exact same binary and the exact same configuration, thus negatively affecting the recall of those runs.

FlowFuzz was not able to effectively minimize the input flows that caused this nondeterministic behavior.

This nondeterminism in Java programs like FlowDroid could have multiple different reasons like the iteration over a hash table as the returned hash values of an object vary from run to run or the use of Java's *Object.toString()* which, among other things, returns the current memory address of the given object [MWME21].

5.2.8 FlowDroid #764: configuration-dependent false negative

This bug candidate affects the recall of FlowDroid, as it misses the taint path from the *source()* call in *onCreate* to the *sink*, traversing *function1* and *function2* two times depicted in listing 5.7.

Using standard configuration, FlowDroid correctly reports this taint leak. If the command line interface is called with the additional option "–pathreconstructionmode PRECISE" though, FlowDroid fails to report this leak.

Listing 5.7: Java code of the configuration-dependent inconsistency in FlowDroid

```java
// Main
String taint_1 = function1(source());
String taint_2 = function1(taint_1);
sink(taint_2); // NOT reported by FlowDroid -pr PRECISE (false negative)


public String function1(String arg1){
    arg1 = function2(arg1);
    return arg1;
}
public String function2(String arg1){
    return arg1;
}
```

5.2.9 FlowDroid #781: missed Leak after Class Instantiation

This bug candidate affects the recall of FlowDroid, as some sinks are missed.

As can be seen in listing 5.8 the instantiation of a variable that is completely unrelated to the generated flow seems to prevent FlowDroid from correctly detecting the leak propagated via an instance attribute of another variable.

Listing 5.8: Java code leading to a missed sink in FlowDroid

```java
MyClass myVar = new MyClass();
myVar.myString = source();

// If this statement is removed the leak is reported correctly
MyClass mySecondVar = new MyClass();

sink(myVar.myString); // NOT reported by FlowDroid
```

5.2.10 FlowDroid #763: false positive with Sanitation in Loop

This bug candidate concerns the precision of FlowDroid, as it seems to report a false positive given the certain input depicted in listing 5.9.

The common denominator of all input flows causing this issue seems to be the use of a function that sanitizes tainted input within a loop.

Listing 5.9: Java code that triggers a false positive in FlowDroid

```
// Main
String s = "";
// If removed no false positive is reported
for (int i = 0; i < 10; i++) {s = sanitize(source());}
sink(s); // Found by FlowDroid (false positive)


public String sanitize(String arg1) {
    // If removed no false positive is reported
    if (Math.random() == 123) {}
    return "NoSecret";
}
```

5.2.11 FlowDroid #775: improper Error Handling as Reaction to missing configuration Option

This issue was not detected while running FlowFuzz, rather than while implementing it. As of release 2.13, FlowDroid can be triggered to crash with a NullPointerException if the handed configuration file lacks a property.

What makes this issue interesting (and probably also is the reason why it has not been fixed before) is the fact that this missing property is not needed for the taint analysis but only for the generation of the output file. Therefore the crash only happens if the property is missing, FlowDroid finds at least one leak and is configured to write those findings to an output file.

The FlowDroid developers responded to this issue report and added nullness checks, but the issue can also by circumvented by simply updating the used configuration.

5.3 Minimization

For each bug candidate discovered by FlowFuzz, we measure how much minimization helped in reducing the length of the generated test cases.

For this we first look at the length of the code files before minimization and compare it to the length of the automatically minimized files and the minimal file we could manually create for each issue.

Bug candidate	Original generated	Automatic minimization	Manual minimization
FD #763	481 LOC	24 LOC (-95%)	15 LOC (-97%)
FD #764	327 LOC	25 LOC (-92%)	14 LOC (-96%)
FD #767	120 LOC	18 LOC (-85%)	11 LOC (-91%)
PY #923	451 LOC	10 LOC (-98%)	7 LOC (-98%)
MT #169 [2]	101 LOC	19 LOC (-81%)	12 LOC (-88%)
FD #781	72 LOC	14 LOC (-81%)	11 LOC (-85%)
MT #172	1814 LOC	21 LOC (-99%)	14 LOC (-99%)
MT #173	68 LOC	15 LOC (-78%)	11 LOC (-84%)
MT #174	528 LOC	15 LOC (-97%)	11 LOC (-98%)
PY #942 [3]	509 LOC	19 LOC (-96%)	7 LOC (-99%)
FD #735	383 LOC	334 LOC (-13%)	- [4]
FD #779	283 LOC	17 LOC (-94%)	9 LOC (-97%)

Table 5.2: Lines of code before and after minimization, not taking into account import statements and source/sink functions

The lengths of the generated test cases before and after minimization can be seen in table 5.2. The difference in length after automatic and manual minimization can mostly be attributed to syntactical merging of statements, for example combining variable declaration and assignment in one statement or replacing variables that are only used as constants with the constant value. Those operations are not carried out by FlowFuzz's minimization component, as it only removes statements as a whole.

The experimental results show that the minimization phase drastically reduces the test case size. They also confirm the assumption mentioned in section 4.4 that most test cases revealing relevant issues contain less than 45 lines of code, as all found issues except FlowDroid #735 can be represented in 26 LOC or less (also including import statements, class declaration and source/sink functions).

It also proves the effectiveness of the used minimization techniques, since the automatic minimization reduced at least 92% of all possibly minimizable statements in all shown cases, thus making it an essential part of our approach.

[2]The issue was dropped because it ultimately traced back to code optimization processes carried out by the Java compiler, but it can still serve for evaluating the effectiveness of the minimization process.

[3]The issue was dropped because it represents expected behavior under the used configuration, but it can still serve for evaluating the effectiveness of the minimization process.

[4]For this issue we were not able to generate a concise test case, as it seemed to be only triggered by long inputs. Therefore we were not able to clearly define which statements or combinations of statements lead to this issue.

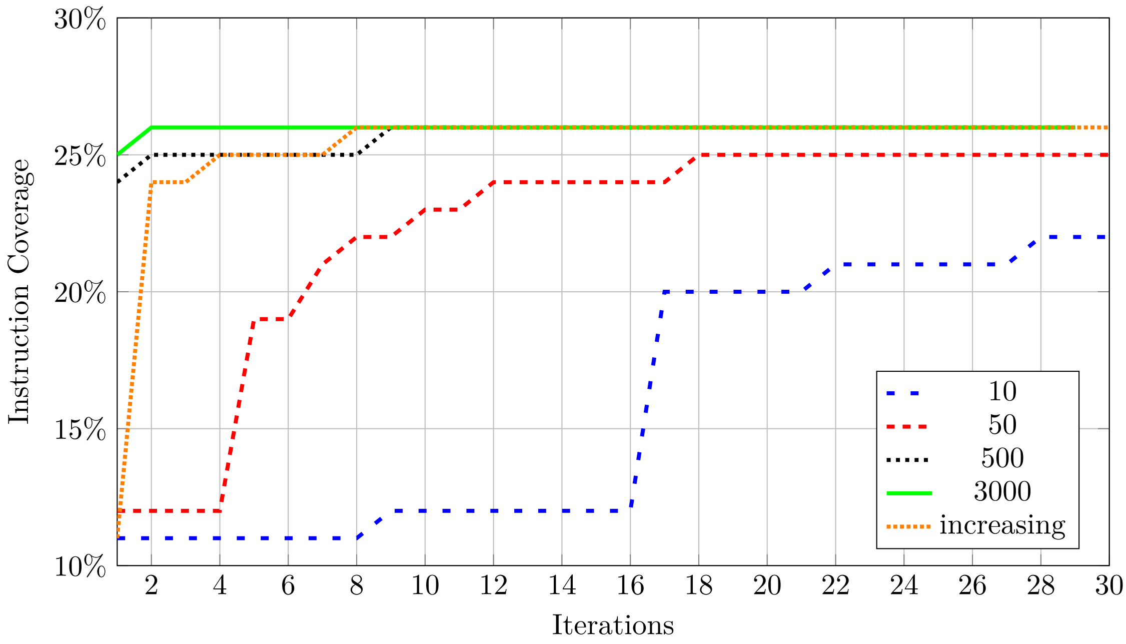

Figure 5.1: FlowFuzz instruction code coverage in FlowDroid, broken down by generated statements and iterations

5.4 Code Coverage Measurements

In order to measure the code coverage reached by FlowFuzz, we run experiments on FlowDroid using the Java code coverage library JaCoCo [JaC12].

The experimental setup includes running five containers with different configurations concerning the generated information flow. There are containers generating 10, 50, 500 and 3000 statements in each iteration as well as one container that starts with generating 10 statements in the first iteration and increases by 100 statements each iteration until it reaches more than 3000 statements.

As shown in figure 5.1 and figure 5.2, the maximum instruction coverage FlowFuzz reaches while testing FlowDroid is 26%, whereas the maximum branch coverage reached is 18%. The containers run for more than 500 iterations, but after a fast increase of the reached coverage in the first 25 iterations, it increases only minimally afterwards.

One interesting discovery is that the code coverage reached after around 25 iterations does not differ in the setups with 500 and 3000 statements generated each iteration. Only the containers generating 10 and 50 statements per iteration stay below the maximum reached by the other containers by achieving an instruction coverage of 23% and 25% and a branch coverage of 14% and 17% respectively.

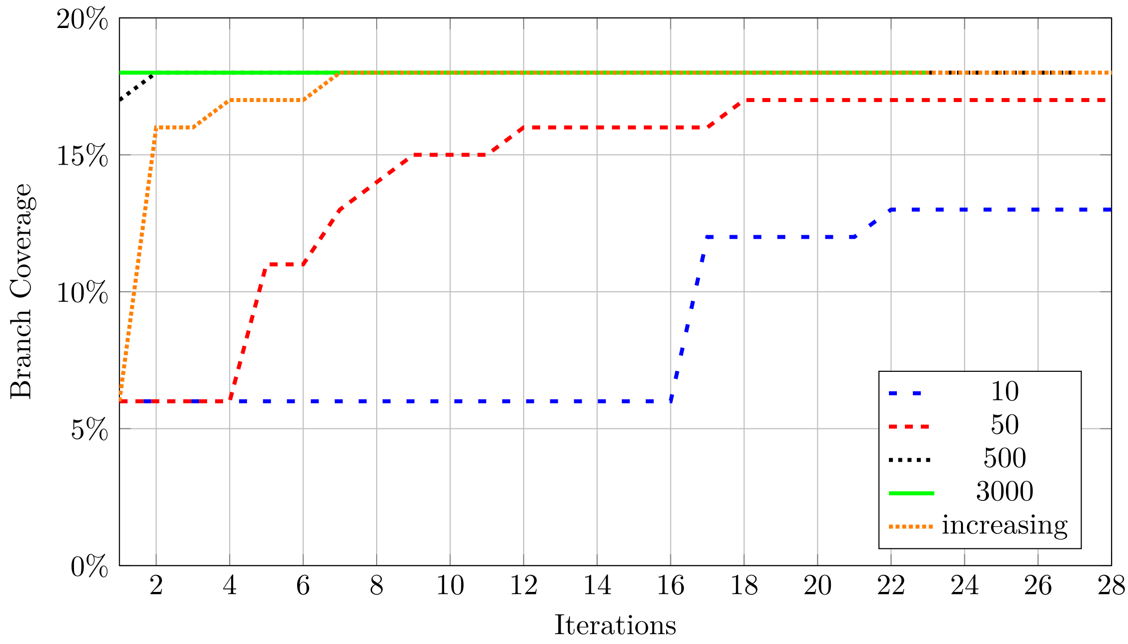

Figure 5.2: FlowFuzz branch code coverage in FlowDroid, broken down by generated statements and iterations

5.5 Runtime Evaluations

5.5.1 Runtime per component

In the context of the evaluation of the performance of FlowFuzz, we analyze the runtimes of the different stages of a differential test round. The results depicted in figure 5.3 clearly show that the execution of the tools under test consumes the largest share of the runtime of one iteration of differential testing, with significant runtime differences among the three tested tools. In comparison, the time for preparing the test cases (flow generation and translation as well as compilation of Java binaries) can almost be neglected, showing little overall performance improvement potential in the optimization of the flow generation component.

Another interesting finding is the fact that the runtime of the tools under test does not correlate to the size of the generated input files. At least for our specific setup, this insight shows that the biggest potential for performance improvement lies in the generation of bigger test cases, as they increase the probability of uncovering issues in the tools under test whilst having little effect on the overall runtime.

5.5.2 Runtime evaluation of found bug candidates

For each bug candidate discovered by a Docker container running FlowFuzz, we measure the time and iterations it took until this bug was discovered. An overview of some of the

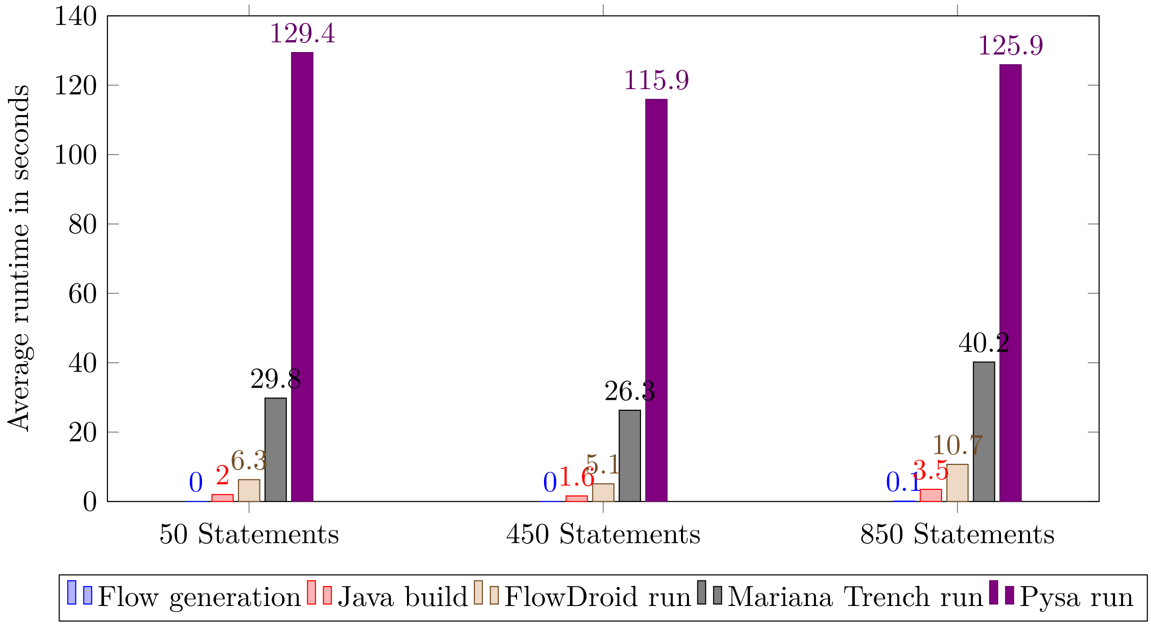

Figure 5.3: Runtime of different parts of an average differential testing cycle grouped by the size of the generated input

most interesting findings can be seen in table 5.3.

The table shows which of the tools described in section 5.1 were tested and how many statements were generated each round. As some containers were configured to run tests with increasing statement lengths, this column might contain a range. The table also shows how many iterations and how much time it took to discover the bug candidate and how much time the automatic minimization needed.

Some bug candidates have been found multiple times either sequentially in one container or by multiple containers. If one container found the same bug candidate multiple times, our analysis resets the time/iterations till bug after the first encounter to zero to facilitate comparability.

As shown in table 5.2, most found bug candidates were triggered by rather small input sets and thus should be detectable with a container generating only few statements. The smallest containers we used in our experiments generated 50 statements each iteration, amounting to around 70 lines of code.

But as we can see on the example of bug candidate **MT #173**, a container generating 50 statements at a time needed 186 iterations until the bug was triggered the first time, whilst a container generating 1300 statements at a time managed to find the bug in the first iteration. However, bigger input files also come with their own downsides. As the minimization time drastically increases with the number of statements generated, bugs found with larger input files consume more time in the minimization phase. In the case

Bug candidate	Statements	Tools	Till Bug		Minimization Time
			Iterations	Time	
FD #781	50	FD, PY, MT	45	01:13:16	01:40:48
	50	FD, PY, MT	68	02:08:22	01:28:34
	50	FD, PY, MT	63	01:55:37	01:25:02
MT #173	50	MT, PY	186	05:37:35	01:42:31
	300	MT, PY	132	03:33:31	02:02:08
	300	MT, PY	95	02:48:05	02:44:54
	1300	MT, PY	9	00:18:20	09:14:40
	1300	MT, PY	1	00:01:50	07:28:33
MT #172	901 to 1151	MT, PY	2574	79:17:16	10:55:04
	2703 to 2753	MT, PY	782	31:04:12	06:16:52
	2753 to 3153	MT, PY	3943	237:28:11	11:35:15
FD #735	50 to 400	FD, PY	19407	589:47:45	14:49:54
	2000	FD, PY	3	00:07:40	23:29:01

Table 5.3: Runtime of some containers finding a bug candidate. All times are in the format HH:MM:SS.

of **MT #173**, the fastest way to get a minimized version of the bug candidate was by generating 300 statements per iteration, which led to its discovery after 95 iterations and needed 05:32:59 including minimization (compared to 07:20:06 with 50 statements and 07:30:23 with 1300 statements).

Having said that, it is also important to consider that some other bug candidates needed large input sizes as well as many iterations to get discovered. If we look at bug candidate **MT #172** we can see that three containers found the issue but the fastest container still needed more than a day until discovery. This can be attributed to the complicated logic that is needed to trigger this issue, which also explains why no container with less than 900 generated statements was able to find the bug within reasonable time.

One bug candidate that reliably and reproducibly could be found with few generated statements and in short time is **FD #781**. This could be attributed to the rather simple logic needed to trigger this issue.

Another interesting result is the analysis of bug candidate **FD #735**. As we can see, a container with increasing code generation size needed almost 20.000 iterations and over 24 days until the issue was found whilst a container generating 2000 statements at a time only needed 3 iterations and 8 minutes to uncover the issue.

The analysis results show that the fastest way to find new bug candidates could be running multiple containers with different statement generation configurations in parallel, as some bug candidates can be found quickly with large input files whilst in other cases it pays off to run more iterations in exchange for shorter minimization times.

It also shows that the needed number of statements and runtime strongly differs from bug candidate to bug candidate and even within containers that found the same bug candidate using almost the same configuration. The vastly different runtimes needed to discover **MT #172** also stress the fact that measurements depicted in table 5.3 are mostly caused by sheer coincidence and come from the nondeterministic nature of FlowFuzz's data flow generation algorithm.

CHAPTER 6

Related Work

6.1 Differential Testing

Differential testing is a commonly used practice to uncover issues in software. McKeeman uses differential testing to find bugs in compilers and runtime systems [McK98]. He describes the advantages of differential testing as it partially solves the oracle problem and can be used to run millions of tests with little effort for programmers.

Cui et al. use differential testing to test transaction implementations in database management systems (DBMSs) [CDD+23]. They randomly generate a database and a group of concurrent transactions which are executed on multiple DBMSs in order to search for discrepancies.

Klinger et al. are employing differential testing in order to test the soundness and precision of program analyzers [KCW19]. They are comparing six state-of-the art analyzers on numerous automatically generated benchmarks, uncovering issues in most analyzers.

6.2 Code Generation

The automatic generation of programs as test cases for various tools is a well established and widely used approach. Kapus and Cadar are searching for bugs in symbolic execution engines using program generation and differential testing [KC17]. They are adapting techniques from the compiler testing area as presented by Yang et al. [YCER11] to test symbolic execution tools which keep multiple path prefixes in memory, as well as such that explore one path at a time.

Nakamura and Ishiura are generating test programs for random testing of C compilers [NI16]. Their method includes generating different test programs by repeatedly applying equivalence transformation to a trivial initial program. Bugariu and Müller are using a similar approach for generating test cases to test string solvers [BM20]. They start with input formulas that are satisfiable or unsatisfiable by construction and apply satisfiability-preserving transformations to generate more complex formulas.

Irfan et al. build the testing framework XDsmith to randomly generate programs using a subset of Dafny [IPR+22]. The generated programs have known verification outcomes and thus can be used to find bugs in the Dafny verifier and apply differential testing to the four different Dafny compilers.

6.3 Testing Static Analyzers

Due to their critical role in software development, testing static analyzers has become a prominent area of research. Kaindlstorfer et al. present a novel testing methodology called interrogation testing in order to find soundness and precision issues in program analyzers [KIWC24]. Their approach addresses limitations in existing metamorphic-testing techniques by utilizing analyzer queries to build more powerful oracles and introducing a knowledge base to track query histories.

Fleischmann et al. employ constraint-based test oracles extending the benchmarking tool FUZZLE [LKC22] in order to generate C programs from SMT constraints [FKI+24]. Based on the satisfiability of the constraint, their implementation derives whether the generated programs contain safety issues.

Cuoq et al. are testing the Frama-C static analysis platform [Tea08] using the random C-program generator Csmith [YCER11]. In the process, they did not only find bugs in Frama-C but also in Csmith itself [CMP+12].

Zhang et al. are testing static analyzers with their heuristic-based automated testing approach Statfier[ZPCT23]. Until now they found 79 bugs in the static analyzers PMD [Teab], SpotBugs [Teac], SonarQube [Son], CheckStyle [Teaa] and Infer [Fac].

Bugariu et al. are testing abstract domains which are the basis of all static analyzers based on abstract interpretation [BWCM18]. Using an automated technique to construct inputs, they test the soundness and precision of abstract domains through gray-box fuzzing.

6.4 Fuzz Testing

Fuzz testing is often used in software development, most often to uncover program crashes and memory errors. Zhou et al. are developing a tool to efficiently fuzz browser APIs [ZZW+22]. Their approach includes automatically parsing the browser's code base to build an API interference graph that is subsequently used to guide the test case generation in order to reduce redundancy and improve code coverage.

Tsankov et al. use fuzz testing for finding vulnerabilities in security protocols [TDB12]. By generating inputs that are not anticipated by the specifications, they try to force the systems under test into an insecure state.

Fowler et al. are employing fuzz testing to enhance automotive cyber security [FBSW18]. Their fuzzer is directly connected to the tested vehicles CAN bus in order to investigate the benefits of fuzz testing in vehicle system development processes.

CHAPTER 7

Discussion

7.1 Limitations

Whilst proving its efficacy by uncovering multiple issues, our work also shows limitations in the current implementation of FlowFuzz. One of the biggest problems is the runtime of the tools under test, which poses a big limiting factor on the amount of automatic test cases that are feasible to run in a given period of time.

Another limitation of our current implementation is the size of the supported instruction set for the generation of random information flow. The current focus on some language concepts might also hinder the detection of other bugs that could be found if the tools under test were given inputs containing other concepts that modern programming languages support. As our experimental evaluations of the code coverage of FlowDroid show, FlowFuzz still has potential for improvement.

Another limitation concerns the analysis results presented in this thesis, in particular the runtime evaluations. Whilst all data presented originate from our empirical measurements carried out during our experiments, the resulting time specifications might not be repeatable. On one hand, this is caused by the random nature of our test case generation process which does not generate repeatable results. On the other hand, the concrete setup of such experiments including the used hardware, the versions of the operation system, used libraries and the tools under test as well as FlowFuzz itself might heavily affect the measured results. However, in cases where we compare different results with one another, we always make sure that the compared metrics are obtained in similar environments.

7.2 Conclusion

In this thesis we presented FlowFuzz, a tool for differential testing of static taint analysis tools. We described the steps necessary to generate random program flow, translate it to source code and run the differential tests as well as the minimization process, if a bug candidate is found.

During our research we uncovered 11 bug candidates in the three static taint analysis tools FlowDroid [Ac18], Pysa [Sou17] and Mariana Trench [Sou20a]. Of these 11, seven already got confirmed by the respective developers. Four of the confirmed bug candidates have either been resolved or are scheduled for future fixes. In the process of bug finding, FlowFuzz also proved that it can serve as a useful tool to carry out regression testing, since two issues (FD #775 and FD #781) were introduced in newer versions of FlowDroid which got released during our research. The minimization component included in FlowFuzz also proved its effectiveness, as it removed a vast majority of unnecessary statements in all discovered test cases that exposed bug candidates.

The results motivate further research in this area, as FlowFuzz still only covers a small subset of all available programming language concepts. The addition of other language concepts as well as the refining of the input generation (e.g. by using program seeds to make special edge cases more likely to be triggered or by applying guided differential testing) could likely lead to the discovery of more issues.

Overview of Generative AI Tools Used

For writing this thesis, the following generative AI tools have been used:

1. **JetBrains AI Assistant** has been used for coding suggestions for FlowFuzz
2. **ChatGPT** has been used to
   a) generate Latex source code (mostly for the design of tables and figures)
   b) generate Python helper scripts that were used to analyze the logs of our containers
   c) find more fitting synonyms for certain words

The listed tools were only used for the listed purposes, no other tools have been used. The complete text body of this thesis is entirely typed by hand, no sentence was generated with AI tools.

List of Figures

List of Tables

List of Algorithms

Glossary

Precision Measures the proportion of true positives (TP) out of all flagged instances consisting of true positives (TP) and false positives (FP). High precision means fewer false positives, thus making the tool more practical and user-friendly. The precision can be calculated as $\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$. 25, 27, 30

Recall Measures the proportion of true positives (TP) out of all actual issues consisting of true positives (TP) and false negatives (FN). High recall indicates that the tool catches most of the true issues. The recall can be calculated as $\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$. 9, 25, 26, 28, 29

Bibliography

[Ac12] Steven Arzt and Soot contributors. Soot - a java optimization framework - github. https://github.com/soot-oss/soot, 2012. Accessed: 2024-10-23.

[Ac18] Steven Arzt and FlowDroid contributors. Flowdroid static data flow tracker - github. https://github.com/secure-software-engineering/FlowDroid, 2018. Accessed: 2024-10-23.

[ADS19] Abdullah Mujawib Alashjee, Salahaldeen Duraibi, and Jia Song. Dynamic taint analysis tools: A review. *International Journal of Computer Science and Security (IJCSS)*, 13(6):231–244, 2019.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, pages 259–269. ACM, 2014.

[Art24a] Maxime Arthaud. False positive with taint propagated via class attribute. https://github.com/facebook/mariana-trench/issues/173#issuecomment-2480916401, 2024. Accessed: 2024-11-21.

[Art24b] Maxime Arthaud. How to define taint propagation with lists. https://github.com/facebook/mariana-trench/issues/171#issuecomment-2355073569, 2024. Accessed: 2024-11-21.

[Art24c] Maxime Arthaud. Inconsistent results in loop. https://github.com/facebook/mariana-trench/issues/172#issuecomment-2480914483, 2024. Accessed: 2024-11-21.

[Art24d] Maxime Arthaud. Missed taint sink caused by aliasing due to shallow list copy. https://github.com/facebook/pyre-check/issues/923#issuecomment-2370572884, 2024. Accessed: 2024-10-27.

[Bar18] Gergö Barany. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction*, CC '18, page 82–92, New York, NY, USA, 2018. Association for Computing Machinery.

[BDR22] Jinsheng Ba, Gregory J Duck, and Abhik Roychoudhury. Efficient greybox fuzzing to detect memory errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[BHM+15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[BM20] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1459–1470, New York, NY, USA, 2020. Association for Computing Machinery.

[BWCM18] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically testing implementations of numerical abstract domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 768–778, New York, NY, USA, 2018. Association for Computing Machinery.

[CDD+23] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. Differentially testing database transactions for fun and profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[CMP+12] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, pages 120–125, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Fac] Facebook. Infer: A static analysis tool for java, c, c++, and objective-c. `https://github.com/facebook/infer`. Accessed: 2024-11-15.

[FBSW18] Daniel S. Fowler, Jeremy Bryans, Siraj Ahmed Shaikh, and Paul Wooderson. Fuzz testing for automotive cyber-security. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 239–246, 2018.

[FKI+24] Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. Constraint-based test oracles for program analyzers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 344–355, New York, NY, USA, 2024. ACM.

[IPR+22] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. Testing dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 556–567, New York, NY, USA, 2022. Association for Computing Machinery.

[JaC12] JaCoCo Contributors. JaCoCo: Java Code Coverage Library, 2012. Accessed: 2024-11-22.

[KC17] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 590–600, 2017.

[KCW19] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 239–250, New York, NY, USA, 2019. Association for Computing Machinery.

[KIWC24] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. Interrogation testing of program analyzers for soundness and precision issues. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 319–330, New York, NY, USA, 2024. ACM.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[LD14] Bhushan Lokhande and Sunita Dhavale. Overview of information flow tracking techniques based on taint analysis for android. In *2014 International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 749–753, 2014.

[LKC22] Haeun Lee, Soomin Kim, and Sang Kil Cha. Fuzzle: Making a puzzle for fuzzers. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, New York, NY, USA, 2022. ACM.

[McK98] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[MWME21] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael Ernst. Verifying determinism in sequential programs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 37–49, 2021.

[NI16] Kazuhiro Nakamura and Nagisa Ishiura. Random testing of c compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 676–679, 2016.

[RS10] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 186–199, 2010.

[SLZS16] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA, 2016. Association for Computing Machinery.

[Son] SonarSource. Sonarqube community edition. `https://www.sonarsource.com/open-source-editions/sonarqube-community-edition/`. Accessed: 2024-11-15.

[Sou17] Meta Open Source. Pyre: Performant type-checking for python. - github. `https://github.com/facebook/pyre-check`, 2017. Accessed: 2024-10-23.

[Sou20a] Meta Open Source. Mariana trench: A security focused static analysis tool for android and java applications. - github. `https://github.com/facebook/mariana-trench`, 2020. Accessed: 2024-10-23.

[Sou20b] Meta Open Source. Sapp: Post processor for facebook static analysis tools. - github. `https://github.com/facebook/sapp`, 2020. Accessed: 2024-10-23.

[Sou24a] Meta Open Source. Mariana trench: Getting started guide. `https://mariana-tren.ch/docs/getting-started/`, 2024. Accessed: 2024-10-23.

[Sou24b] Meta Open Source. Pysa basics. `https://pyre-check.org/docs/pysa-basics/`, 2024. Accessed: 2024-10-23.

[TDB12] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. Secfuzz: Fuzz-testing security protocols. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 1–7, 2012.

[Teaa] Checkstyle Team. Checkstyle: A tool for checking java source code for adherence to coding standards. `https://github.com/checkstyle/checkstyle`. Accessed: 2024-11-15.

[Teab] PMD Development Team. Pmd: A source code analyzer. `https://github.com/pmd/pmd`. Accessed: 2024-11-15.

[Teac] SpotBugs Team. Spotbugs: A static analysis tool for java. `https://spotbugs.github.io/`. Accessed: 2024-11-15.

[Tea08] Frama-C Team. Frama-c: A platform for static and dynamic analysis of c programs. `https://frama-c.com/`, 2008. Accessed: 2024-11-15.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[ZPCT23] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. Statfier: Automated testing of static analyzers via semantic-preserving program transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 237–249, New York, NY, USA, 2023. Association for Computing Machinery.

[ZZW+22] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. Minerva: browser api fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1135–1147, 2022.