# Explaining Static Analysis with Rule Graphs

Lisa Nguyen Quang Do, and Eric Bodden

**Abstract**—As static data-flow analysis becomes able to report increasingly complex bugs, using an evergrowing set of complex internal rules encoded into flow functions, the analysis tools themselves grow more and more complex. In result, for users to be able to effectively use those tools on specific codebases, they require special configurations—a task which in industry is typically performed by individual developers or dedicated teams.

To efficiently use and configure static analysis tools, developers need to build a certain understanding of the analysis' rules, i.e., how the underlying analyses interpret the analyzed code and their reasoning for reporting certain warnings. In this article, we explore how to assist developers in understanding the analysis' warnings, and finding weaknesses in the analysis' rules. To this end, we introduce the concept of *rule graphs* that expose to the developer selected information about the internal rules of data-flow analyses. We have implemented rule graphs on top of a taint analysis, and show how the graphs can support the abovementioned tasks.

Our user study and empirical evaluation show that using rule graphs helps developers understand analysis warnings more accurately than using simple warning traces, and that rule graphs can help developers identify causes for false positives in analysis rules.

**Index Terms**—Program analysis, Data-flow analysis, Rule graphs, Analysis configuration, Explainability, Usability

## 1 Introduction

To detect bugs and vulnerabilities in their code, more and more developers and companies use static analysis, a method of reasoning about the runtime behaviour of a program from its source code without executing it. In past years, researchers and practitioners have improved the capabilities of static analyses, enabling them to find increasingly complex bugs [1], [2], support more languages [3], and report more accurate warnings [4], [5] in a faster time [6], [7].

As analyses detect more complex warnings, reporting and explaining such warnings to the developer becomes more difficult. In practice, analyses are known to report many false positives. Some are due to over-approximations (e.g., for collections or arrays), others to missing knowledge about the particular codebase (e.g., specific libraries or coding constructs). An analysis interprets the source code and builds its own understanding of how it works. Sometimes, this understanding may not match the developer's, which results in uncertainties, a wrong treatment of critical warnings, wrong tool configurations, or even tool abandonment [8], [9].

As a result, analysis tools are typically configured by dedicated teams (or the developers themselves, depending on the company's resources) before they are deployed in a company. Such teams typically configure the analysis options and edit the analysis rules to customize them to particular code bases [8]. Analysis *rules* determine how the analysis reasons about the analyzed code. In the case of data-flow analysis, those rules are encoded as flow / transfer functions.

Past research in static-analysis usability highlights limitations in *explainability* of the analysis warnings and *customization* of the analysis rules [8]–[11]. To configure and use static analysis tools to their full potential, developers and configuration teams need to *understand* analysis warnings in order to determine which ones are true positives and which ones can reveal problems in the analysis rules that they can then *correct*. Developers have external knowledge about the code base that the analysis does not possess, and the contribution of such heuristics can help direct the analysis in yielding more accurate warnings.

The understanding task requires the developers to build a solid understanding of how the analysis rules work for their code base. Static analyses are typically used as black boxes, their warnings being post-processed using information that is *external* to the analysis rules to provide developers with more complete warnings (e.g., warning type, code location). In an effort to bridge the understandability gap, we instead propose to make use of *internal* information: how the analysis interprets the analyzed code. Focusing on data-flow analysis, we introduce the novel concept of *rule graphs* that encode internal analysis information, and explain how to use them to give developers more insight into the analysis' reasoning. In our evaluation, we show that the use of rule graphs can improve the developers' understanding of analysis warnings, and help them identify weak or missing analysis rules that can then be corrected.

This article makes the following contributions:

- It advocates for more transparency of the analysis' internal behaviour to address the explainability and customization problems of static analysis tools.
- It introduces the concept of rule graphs to expose the analysis' internal behaviour, and explains how to adapt it to any data-flow analysis solver.
- It details how to use rule graphs in the tasks of *understanding* warnings and *identifying weak* or *missing* analysis rules.
- It presents an implementation of the concept and tasks for taint analysis on Java and Android applications with an IntelliJ plugin: MUDARRI.
- With a user study on 22 participants, it shows that the use of rule graphs significantly improves warning

- *L. Nguyen Quang Do is with Paderborn University, Germany. E-mail: lisa.nguyen@upb.de*
- *E. Bodden is with Paderborn University and Fraunhofer IEM, Germany. E-mail: eric.bodden@uni-paderborn.de*

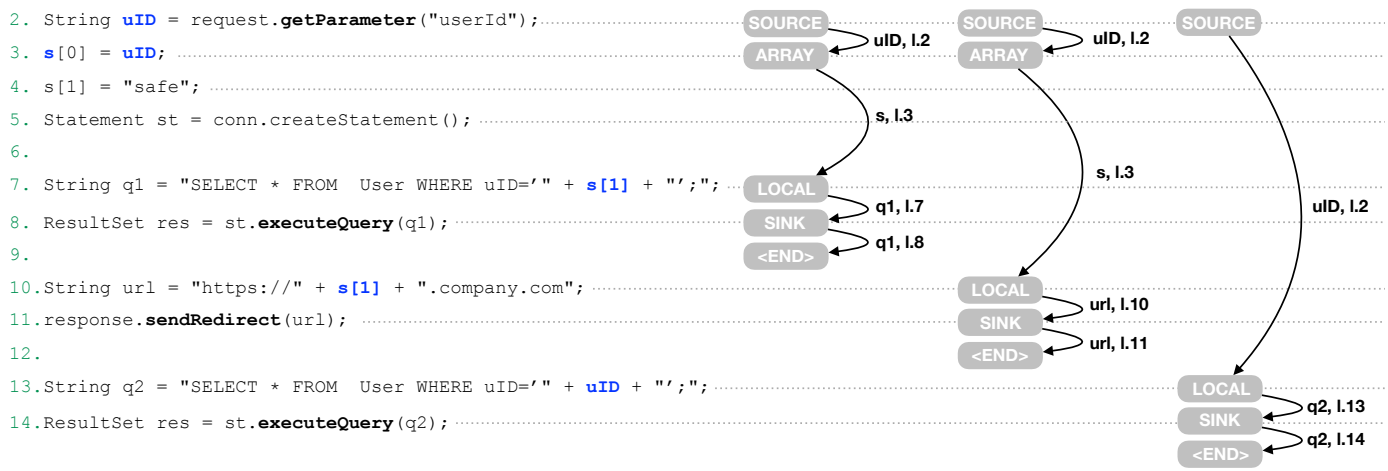



Figure 1: Potential SQL Injections (line 8, line 14) and Open Redirect (line 11), and their Simplified Rule Graphs. The first SQL injection and the open redirect are false positives.

understandability compared to traditional code traces that only report on the path of the warning.
- Through an empirical evaluation on DroidBench [2], it shows that rule graphs can help developers discover weak or missing analysis rules.

## 2 Motivating Example

In this section, we provide background on data-flow analysis and we show how exposing an analysis' internal rules can assist the developer in two different tasks: understanding analysis warnings and identifying weak or missing analysis rules. To illustrate them, we use the running example in Figure 1. The example contains an SQL injection [12] from the source `getParameter()` at line 2 to the sink `executeQuery()` at line 14. The SQL injection at line 8 and the open redirect [13] (line 11) do not occur, because `q1` and `url` contain the safe string "`safe`" from `s[1]`.

### 2.1 Background on Data-Flow Analysis

*Data-flow analysis* is a branch of static analysis in which data is tracked along the control-flow of the program through a Control Flow Graph (CFG) that models in which order the statements of the program are executed. In the example Figure 1, the CFG would be linear, from top to bottom, with each node being a statement (a line of code). SQL injections and open redirects can be detected using a *taint analysis*, a type of static data-flow analysis that tracks tainted data through a program from *source* methods (e.g., `getParameter()`) to *sink* methods (e.g., `executeQuery()` or `sendRedirect()`). In the example, a taint analysis would track potential user-controlled data starting from the source at line 2, and track the variables in which the data is stored until it reaches a sink. Then, it would report a potential warning.

The monotone framework [14] is often used to define and solve classical data-flow analysis problems. We use it in this article for our formalizations. In the monotone framework, the *analysis solver* flows *data-flow facts* along the CFG in a *fixed point iteration* as shown in Algorithm 2 (ignoring the $'$ signs and the $mi$ highlighted in gray). The analysis starts at the program's entry points (typically the `main` function) (line 2), and keeps a worklist of the statements it visits. For each statement, it applies the flow function $f_{stmt}$, which generates *data-flow facts* that it stores in memory (line 7). Those data-flow facts are used as the input to the flow function for the successor statements in the CFG, at the next iterations (line 10). If a statement has multiple predecessors, we use a *merge* function $\sqcap$ to merge the data-flow facts from all of the predecessors before applying the flow function (line 6). The iteration stops when a fixed point is reached, meaning that the data-flow facts do not change anymore (line 9). In the monotone framework, the fixed point iteration is guaranteed to eventually reach a fixed point and terminate.

The flow function $f_{stmt}(in)$ describes how, from an in-set of *data-flow facts* $in = (d_1, ..., d_n), d_i \in D$, and a statement of the program, we obtain an out-set of new data-flow facts $out = (d_1, ..., d_m), d_j \in D$. The flow function thus describes how the analysis interprets the statements of the program. It consists in a set of *analysis rules*. In the case of our taint analysis, $D$ is the set of variables of the program. The in-set corresponds to the set of tainted variables (variables that may contain malicious data) before the statement is executed, and the out-set corresponds to the set of tainted variables after the statement is executed. The analysis reports a warning when a tainted variable reaches a sink statement. Algorithm 1 (ignoring the parts highlighted in gray) shows a flow function for a taint analysis. It contains five analysis rules describing how to handle different types of statements: sources (line 6), sinks (line 15), assignments to arrays (line 11) and to local variables (line 13), and variables that are not affected by the statement (line 3).

Let us consider the source rule (line 6). The analysis detects a source if the statement is an assignment statement, and if the right side of the assignment is a known source (e.g., `x = getSecret();`). If this is the case, the left side of the statement must be tainted, as it now contains a potential secret. The flow function does this by generating

the corresponding data-flow fact `x` (line 7), and adding it to the out-set (line 8).

Similarly, if the statement is an assignment statement and the right side of the statement is already tainted (if it is in the in-set) (line 9), then the left side must also be tainted (line 12 and line 14). For example, for `x = y;`, if `y` contains a secret, then `x` will also contain it after the statement. In the case of a sink statement (e.g., `sendByEmail(x);`), if the statement is a known sink and if its parameter is tainted (line 15), we report a potential data leak (line 18), and we transfer the taint to the next statement (line 17). Note that all tainted variables that are not affected by the statement are transferred as well (line 3). E.g., at `x = y;`, if `z` was tainted before, it should still be tainted after.

Let us focus on the particular example of the assignment to an array (line 11). The flow function marks that if the statement is an assignment statement, if the right side of the assignment is tainted (line 9), and if the left side is an array (line 11), then the array should be tainted (line 12). This particular rule is an *over-approximation*: instead of tracking the individual elements, the entire array is considered tainted if one of its elements is, thus leading to the false positives down the line: the first SQL injection and the open redirect.

### 2.2 Understand a Warning

To configure or use an analysis, developers must be able to evaluate if the warnings it yields are correct, and if they are of interest to their particular situation. To do so, they have to gain a full understanding of the warnings, and in particular, of why the analysis thinks they are legitimate bugs. Current tools generally provide external information that can range from vulnerability descriptions to more complex data such as warning severity, detailed traces, exploit examples, or even fix suggestions.

An often overlooked aspect is that the analysis algorithm can be faulty or approximative, and can misinterpret certain parts of the code. For example, for scalability, static analyses often over-approximate certain constructs, such as arrays or collections. In Figure 1, such an analysis would consider all elements in `s` as dangerous after line 3, and mistakenly report the SQL injection line 8 and the open redirect.

Even for a developer who knows how the vulnerabilities occur, figuring out why the analysis reports those two false positives is not straightforward, because the array over-approximation is internal to the analysis and thus, completely hidden from the developer. Analysis shortcomings may come from mishandling varied coding concepts: collections, aliasing, multithreading, etc. We argue that making the analysis' internal rules more explicit to the developer might address this issue and improve the understandability of analysis results.

### 2.3 Identify Weak and Missing Analysis Patterns

Because of its necessity to over-approximate, static analysis is known to report too many false positives [8], [9]. To help developers differentiate between true and false positives, many analysis tools calculate a confidence metric (how confident the analysis is that a warning is a true positive) generally based on external features, such as the bug type (e.g., SQL injection) [15], [16].

False positives are mainly due to the analysis' weaknesses, i.e., to the analysis rules that do not interpret the analyzed code as it should. For example, arrays over-approximations such as described above may result in false positives. Over time, developers may deduce some of the analysis' weaknesses. Allowing them to integrate them in the analysis and to modify the analysis rules would help reduce the number of false warnings.

To hasten this process, analysis tools can assist developers discover which analysis rules—or rule combinations—are at fault, or are missing. We differentiate between *weak rules* and *missing rules*. The former correspond to existing analysis rules that create false positives, e.g., over-approximations such as the write-access to the array at line 3 that mark the warnings line 8 and line 11 as rather likely false positives in Figure 1. The latter correspond to rules that are not present to in the analysis, and which absence also causes false positives. In this article, we argue that from a set of known true and false positives, we can help developers identify weak or missing analysis rules.

## 3 RULE GRAPHS

To address the tasks presented in Section 2, we introduce the concept of *rule graphs* that exposes the analysis' internal information to the end-user, and show how to modify existing data-flow analyses to support rule graphs.

### 3.1 Running Example and Definition

Let us consider the running example of Figure 1 and a *taint analysis* that over-approximates array elements to the entire array. The analysis would report the two SQL injections and the open redirect by tracking the variables containing the tainted data from the source `getParameter()` to the sinks `executeQuery()` and `sendRedirect()`. Figure 1 shows the three rule graphs representing each of the analysis warnings. For example, following the edges of the second graph (open redirect), we see that the data is first assigned to `uID` at line 2, then to `s` at line 3 (because of the over-approximation of `s[0]` to `s`), and finally to `url` at line 10 (over-approximation of `s[1]` to `s`) before being reported at line 11. In addition to the code-related information found in the edge labels, the nodes encode the internal analysis rules. For the open redirect, the root node is the source, creating the taint to `uID`. The taint is then assigned to `s` because it is an array, and then transferred to the local variable `url` by a part of the analysis which handles locals and is different from the one which handles array assignments. Sinks are handled separately, which is represented by the SINK node. The warning stops at an artificial node <END>, introduced so that its edge from the sink stores the last step of the trace.

The nodes of the rule graphs mark the different rules of the analysis that handle the different constructs of the analyzed code, as highlighted in gray in the flow function of our taint analysis, in Algorithm 1. For a given statement `s` and a set of variables tainted before the statement `in`, it generates, transfers, and kills taints, yielding an updated set of tainted variables after the statement: `out`. The five analysis rules described in Section 2.1 are encoded in this flow function: the generation of taints when a source is

**Algorithm 1** Flow function for a taint analysis for Figure 1, with an over-approximation of arrays at line 11. The marker information is shown in gray.

```
1:  procedure f_stmt(⟨in⟩)
2:      out := ∅
3:      for d in in do
4:          d' := newDataFlowFact(d, stmt)
5:          out = out ∪ (d', ({d}, ID))
6:      if isAssignstmt(stmt) ∧ rightIsSource(stmt) then
7:          d' := newDataFlowFact(leftSide(stmt), stmt)
8:          out = out ∪ (d', ({⟨ZERO⟩}, SOURCE))
9:      if isAssignstmt(stmt) ∧ (rightSide(stmt) ∩ in ≠ ∅) then
10:         d' := newDataFlowFact(leftSide(stmt), stmt)
11:         if leftIsArray(stmt) then
12:             out = out ∪ (d', ({d}, ARRAY))
13:         else
14:             out = out ∪ (d', ({d}, LOCAL))
15:     if isSink(stmt) ∧ sinkParameter(stmt) ∈ in then
16:         d' := newDataFlowFact(sinkParameter(stmt), stmt)
17:         out = out ∪ (d', ({d}, SINK))
18:         report(d')
19:     return out
```

detected (denoted with the SOURCE marker, line 6), the taint transfers for arrays (ARRAY, line 11) and local variables (LOCAL, line 13), the taint transfer and reporting at sinks (SINK, line 15), and the transfer of existing taints (ID, line 3). Graph nodes thus correspond to *rule markers* and give insight into *how* the analysis works. We have removed the ID nodes from Figure 1 to simplify the example.

We define a rule graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as a set of nodes $\mathcal{V}$ (the set of markers in the analysis) and edges $\mathcal{E}$, which labels contains code locations.

**Size of the rule graphs:** Depending on which rules the developers wish to track, different markers can be chosen. As a result, rule graphs can approximate a traditional warning trace, or be vastly different. However, they remain at worst in the size range of a traditional trace ($\mathcal{O}(tm)$ with $t$ the size of the trace, and $m$ the number of markers).

When reasoning about analysis rules without taking into account the code base (e.g., for determining weak analysis rules), code-specific information is not needed. As a result, the edge labels can be dropped, making the rule graphs much smaller and simpler to handle. In the example Figure 1, the first two graphs would then become identical.

### 3.2 Generating Rule Graphs

We propose a general recipe to modify an existing data-flow analysis expressed in the monotone framework [14] in order to support rule graphs. This is done in two phases: as the analysis runs, it stores rule marker information, and after it terminates, it uses it to extract the individual rule graphs.

#### 3.2.1 Collecting Rule Marker Information

Algorithm 2 presents the traditional fixed-point iteration algorithm of data-flow analyses that applies the flow function $f_stmt$ to the statements of the program until the out-

**Algorithm 2** Fixed-Point Algorithm for a Data-Flow Analysis. The modifications to support rule graphs are shown in gray.

```
1:  procedure ANALYZE
2:      wl := entrypoints()
3:      while wl ≠ ∅ do
4:          pop stmt off wl
5:          OLD := { d ∈ OUT[stmt] }
6:          IN[stmt] := ⊓'{ (d, mi) ∈ OUT[r] | r ∈ pred(stmt) }
7:          OUT[stmt] := f'_stmt(IN[stmt])
8:          NEW := { d ∈ OUT[stmt] }
9:          if OLD ≠ NEW then
10:             wl ∪ = successors(stmt)
```

$f'_{stmt}$ is a modified flow function, defined in Figure 2.
$\sqcap'$ is the merge function, adapted to handle marker information.

$$
\begin{aligned}
f'_{stmt}(in) &= f'_{stmt}(\{(d_1, mi_1), ..., (d_n, mi_n)\}) \\
&= \{(e_1, ni_1), ..., (e_m, ni_m)\}
\end{aligned}
$$

with $d_i \in D$,
$e_i \in D$ such that $f_{stmt}(d_1, ..., d_n) = \{e_1, ..., e_m\}$,
$f_{stmt}$ the original flow function,
$mi_i = (D_i, rule\_marker_i)$ such that $D_i \subseteq D$,
$ni_i = (D_i, rule\_marker_i)$ such that $D_i \subseteq \{d_1, ..., d_n\}$.

Figure 2: The Modified Flow Function.

sets stabilize, and the modifications made to support rule graphs. The main change is the introduction of rule marker and predecessor information, highlighted in gray. As shown in Figure 2, the modified flow function does not only report data-flow facts (tainted variables in terms of taint analysis) in its out-set: it encapsulates each data-flow fact with *marker information* ($mi$) containing the data-flow fact's predecessors and a rule marker explaining the reason why the data-flow information was transferred from the predecessors to the current fact. For example, in Algorithm 1, the rule at line 11 states that if the right side of an assignment statement is tainted (if the variable is in the $in$ set), the array on the left side of the assignment should be tainted. As a result, at line 12, $d'$, the data-flow fact representing the left side of the assignment is marked with its predecessor $d$ and the rule marker ARRAY. This notation means that the taint from $d$ is transferred to $d'$ at statement $s$ because of an assignment to an array. The merge operator $\sqcap$ (line 6) should also be adapted to merge marker information along with the original data-flow facts.

**Soundness and termination:** The modifications added to the analysis do not affect its termination and soundness since the marker information piggybacks on the data-flow facts without influencing them. Thus, Algorithm 2 is as sound as the original analysis and terminates in as many iterations.

**Algorithm 3** Graph Reconstruction.

```
1: procedure WALKBACK(⟨df, next_rule_marker, a⟩)
2:     if stoppingCondition() then
3:         pruneInfeasiblePaths(a)
4:         return
5:     s := getStatement(df)
6:     (D′, rule_marker) := mi ∈ OUT[s] | d = df
7:     for pred in D′ do
8:         a.addEdge(rule_marker, next_rule_marker, df)
9:         walkBack(pred, rule_marker, a)
```

#### 3.2.2 Extracting the Graphs

Once the marker information is computed, we run Algorithm 3 for each warning, to retrieve its rule graph. The algorithm implements a modified depth-first search algorithm (DFS). It recursively reconstructs the graph from the <END> node to the different sources, using predecessor information for the edges, and rule markers for the nodes. The stopping condition (line 2) holds if a source is reached when invalid paths are visited (where returns do not match calls), or when the DFS runs into a loop (causing unnecessarily complex patterns). In the latter cases, we then clean up the graph, removing data from the invalid path.

**Soundness and termination:** With a DFS, the algorithm steps back into all possible traces tracked by the marker information, including invalid ones. However, as long as the analysis' merge operator keeps the correct predecessors in the marker information, the DFS finds all correct traces, along with potential false positives. Since the number of generated data-flow facts is finite and the stopping condition keeps track of loops, the algorithm must terminate. Retrieving all paths between two nodes with a DFS has a complexity of $\mathcal{O}(se)$ with $s$ the number of nodes and $e$ the number of edges, so we apply optimizations to reduce the number of explored paths, which we detail in Section 5.1.

### 3.3 Requirements

We now summarize the requirements for implementing rule graphs in an existing data-flow analysis:

- The base analysis must terminate.
- The analysis solver must be modified as shown in Algorithm 2 to piggyback marker information on top of the data-flow facts.
- The analysis must run the graph extraction pass for each warning, as shown in Algorithm 3.
- The flow-functions must be annotated with the markers chosen by the developers, as shown in Figure 2, with Algorithm 1 as an example for a taint analysis.

Note that the instrumentation time for collecting rule marker information and extracting rule graphs is negligible next to the analysis itself: loading all classes, running the analysis, and reconstructing the warning path are known bottlenecks of the analysis process. Implementing the rule graphs does not add more time complexity: it does not modify the existing algorithms. It just piggybacks data on top of the base data-flow facts when running the analysis (collecting rule marker information on top of the fixed point algorithm) and extracting a witness for the warning (extracting the rule graph on top of the DFS).

## 4 Using Rule Graphs

In this section, we discuss how to use rule graphs to address the tasks presented in Section 2.

### 4.1 Understand a Warning

Traditional static analysis tools provide their users with warnings traces, i.e., a path from the source(s) to the sink(s) [2], [17], [18]. While useful to track where the sensitive data flows, such traces do not provide information on the analysis' reasoning. For example in Figure 1, at line 3, if the developer does not know about array over-approximations, it would be difficult for them to understand why `s` is tainted.

The rule information contained in the rule graphs can be used to enhance the explainability of warning traces. In the example, the ARRAY rule is contained in the warnings' rule graph, and can thus be used to help the developer to understand how the analysis handles arrays. Rule graphs thus enable the analysis tool to thus provide a more detailed analysis trace from the source to the sink that explains the analysis' reasoning step by step. Each edge `e` represents a step of the trace from the point of view of the analysis, and gives access to the following step information:

- The data-flow fact of interest: in `e`'s label.
- Why it is of interest to the analysis: `e`'s origin node.
- The location of the step in the code: in `e`'s label.
- The next step: the edges departing from `e`'s destination node.
- The previous step: the edges arriving at `e`'s origin node.

Looking up the first three points has a complexity of $\mathcal{O}(1)$. The lookup of the last two points has a worst-case complexity of $\mathcal{O}(n)$, with $n$ the number of edges in the graph.

For the example in Figure 1, the enhanced information about the open redirect can be read from the middle graph and reported to the developer as follows:

- **l.2**: SOURCE statement: `uID` is tainted.
- **l.3**: Assignment to an ARRAY: `s` is tainted from `uID`.
- **l.10**: Assignment to a LOCAL: `url` is tainted from `s`.
- **l.11**: SINK statement: `url` is reported.

Unlike traditional traces, this gives developers insight into the inner-workings of the analysis and allows them to pinpoint points of uncertainty, such as the ARRAY rule in the example above: looking at the code, we see that `s` should not be tainted from `uID`, which the analysis does. Note that the traces in Figure 1, the graphs approximate traditional traces, but more targeted sets of markers can be used, for example reporting only on the SOURCE, SINK, and ARRAY markers if LOCAL is of no interest.

### 4.2 Identify Weak Analysis Patterns

In the previous section, we have shown how to use rule markers to expose the analysis' internal rules to the developer, allowing them to pinpoint their shortcomings. We can take that a step further and semi-automate the detection of *rule patterns* that can lead to wrong results, for example, the array over-approximation in Figure 1.

Given a *training set* of labeled true and false positives, the corresponding rule graphs can be used to learn the most likely causes for the false positives and thus detect weak

```
1  try {
2    setImeiAsSourceAndCreateRuntimeException();
3    sms.sendTextMessage(num, null, imei, null,
         null);
4  } catch (RuntimeException ex) { ... }
```

Listing 1: No Data Leak: the Program Jumps into the Catch before Reaching the Sink

analysis patterns. Those weak patterns are combinations of analysis rules that lead to false positives. In terms of rule graphs, this translates to the presence or absence of subsets of edges (or nodes) in the rule graph indicating its likelihood of being a false positive. In the example, the first SQL injection and the open redirect are false positives, while the second SQL injection is a true positive. Comparing the graphs of the false positives (the first two graphs in Figure 1) and the graph of the true positive (rightmost graph in Figure 1), we see that the ARRAY rule is the problematic one.

Weak patterns are retrieved from a machine-learning classifier, initialized with a training set labeled with the classes *false positive* and *true positive*. The features used to characterize a rule graphs are the presence or absence of its nodes or unlabeled edges (combination of two nodes), in the form of a boolean vector. Once the classifier learns which combinations of features are more likely to lead to false positives, we use its decision rules to reveal the analysis' weak patterns. This method requires the classifier to be a rule-based classifier, such as a decision tree.

Following the example of Figure 1, a rule-based classifier would learn that rule graphs containing the node ARRAY are more likely to be false positives than others without. We can report that rule to the user, bringing their attention to the weak ARRAY rule, and showing that the way arrays are handled in the flow function is not entirely correct. In the case where the classifier would use the rule graphs' unlabeled edges rather than the nodes, it would report any of the following edges as potential indicators for false positives: SOURCE $\rightarrow$ ARRAY, ARRAY $\rightarrow$ LOCAL, also indicating the ARRAY rule as problematic.

### 4.3 Identify Missing Analysis Patterns

As mentioned in Section 2, another cause for false positives in an analysis is missing analysis rules. For example, let us consider Listing 1. In this code snippet, let us assume that the analysis does not handle try/catch constructs. It would report a data leak from the source line 2 to the sink line 3. This is a false positive, since the sink is never reached. The analysis is missing a rule about handling try/catch constructs. Missing rules are difficult to identify automatically, since there are no rules to point towards in the first place. In that case, it is more efficient to show the developer inconsistencies in the treatment of two warnings, so that they can manually determine whether or not the analysis behaves as expected.

Rule graphs can be used to pick good candidates to show the developer. If two rule graphs are similar (showing that the analysis handles the corresponding warnings similarly), but one is a false positive and one is a true positive, the analysis could be missing rules to properly handle the false positive, and the difference between the two warnings may indicate what that missing rule could be. For example, let us assume that in Figure 1 line 7 refers to `s[0]` instead of `s[1]`, making the first SQL injection a true positive. Ignoring the edge labels, the two first rule graphs (first SQL injection and open redirect) are identical, yet one is a true positive and one, a false positive. This can indicate a missing rule: here, for managing individual array elements. Reporting the warning pair to the developer can thus help them identify the missing rule.

To calculate the similarity between two warnings, we define the following *similarity coefficient* based on the number of edges their rule graphs have in common. The greater the coefficient, the more similar the warnings.

$$similarity(\mathcal{G}_1, \mathcal{G}_2) = \frac{1}{2} \cdot \left( \frac{\#(e_{\mathcal{G}_1} \cap e_{\mathcal{G}_2})}{\#e_{\mathcal{G}_1} + \#e_{\mathcal{G}_2}} + \frac{\#(el_{\mathcal{G}_1} \cap el_{\mathcal{G}_2})}{\#el_{\mathcal{G}_1} + \#el_{\mathcal{G}_2}} \right)$$

with $\mathcal{G}_i$ the warning graph, $e_{\mathcal{G}}$ the set of unique edges in $\mathcal{G}$ ignoring the edge labels, and $el_{\mathcal{G}}$ the set of unique edges in $\mathcal{G}$ taking the edge labels into account. The first part of the equation encodes the analysis' treatment of the warning regardless of the particular lines of code (edge labels). The second part is used when two warnings are semantically similar: the coefficient differentiates them by code location, using the edge labels. The $1/2$ normalizes the coefficient to a maximum value of 1.

Applied to Figure 1, we see that the warnings that are most similar are the SQL injection line 8 and the open redirect, with a similarity coefficient of 0.75 (all eight edges in common, four labeled edges in common), compared to 0.14 for the other two relationships (two of seven edges in common, no labeled edges in common).

## 5 Implementation

We now detail the implementation of the rule graphs and of the three modules that use them. For warning understandability, we present MUDARRI, an IntelliJ plugin that displays warning traces augmented with marker information. For the identification of weak analysis patterns, we show a machine-learning based pattern detection module. A last module for similarity computations is used for the identification of missing analysis rules. The source code is available online [19].

### 5.1 Rule Graphs

We have implemented the rule graphs on top of a taint analysis for Java and Android applications in the Soot-based [20], [21] IFDS [22] solver HEROS [23]. To compute the rule marker information (Section 3.2.1), we use a modified version of the path-reconstruction approach from FlowTwist [24]: we encapsulate the data-flow facts with additional information containing the original source statement of the current data-flow fact, the data-flow fact's predecessors and their corresponding rule markers, and the data-flow fact's neighbors, used for trace reconstruction. We also adapt the analysis solver to propagate data-flow facts along neighbors.

Our analysis' flow functions span 650 LOC, which contain a total of 39 rule markers. Thirteen markers are attached

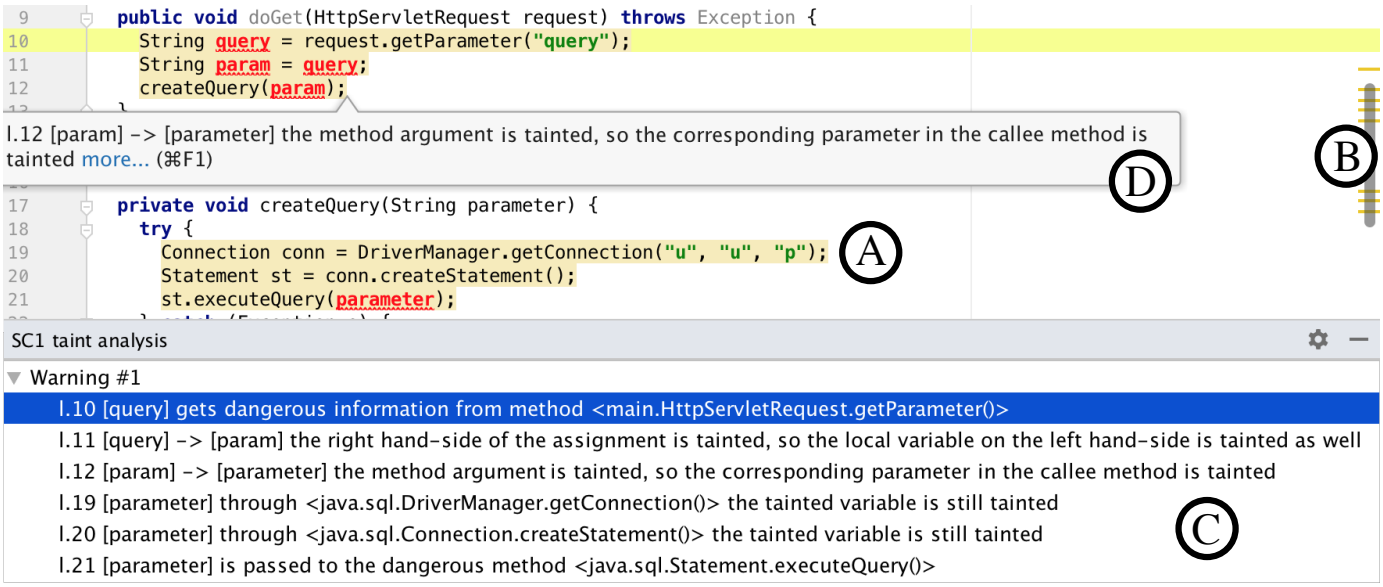



Figure 3: GUI of MUDARRI.

to analysis rules specific to calls to various library APIs, eight mark rules about taint transfers at call and return sites, six mark rules that describe the different types of taint transfer at assignment statements (e.g., LOCAL, ARRAY, etc.), three mark rules for sources and sinks, eight cover rules for manipulating aliases, and one is a marker used for a custom rule in the trace lookup algorithm.

For scalability, we limit the graph extraction (Section 3.2.2) to one trace per warning. We also reduce the number of visited paths by checking for loops and invalid paths (i.e., paths with non-matching call stacks), and by using FlowTwist's neighbor system. The sets of nodes and edges are stored in hashsets, allowing for a constant lookup complexity. The rule markers consist in a simple description string, and the edge labels contain the fully qualified name of the analyzed statement (i.e. including the statement, its line number, and its class name), to avoid confusions across multiple files.

### 5.2 Rule Graphs in Practice

The main overhead for using rule graphs is to define which rules to set in the flow functions (as shown in Algorithm 1), and to assign a human-understandable explanation to show the developer at runtime (as illustrated in Section 4.1). Depending on the use case and the codebase, different sets of markers can be used to pinpoint specific usages of certain analysis rules and surface them to the developer. In practice, this setup can be done once, when the tool is configured initially.

To provide more or less granularity in the warning explanations, it is not necessary to generate exactly one level of markers per call to the flow function, as shown in Section 3.2.1. When rules are not of interest, there is no need to mark them. When more detailed explanations are required, it is possible to generate a chain of data-flow facts in the same flow function. For example, if a variable is tainted at a source and that variable is already aliased to another variable, a chain of two data-flow facts can be produced at once: SOURCE and ALIAS. In our taint analysis, the longest chain in our analysis is of length 4.

The granularity of the markers is an important factor. In case where the user wishes to have as much data as possible, they can mark every line of the flow functions, providing simple explanations for each individual marker. This would result in multiple markers being assigned sequentially to the same line of code. The graph would then contain a sequence of multiple simple explanations for a single line of code, making the global explanation easier to understand, but also more verbose. In the case of proprietary information or code parts where the analysis cannot give details (e.g., native code, closed-source libraries, etc.), the user can annotate this case with a specific marker (e.g., NATIVE_CODE) and assign a description of how the analysis is supposed to behave there.

### 5.3 Graphical User Interface

The taint analysis can run as a standalone command-line tool, but we have also integrated it in an IntelliJ plugin [25] named MUDARRI. Figure 3 illustrates MUDARRI's Graphical User Interface (GUI). The example code contains an SQL injection highlighted in the editor (Ⓐ) and in the right gutter (Ⓑ). The bottom view (Ⓒ) details the warnings found in the application and each step of the warnings. For each step of a warning, the plugin displays the line of code, the tainted variable(s), and an explanation on the corresponding rule marker. For example, the first line in the view shows that the variable `query` is tainted because of the source method `getParameter`, and the second line explains that the taint is transferred from `query` to `param` because the tainted `query` is assigned to `param`. Selecting a warning in the bottom view highlights the corresponding lines of code in the editor and marks the tainted variables in a bold red font. Those details also appear in a tooltip (Ⓓ) when hovering over a highlighted line of code. The full

warning can be seen in (Ⓒ). Double-clicking on a step in the bottom view opens the corresponding file in the editor.

MUDARRI thus aims at improving the explainability of the analysis warnings by explaining how they are detected in the code, and by visualizing them in the interface.

### 5.4 Offline Functionalities

From a set of warnings marked as true or false positives, the pattern detection module pre-processes the warnings' rule graphs and extracts their nodes. It then passes them to a rule-based machine-learning classifier, and retrieves its rules, as shown in Section 4.2. The machine learning module is implemented using the WEKA machine learning framework [26]. Both graph nodes and edges can be used as learning features, and their presence or absence in a particular rule graph provides binary inputs to help classify the graph. Since the pattern detection module depends on machine learning, it runs into the scalability issues known to that domain [27]. As a result, a full learning/classification cycle cannot be used at runtime in the developer's Integrated Development Environment (IDE). Instead, we run them offline, as part of post-processing modules after the analysis terminates.

Finally, the similarity computations module pairs warnings that have opposite labels (i.e., a false positive and a true positive), and yields a list of similar warnings to explore, allowing the user to identify missing analysis rules, as described in Section 4.3. Pairs are returned in decreasing similarity order, so that the pairs that have the highest chances of indicating missing analysis rules are shown first.

## 6 EVALUATION

To evaluate the use of rule graphs for the three tasks described in Section 4, we ask the following research questions:

**RQ1:** Can rule graphs help understand warnings?
**RQ2:** Can rule graphs help find weak patterns?
**RQ3:** Can rule graphs help find missing patterns?

We ran our experiments on a 64-bit 10.13 Mac OS X laptop with an Intel Core i5 2,9 GHz CPU running Java 1.8, with a Java heap space of 8 GB.

### 6.1 User Study (RQ1)

Through a user study, we evaluate how internal analysis information assists the developer in understanding warnings. We compare MUDARRI to TRACE, a version of MUDARRI that does not contain marker information (illustrated in Figure 4). The two tools are visually identical, except for the details of the trace in the bottom view (Ⓒ in Figure 3): TRACE does contains the tainted variables and the lines of code, but the rule marker information is replaced by the Java statement at that line of code, showing a traditional trace.

#### 6.1.1 Experimental Setup

We ran a comparative, within-subjects user study between MUDARRI and TRACE with 22 participants, referred to as **P1–P22** (13 students, 9 researchers). Computer science students are considered reasonable proxies for developers in software engineering studies [28]–[31]. Of all participants,

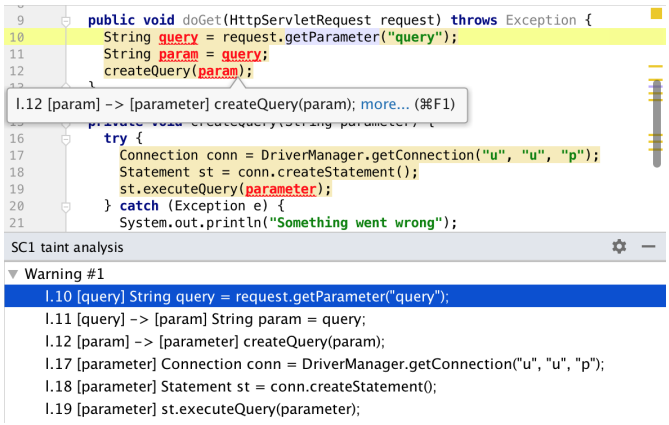



Figure 4: GUI of TRACE, for the same code example as for Figure 3. Similarly to many current analysis tools the explanations are not generated from the markers, but are code snippets.

13.6% have between one and two years of experience as professional software developers, 9.1% of the have more than five years of experience, 22.7% have between three and five years of experience, 13.6% have between two and three years of experience, and 40.9%, a year or under. We refer to "years of experience" professional, remunerated experience.

The participants were given a 15-minutes task: to go through a list of data leaks reported by the taint analysis, decide if they are true positives or false positives, and explain their reasoning. We did not measure the correctness of their choice, but whether or not they understood the warning, i.e., if they could name the correct source and sink, which variables were tainted between them, and how the taints were transferred at each statement. This is the typical reasoning of a developer when dealing with taint analysis warnings. We consider this the least subjective way of evaluating if a participant understood a warning.

The participants performed the task twice: once with MUDARRI, and once with TRACE. A latin-square design was used to counter the learning effects: half of the participants used MUDARRI first, the other half used TRACE first. The tools were named "Tool 1" and "Tool 2", in the order in which the participants were introduced to them. For the two tasks, we used two real-world Android applications from F-Droid: Balance [32] and Sparkleshare [33] containing respectively 8 and 16 data leaks over their respective 1,000 and 1,700 LOC. All participants performed the first task with Balance and the second with Sparkleshare.

During the tasks, we asked the participants to think aloud, allowing us to determine which warnings they understood correctly. We also obtained information about the features of MUDARRI and TRACE that were perceived as most or least useful. After each task, the participants also graded the the tool they used on a Likert scale from 0 to 10, for a Net Promoter Score (NPS) [34].

#### 6.1.2 Results

Figure 5 presents the percentage of participants who could correctly explain each warning. For the first task, we see that with MUDARRI, participants understood Balance's warnings

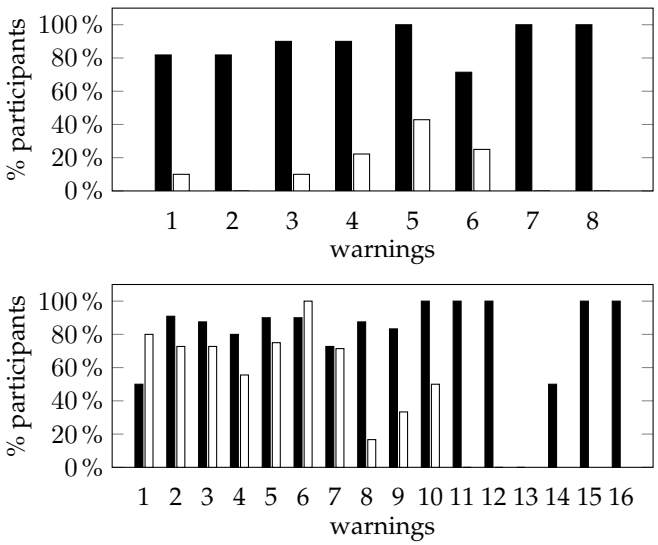



Figure 5: Percentage of Participants Who Correctly Understood the 8 Warnings of Balance (top) and the 16 Warnings of Sparkleshare (bottom) with MUDARRI (black) and TRACE (white).

better than with TRACE: on average, 86.67% of the participants understood the warnings correctly using MUDARRI, but only 14.81% using TRACE. This is mostly due to MUDARRI's clarification of the source and sink methods (**P12** "[With TRACE ] I don't know if it is a true sink") and the taint transfers in cases of complex code constructs. For example, the first warning in Balance had an Intent constructor as a source, which was not explicitly stated in TRACE. Since the constructor does not look like a typical source method, most participants overlooked it and misidentified the source for another method. MUDARRI explicitly names the source method, allowing participants to easily identify it. In another example, a wrong analysis rule mistakenly transferred a taint in a listener object. Most MUDARRI users immediately identified the problem, because MUDARRI reported a taint transfer due to an assignment on a line with no assignment. TRACE users spent more time speculating on how the taint entered the listener. We thus see that rule graphs allows the tool to show the developer an easily understandable message that explains how the analysis reasons. With this message, they can then determine whether the reasoning is sound or not, for each statement of the code, step by step.

For the second task, the warnings of Sparkleshare were of two kinds: warnings 1 to 7 were similar to a few warnings already seen in Balance, and warnings 8 to 16 were new types of warnings. We see that TRACE users perform much better on warnings they are familiar with (74.60%) than not (26.67%), while MUDARRI users perform equally well over both groups (respectively 82.81% and 82.61%). With new warnings, we again see the general trend of MUDARRI helping participants understand warnings better than TRACE. For warnings 1 to 7, the similar performance of both participant groups illustrates a learning effect: TRACE users had used MUDARRI for their first task, and the knowledge gained then allowed them to perform almost as well as MUDARRI users for warnings that they had already seen before (**P8** "If I didn't [already] know what it was, I could not find out what it is about"). With new types of warnings, their performance decreased, showing that they were not given the necessary knowledge to understand and assess them. Over both tasks, a two-tailed Wilcoxon Rank-Sum test shows that the participants understand significantly more warnings with MUDARRI ($p = 0.00008 < 0.05$).

Mudarri's NPS [34] is of 13.6, denoting a positive score. In comparison, TRACE received a negative score of -50. Participants noted the highlighting and navigation capabilities of both tools as very useful. Since participants were not told when they interpreted the warnings incorrectly, their impression of the two tools depended entirely on the GUI. Based on the grades, 16 of the 22 participants preferred MUDARRI, which we attribute to the analysis details provided by the tool (**P6** "I can recognize the problem quickly", **P15** "It helps me know why the tool thinks that"). Despite their better performance when using MUDARRI, three participants preferred TRACE. We attribute this discrepancy to the large amount of text in MUDARRI, which can be time-consuming to read, especially when the participant already knows how the warning works (**P13** "There is too much information [...] it breaks my workflow.").

**RQ1**: Analysis-based information helps developers understand warnings significantly better than traditional traces for warnings that they have not seen before. It also helps build up a reusable knowledge of the tool and warning types.

Table 1: List of False Positives Reported by the Taint Analysis in DroidBench, and their Root Causes. **FP1–FP5** are caused by weak analysis rules, **FP6–FP14**, by missing analysis rules.

| ID | Test case | Cause |
|---|---|---|
| **FP1** | ArrayAccess2 | Array over-approximation |
| **FP2** | ArrayAccess5 | Array over-approximation |
| **FP3** | IntentSink2 | Wrong Android lifecycle model |
| **FP4** | IntentSink2 | Wrong Android lifecycle model |
| **FP5** | Merge1 | Incomplete handling of aliases |
| **FP6** | Button2 | Type of callback not handled |
| **FP7** | Exceptions3 | Type of exception not handled |
| **FP8** | Exceptions7 | Type of exception not handled |
| **FP9** | HashMapAccess1 | Collection over-approximation |
| **FP10** | SimpleUnreachable1 | No check for unreachable code |
| **FP11** | UnreachableBoth | No check for unreachable code |
| **FP12** | UnreachableSource1 | No check for unreachable code |
| **FP13** | UnreachableSink1 | No check for unreachable code |
| **FP14** | Unregister1 | Callback behaviour not handled |

## 6.2 Empirical Evaluation, Weak Patterns (RQ2)

We evaluate the weak pattern detection module, using analysis warnings generated with the taint analysis from **RQ1** on DroidBench [2].

### *6.2.1 Experimental Setup*

DroidBench [2] is an open-source benchmark for taint analysis in Android applications, annotated with the exact list of data leaks it contains. At the time of our study, DroidBench contains 187 Android applications on which our taint analysis finds 202 warnings, 14 of which are false positives,

J48:

```
TAINT_ALIAS true
  RETURN_PARAMS true: true positive (2)
  RETURN_PARAMS false: false positive (3)
TAINT_ALIAS false
  API_TO_STRING true: true positive (14)
  API_TO_STRING false
    ARRAY_WRITE true: true positive (3)
    ARRAY_WRITE false
      API_BASE_LOCAL true
        FIELD_ASSIGN true: true positive (8)
        FIELD_ASSIGN false
          RETURN_THIS true: false positive (2)
          RETURN_THIS false: true positive (8)
      API_BASE_LOCAL false: true positive (162)
```

JRip:

(1) ID ∧ TAINT_ALIAS ∧ ¬RETURN_PARAMS ⟹ false positive (2)
(2) ID ∧ ¬LOCAL ∧ RETURN_THIS ⟹ false positive (2)
(3) ⟹ true positive (198)

DecisionTable:

(1) ⟹ false positive (202)

Figure 6: Decision Tree of J48 and Rules of JRip and DecisionTable on DroidBench, and the Number of Warnings Matching Each Rule.

detailed in Table 1. We refer to them as **FP1**–**FP14**. We focus on **FP1**–**FP5**, the ones caused by weak analysis rules.

To answer **RQ2**, we examine the rules yielded by four different rule-based classifiers of WEKA: J48, DecisionTable, RandomForest, and JRip, and verify whether or not the reasons for **FP1**–**FP5** appear in those rules. Using the warnings generated on DroidBench, we use the nodes of the rule graphs as features: the usage of unlabeled edges as features yields a lower precision than with nodes (on average on a 10 cross-fold validation , 0.875 with the nodes and 0.866 with the edges) and recall (on average on a 10 cross-fold validation , 0.922 with the nodes and 0.919 with the edges) and more complex rules. We thus report on the pattern detection module using the nodes as features.

#### 6.2.2 Results

Figure 6 presents the decision tree created by J48, along with the rules generated by JRip and DecisionTable. We do not represent the 100 trees generated by RandomForest, for space reasons.

Let us focus on J48. We note two main false positive patterns: first, when an alias is tainted, the corresponding warning is likely to be a false positive if the taint does not return to a caller through its parameter. This corresponds to **FP5**. Second, the conjunction of API_BASE_LOCAL (API calls tainting the base object if one of the parameters is tainted), and RETURN_THIS (tainting the base object of a caller if the `this` variable is tainted in the callee) is also a false positive pattern. This corresponds to **FP3** and **FP4** and happens because of the incomplete modeling of the Android lifecycle, which causes an entire Activity to be tainted if one of its static attributes are. For true positive patterns, we also see that warnings using the rule ARRAY_WRITE (corresponding to **FP1** and **FP2**) are considered true positives. This is due to DroidBench's five array test cases, three of which are true positives. Using true positive patterns in conjunction with false positive warnings matching those patterns allows us to locate the analysis' weaknesses. From the classifiers' rules and classification results, we can see which analysis patterns match the DroidBench false positives that are due to weak analysis rules: use of arrays, and mishandling aliases and Android Intents, which can then be used to fix the weak analysis rules.

JRip contains fewer rules than J48 (Figure 6): just three. The first one means that the combination of the rules ID and TAINT_ALIAS, along with the absence of the rule RETURN_PARAMS is likely to indicate a false positive, and that two entries of the training set match this rule. Similarly, two entries match the second rule, and the 198 other entries match the default rule: true positive. We see that JRip's rules confirm the main rules from Figure 6: the combination of TAINT_ALIAS and ¬RETURN_PARAMETERS and the RETURN_THIS denoting false positives, while the rest is classified as true positives. We attribute the better performance of J48 to the more efficient pruning of JRip, which is implemented as part of RIPPER [35], an improvement of the REP method used by J48. Combined with the low number of false positives in the training set, JRip's more efficient pruning policy yields simpler rules. This effect is also observed with DecisionTable's unique rule, which classifies all warnings as true positives: since DecisionTable is a majority classifier, the false positives are considered as noise, and the resulting rule selects the majority class: true positives. RandomForest goes in the opposite direction—generating 100 trees of a larger size (from 29 leaf nodes to 69) most of which also contain the three rules discussed above, which we attribute to RandomForest's known overfitting behavior in the case of noisy datasets [36].

**RQ2**: Classifier rules obtained from rule graphs can help determine weak analysis rules. In the case of DroidBench's weak analysis patterns, J48 provides the best tradeoff between pattern size and precision. Integrating a weak rule detection module in an analysis tool can help software developers determine which analysis rules should be adjusted when configuring or using the tool.

### 6.3 Empirical Evaluation, Missing Patterns (RQ3)

To evaluate the missing pattern suggestion module, we use the same benchmark as for **RQ2**, DroidBench, with the same taint analysis as for **RQ1** and **RQ2**. This time, we focus on **FP6**–**FP14** (detailed in Table 1), the false positives caused by missing analysis rules.

#### 6.3.1 Experimental Setup

To assist developers in the detection of missing analysis rules, we generate pairs of similar true / false positives from the DroidBench training set, using the similarity coefficient. As mentioned in Section 4.3, the difference between a pair

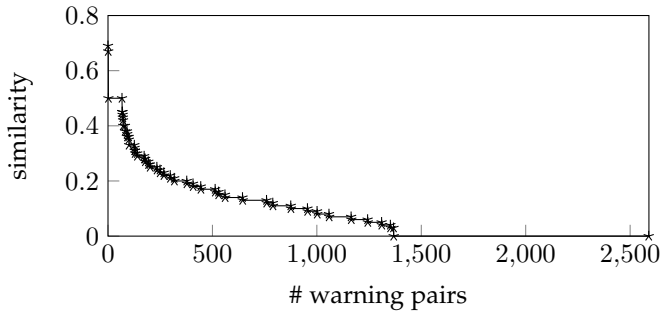



Figure 7: Distribution of the Similarity Coefficient Between Each Pair of Warnings.

of true / false positives which share a high similarity can be indicative of missing rules. So, we point the developer towards the pairs with the highest similarity coefficient.

### 6.3.2 Results

We obtain a total of 2,590 pairs, with similarity coefficients ranging from 0 to 0.691 ($avg = 0.085$, $\sigma = 0.115$). Figure 7 shows the similarity coefficient for each of the pairs in the list. Only two pairs of warnings have a coefficient higher than 0.5, 956 are at 0.1 or more, and 1,222 have a similarity coefficient of 0.

The top two pairs (with coefficients of 0.691 and 0.669) report on **FP6**. In this test case, the true positives share a large part of their traces with the false positive. The difference between the traces mainly consists in calls to the `dummyMain` method, which we use to model the Android lifecycle [2]. This observation reveals the missing analysis support for the particular type of callback found in the false positive. Two pairs concerning exception handling (**FP7** and **FP8**) with coefficients of 0.5 and 0.421, also provide ideal comparisons: the source code, illustrated in Listing 2 (true positive) and Listing 3 (false positive), is almost identical but for the exception types. This similarity leads to almost identical rule graphs (with unlabeled edges), and allows the user to easily identify the missing rule: the analysis does not distinguish between different types of exceptions.

However, other pairs provide less explicit explanations. **FP10** and **FP12** are involved in 64 pairs, all with a similarity coefficient of 0.5. All of those warnings have a simple SOURCE → SINK graph. As a result, it is not easy to infer which rule is problematic since the true positives are so different from the false positives. Other pairs share this issue, namely those concerning **FP9**–**FP14**. Better pairs would require the true positive to be semantically close to the false positive (like in the exception examples), but DroidBench does not contain such test cases. A possible solution to this problem would be to add code-specific information in the similarity coefficient, to differentiate between two identical traces such as the SOURCE → SINK edges of **FP10** and **FP12**.

We note that six of the nine false positives are found in the first 60 pairs, with a similarity coefficient higher than 0.421 or higher. This shows that the coefficient can be used to indicate potential missing patterns. In practice, the user can set a threshold of pairs to investigate, which can be tuned at configuration time.

```
5  try {
6    setImeiAsSourceAndCreateInvalidCastException();
7  } catch (RuntimeException ex) {
8    sms.sendTextMessage(num, null, imei, null,
         null);
9  }
```

Listing 2: DroidBench Exceptions3 Test Case. True Positive (InvalidCastException is a RuntimeException).

```
10 try {
11   setImeiAsSourceAndCreateInvalidCastException();
12 } catch (ArrayIndexOutOfBoundsException ex) {
13   sms.sendTextMessage(num, null, imei, null,
         null);
14 }
```

Listing 3: DroidBench Exceptions7 Test Case. False Positive.

**RQ3**: Rule graphs can be used to compute warning similarities and to point the developer towards missing analysis rules, helping them add missing rules in the analysis' ruleset. Code-specific features can also be used in the similarity computation to increase the precision of the approach.

## 7 Limitations and Future Work

We ran the user study in a controlled environment in which participants only had 20 minutes to work on small-scale applications. Because getting used to a larger code base or to too many analysis results would have taken longer than the 15-minutes dedicated to each task, we chose to limit the size of the applications. The two applications are still real-world applications from F-Droid. While running the experiments in a controlled environment allowed us to remove external threats to validity, it would be interesting to also evaluate MUDARRI in a real-life environment.

The user study was a within-subjects study, so we observed a learning effect in which participants performed better on the second task. We addressed this issue with a latin-square design, in which half of the participants used MUDARRI first, and the other half, TRACE first, and reported on the aggregated results of both halves.

It is subjective to decide whether the developer understood a warning or not, because it depends on the rater's interpretation of "understanding". To limit the subjectivity of the rater's judgement, we considered that the participant correctly understood a warning if they could name the correct source and sink, which variables were tainted along the way, and how the taint transfers were made for each step. This is the reasoning a developer would need to build when dealing with taint analysis warnings. Subjective factors such as whether the leaked information contains sensitive data, or whether the warning has a high impact or not were thus kept out of the decision.

As seen in Section 6.2, the use of analysis rules for pattern extraction is not enough to completely distinguish all warning classes. External code and warning-specific features could improve the classification. We leave this for future work.

Participants have reported that the explanations provided by MUDARRI can be time-consuming to read, especially when the participant already knows about the warning. Learning which level of detail to display to which developer in which circumstances is an interesting research topic that we leave for future work.

The quality of the rule patterns found in Section 6.2 heavily depend on the quality of the training set. Like all machine learning approaches, a bad training set introduces uncertainties and yields inaccurate results. In our evaluation, we used DroidBench, a benchmark of minimal examples for taint analysis with both true and false positives as our training set. While not complete, this benchmark covers the most common use cases for Android. To apply our approach to other types of applications or languages, it will be important to define a good training set first.

A drawback of the pattern-detection module using machine learning is its scalability, which we mitigate by running it offline. It would be possible to apply the learning incrementally, and then use the module interactively in the IDE, which we leave for future work. Another alternative would be to reduce the patterns and graph matching to simpler heuristics that can be quickly verified, and to rerun the entire method offline from time to time.

The rule graph approach aims to help developers understand how the analysis reasons and give them insights into what could potentially go wrong with the analysis. For developers to make best use of the rule graphs, it is important to use a set of markers adapted to the codebase and to the needs of the users. This requires from the person doing the set up some knowledge of static analysis and its weaknesses in the context in which the tool is meant to be used. However, once the rule graphs are set up, the tool can be used like any existing static analysis tool, without the need for domain-specific knowledge. On the contrary, it surfaces this domain-specific knowledge to the user by displaying it in the interface.

## 8 RELATED WORK

In this section, we present past research about improving warning explainability, and the integration of developer-specific knowledge in the analysis rules.

### 8.1 Warning Explainability

Past studies have highlighted warning explainability as one of the major issues of static analysis tools. Fourteen of the twenty developers interviewed by Johnson et al. [8] reveal that poorly presented output is one of the main reasons for tool underuse. With a study on Microsoft developers, Christakis et al. [10] show that the second most popular pain point of static analyzers is the bad warning messages, the lack of fix suggestions coming fifth on the list of 15 pain points. In a study at Google, Lewis et al. [9] cite "obvious reasoning" as a desirable feature of an analysis tool, where they advocate for the analysis tool to provide clear proof of why a warning is reported. In a study of Fortify SCA [18], Ayewah et al. [11] find that badly explained traces harm the understanding of the bug and the confidence of the developer in the tool. Those studies motivate the need for better developer support in understanding analysis warnings.

Similarly, for warning understanding, current approaches base themselves on external information. Phang et al. [37] only focus on the code when visualizing warning traces. Nanda et al. [38] visualize warnings based on warning and code information, like modern commercial tools such as Checkmarx [39] or CodeSonar [17]. With our approach, we argue that internal analysis-specific information gives more insight into the warning.

### 8.2 Usage of Internal Analysis Rules

The idea of integrating analysis-specific knowledge to assist developers has been mentioned by Jung et al. [40]. They use a statistical analyzer to help triage false positives from true positives, based on "symptoms". In their conclusion, they advance the idea that using the analysis' weaknesses as symptoms would yield better results than external symptoms such as coding features for example. Past work on nano-patterns has proven data-flow and control-flow features useful to characterize Java methods [41] and detect software vulnerabilities [42]–[44], but those features are not used for warning understanding and analysis configuration. In our approach, we use internal information to aid developers understand and fix warnings, and help them identify weaknesses in the analysis rules.

## 9 CONCLUSION

In this article, we advocate for more transparency of the analysis' inner-workings in order to enhance the understandability of static analysis warnings, and help developers configure the analysis rules by identifying weak and missing analysis rules. To this end, we have presented the concept of rule graphs and presented a general recipe for applying it to how to data-flow analysis frameworks. With our implementation of the rule graphs and of the modules illustrating their usages: the MUDARRI IntelliJ plugin, the pattern detection module, and the similarity computation module, we have demonstrated how to apply the concept of rule graphs for taint analysis.

Through a user study with 22 developers and an empirical evaluation on the 187 applications of DroidBench, a reputable benchmark for static analysis for Android applications, we show that rule graphs can enhance the reporting of analysis warnings and assist developers in evaluating the analysis rules, thus providing them with a better user experience of static analysis when using and configuring it.

With this research, we encourage other researchers and practitioners to explore other applications of rule graphs for a more transparent use of analysis rules in static analysis tools. A few examples could be for warning classification based on rule graph similarities, or fix generation using the comparison of a warning's rule graph and its fix's.

## ACKNOWLEDGMENTS

This research has been partially funded by the Heinz Nixdorf Foundation, by the BMBF within the Software Campus initiative, the DFG project RUNSECURE, and the NRW Research Training Group on Human Centered Systems Security (nerd.nrw).

## REFERENCES


[1] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, 2018, pp. 10:1–10:27. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2018.10

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594299

[3] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS. Springer, Cham, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-17465-1_22

[4] J. Späth, K. Ali, and E. Bodden, "Ideal: Efficient and precise alias-aware dataflow analysis," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 99:1–99:27, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133923

[5] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Krishnamurthi and B. S. Lerner, Eds., vol. 56. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:26. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6116

[6] B. G. Ryder, "Incremental data flow analysis," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: ACM, 1983, pp. 167–176. [Online]. Available: http://doi.acm.org/10.1145/567067.567084

[7] L. Nguyen Quang Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 307–317. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092705

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.

[9] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 372–381.

[10] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 332–343. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970347

[11] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, ser. DEFECTS '08. New York, NY, USA: ACM, 2008, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1390817.1390819

[12] MITRE, "Cwe-89: Improper neutralization of special elements used in an sql command ('sql injection')," https://cwe.mitre.org/data/definitions/89.html, 2019.

[13] ——, "Cwe-601: Url redirection to untrusted site ('open redirect')," https://cwe.mitre.org/data/definitions/601.html, 2019.

[14] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2009.

[15] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 462–473. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786851

[16] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 341–350. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368135

[17] Grammatech, "Codesonar home page," https://www.grammatech.com/products/codesonar, 2019.

[18] F. Software, "Fortify home page," https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview, 2019.

[19] L. Nguyen Quang Do and E. Bodden, "Source code of MUDARRI," https://github.com/secure-software-engineering/mudarri, 2019.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[21] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?" in *Compiler Construction*, D. A. Watt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 18–34.

[22] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[23] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, ser. SOAP '12. New York, NY, USA: ACM, 2012, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259052

[24] J. Lerch, B. Hermann, E. Bodden, and M. Mezini, "Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 98–108. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635878

[25] JetBrains, "Intellij home page," https://www.jetbrains.com/idea/, 2019.

[26] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[27] A. Ulanov, A. Simanovsky, and M. Marwah, "Modeling scalability of distributed machine learning," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017, pp. 1249–1254.

[28] P. Berander, "Using students as subjects in requirements prioritization," in *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ser. ISESE '04. USA: IEEE Computer Society, 2004, p. 167–176.

[29] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjects – a comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, pp. 201–214, 2000.

[30] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 666–676.

[31] M. Svahnberg, A. Aurum, and C. Wohlin, "Using students as subjects - an empirical evaluation," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 288–290. [Online]. Available: https://doi.org/10.1145/1414004.1414055

[32] F-Droid, "Balance," https://f-droid.org/en/packages/de.mangelow.balance/, 2019.

[33] ——, "Sparkleshare," https://f-droid.org/en/packages/org.sparkleshare.android/, 2019.

[34] F. F. Reichheld, "The one number you need to grow," *Harvard Business Review*, vol. 81, no. 12, pp. 46–55, 2003.

[35] W. W. Cohen, "Fast effective rule induction," in *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 115–123.

[36] M. R. Segal, "Machine learning benchmarks and random forest regression," *UCSF: Center for Bioinformatics and Molecular Biostatistics*, 2004.

[37] Y. Phang, J. S. Foster, M. Hicks, and V. Sazawal, "Triaging checklists: a substitute for a phd in static analysis," in *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, ser. PLATEAU'05, 2009.

[38] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making defect-finding tools work for you," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810310

[39] Chechmarx, "Chechmarx home page," https://www.checkmarx.com/, 2019.

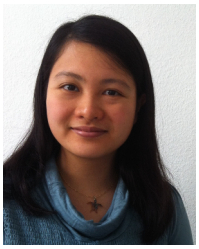
[40] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis," in *Proceedings of the 12th International Conference on Static Analysis*, ser. SAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 203–217. [Online]. Available: http://dx.doi.org/10.1007/11547662_15

[41] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis, "Fundamental nano-patterns to characterize and classify java methods," *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 7, pp. 191–204, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2010.08.042

[42] K. Z. Sultana, A. Deo, and B. J. Williams, "Correlation analysis among java nano-patterns and software vulnerabilities," in *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, Jan 2017, pp. 69–76.

[43] G. Destefanis, R. Tonelli, E. Tempero, G. Concas, and M. Marchesi, "Micro pattern fault-proneness," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, Sep. 2012, pp. 302–306.

[44] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?" in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 1963–1969. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774504



**Lisa Nguyen Quang Do** is a software engineer at Google. She has received her Ph.D. in Computer Science from Paderborn University in 2019. Her research focuses on improving the usability of analysis tools for code developers and analysis developers through different aspects that range from the optimization of the analysis algorithm to the implementation of its framework to the usability of its interface.

**Eric Bodden** is a full professor for Secure Software Engineering at the Heinz Nixdorf Institute of Paderborn University, Germany. He is further the director for Software Engineering at the Fraunhofer Institute for Engineering Mechatronic Systems. Prof. Bodden has been recognized several times for his research on program analysis and software security, most notably with the German IT-Security Price and the Heinz Maier-Leibnitz Price of the German Research Foundation, as well as with several distinguished paper and distinguished reviewer awards.