Testing Static Taint Analyzers with Equivalence Modulo Taint

MARIA CHRISTAKIS, TU Wien, Austria
ANASTASIA ISYCHEV, TU Wien, Austria
SAMUEL PILZ, TU Wien, Austria
FLORIAN TESAREK, TU Wien, Austria
VALENTIN WÜSTHOLZ, Consensys Diligence, Austria

Static taint analyzers are widely used to detect security vulnerabilities, yet their complexity makes them prone to soundness and precision issues. Validating these analyzers is challenging because ground-truth taint flows are rarely available and differential testing requires multiple comparable tools. To address this challenge, we introduce *Equivalence Modulo Taint* (EMT), a testing oracle for static taint analysis that defines program equivalence in terms of preserved source-sink flows rather than program semantics. EMT enables testing a single analyzer without ground-truth labels by checking consistency of reported flows across equivalent-modulo-taint program variants. Based on EMT, we present TaintCC, a framework that generates equivalent-modulo-taint variants through semantically equivalent, taint-oblivious, and taint-aware transformations targeting recurring difficulty dimensions in taint analysis. We evaluate TaintCC on four widely used analyzers—FlowDroid, Mariana Trench, Pysa, and Semgrep—and uncover 16 unique developer-confirmed issues, showing that even mature analyzers, whether academic or industrial, remain susceptible to reliability issues.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: static taint analysis, metamorphic testing, equivalence modulo taint, program analysis testing, information-flow analysis, soundness and precision

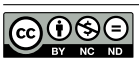
ACM Reference Format:
Maria Christakis, Anastasia Isychev, Samuel Pilz, Florian Tesarek, and Valentin Wüstholz. 2026. Testing Static Taint Analyzers with Equivalence Modulo Taint. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA140 (October 2026), 22 pages. https://doi.org/10.1145/3832231

1 Introduction

Static taint analysis [11] is a key approach for identifying information-flow vulnerabilities, such as injection attacks and data leaks, before software is deployed. These analyses track the propagation of untrusted data from *sources* (e.g., user input or network interfaces) to *sinks* (e.g., database calls, file writes, or logging statements), and report a potential vulnerability whenever tainted data can flow from a source to a sink. By reasoning statically about how untrusted inputs propagate through a program, taint analysis can provide coverage across all execution paths without requiring concrete inputs or runtime monitoring. This makes taint analysis particularly attractive in security-critical environments, where a single missed flow can have security consequences at scale.

The importance of taint analysis is underscored by its widespread adoption in industrial-scale settings. For example, to secure its massive codebases, Meta has developed several static taint analyzers tailored to different languages and platforms. In particular, Zoncolan [22], a closed-source

Authors' Contact Information: Maria Christakis, TU Wien, Vienna, Austria, maria.christakis@tuwien.ac.at; Anastasia Isychev, TU Wien, Vienna, Austria, anastasia.isychev@tuwien.ac.at; Samuel Pilz, TU Wien, Vienna, Austria, samuel.pilz@tuwien.ac.at; Florian Tesarek, TU Wien, Vienna, Austria, florian.tesarek@tuwien.ac.at; Valentin Wüstholz, Consensys Diligence, Vienna, Austria, valentin.wustholz@diligence.security.



ACM 2994-970X/2026/10-ARTISSTA140
https://doi.org/10.1145/3832231

tool, protects Hack applications by encoding rules distilled from recurring bug patterns observed in production. Pysa [2] scales to millions of lines of Python code powering Instagram, delivering results quickly enough to influence everyday development. Mariana Trench [1] targets Android applications, where delays in patch deployment make early detection especially critical. Collectively, these analyzers illustrate the central role of taint analysis at one of the largest technology companies.

Despite their widespread adoption, static taint analyzers—like program analyzers more generally—remain complex and prone to soundness and precision issues (e.g., [7, 19, 20, 27, 31, 34]). Developers and security teams rely on these tools to flag vulnerabilities early, yet their soundness and precision are difficult to validate systematically. In practice, incorrect results have tangible consequences: false positives burden engineers with manual investigation and erode trust in the analyzer, while false negatives allow bugs to slip through undetected and potentially reach production. Validating taint analyzers is therefore a fundamental challenge for securing software systems.

Existing work. Several techniques have been proposed for testing program analyzers, with the most prominent being program generation, specification-based testing, differential testing, and metamorphic testing.

Program generation aims to automatically generate input programs whose correctness is known by construction, for instance, through constraint solving or controlled synthesis (e.g., [6, 13, 16]). An analyzer is then run on these programs, and an issue is detected whenever its result contradicts the expected one. Since predicting an analyzer's behavior on arbitrary programs is undecidable in general, program-generation approaches often restrict the diversity or complexity of the generated programs to ensure that correctness can still be determined.

Specification-based testing instead relies on an explicit description of the analyzer's intended behavior, for instance, in the form of formal specifications or reference rules (e.g., [7, 27, 31]). The analyzer is tested, often in combination with fuzzing, against this specification. While effective when precise specifications are available, this approach is difficult to scale: writing comprehensive specifications for large and evolving analyzers is labor-intensive and often infeasible in practice.

Differential testing [26] evaluates analyzers by running multiple tools on the same input program and flagging discrepancies as potential issues (e.g., [12, 19, 20, 29, 32]). This approach faces several challenges. It inherently requires multiple analyzers, yet it is often unclear which analyzer, if any, is correct. Moreover, analyzers do not always accept the same input formats or even the same input language, making it difficult to ensure that the compared programs are exactly equivalent. In addition, heterogeneous design assumptions further complicate interpretation: differences often stem from varying levels of precision, for example in the sophistication of an alias analysis. As a result, the discrepancies reported by differential testing are costly to triage, since they require careful human reasoning to distinguish genuine issues from benign disagreement.

Metamorphic testing [9, 30] addresses the oracle problem by defining relations between program variants, asserting that specific transformations should preserve or predictably change behavior (e.g., [14, 23–25, 28, 33–36]). Equivalence Modulo Inputs (EMI) [21], a prominent realization of this idea for compiler testing, generates program variants by mutating code that does not affect a given input, and flags deviations in execution outcomes as issues. Recently, interrogation testing [17, 18] extended metamorphic testing by leveraging analysis justifications (i.e., explanations why an analyzer finds a program safe or unsafe) to generate more sophisticated transformations, tailored to the analyzer's reasoning process. In general, metamorphic testing has proven effective not only for compilers [8] and program analyzers, but also more broadly, in domains such as machine-learning and zero-knowledge systems [10, 15]. These lines of work demonstrate the power of metamorphic testing to uncover issues without requiring precise specifications or multiple analyzers.

However, the vast majority of existing metamorphic-testing techniques rely on equivalence notions aligned with concrete execution behavior or full program semantics. Static taint analyzers,

by contrast, reason over abstract dataflow properties—specifically, source-sink information flows. Preserving equivalence with respect to concrete execution or full program semantics is therefore unnecessary and overly restrictive in this domain. As a result, directly reusing existing metamorphic techniques would overconstrain transformations. This mismatch motivates the need for a taint-specific oracle that preserves exactly the property of interest for taint analysis.

Our approach. We present the *first systematic testing framework for static taint analyzers* built around an analyzer-relative, domain-specific metamorphic oracle that systematically detects both soundness and precision issues. In interrogation testing, metamorphic transformations are guided by the analyzer's justifications that explain why a program is considered safe or unsafe. In our taint-analysis context, these justifications can take the form of a *witness path*, that is, a program path demonstrating how data flows from a source to a sink. We, therefore, exploit witness paths to design transformations that manipulate taint behavior in predictable ways. In addition, we introduce the notion of *Equivalence Modulo Taint* (EMT), where two program variants are considered equivalent if the analyzer under test reports the same set of source-sink flows. We intentionally define this oracle to be the *weakest equivalence relation sufficient for taint analysis*: unlike prior metamorphic approaches that enforce full semantic equivalence, EMT permits transformations that arbitrarily change program semantics as long as taint flows are preserved. By checking whether a taint analyzer produces consistent results under transformations that preserve EMT, we can automatically uncover both soundness issues (missed flows) and precision issues (spurious flows).

We implemented our technique in a tool called TaintCC, short for Taint Carbon Copy. The name reflects the key idea: we generate a "carbon copy" of the original program in the form of an equivalent variant. Although its syntax and semantics may differ substantially, it is designed to preserve the same information flows as the original. These carbon copies form the basis for checking analyzers against EMT. We applied TaintCC to four popular and mature static taint analyzers—namely, FlowDroid [4], Mariana Trench [1], Pysa [2], and Semgrep [3]—and found 16 unique developer-confirmed issues in all four analyzers, of which eight are soundness issues and eight are precision issues. Notably, Mariana Trench and Pysa are developed and deployed at Meta, yet our experiments reveal that even such industrial-strength analyzers remain susceptible to reliability issues.

Our contributions. This paper makes the following contributions:

- We present the first systematic testing framework for static taint analyzers to detect soundness and precision issues. Our approach introduces Equivalence Modulo Taint, a domain-specific metamorphic oracle that requires preserving only reported source-sink flows.
- We implement this framework in TaintCC, which generates program variants that are syntactically and semantically different, yet they exhibit the same information flows.
- We evaluate TaintCC on four popular and mature static taint analyzers, namely, FlowDroid, Mariana Trench, Pysa, and Semgrep, and report the discovery of 16 unique developer-confirmed issues, of which eight are soundness issues and eight are precision issues.

2 Overview

Workflow. An overview of TaintCC is shown in Fig. 1. To test analyzers that operate on different programming languages (e.g., Python or Java), TaintCC generates and manipulates programs at the level of a restricted abstract syntax tree (AST), which serves as the internal program representation throughout testing. At a high level, the workflow consists of five stages: AST generation, AST translation to the target language, AST transformation, taint analysis, and issue detection.

As shown in the figure, the AST-generation stage produces a random, original abstract syntax tree, denoted T_{og}. This AST is translated to the target language, yielding the program P_{og}. The taint

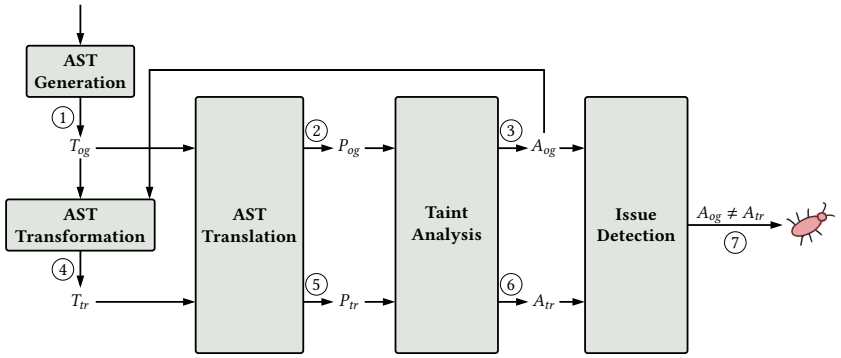


Fig. 1. Overview of TaintCC.

analyzer under test is then run on P_{og}, producing analysis results A_{og}. Next, T_{og} is transformed using EMT transformations into an equivalent AST T_{tr}. Importantly, the transformations may make use of A_{og}, i.e., the flows reported for P_{og}. The transformed AST T_{tr} is then translated into P_{tr}, which is in turn analyzed to obtain results A_{tr}. If A_{tr} differs from A_{og}, an issue is detected. Otherwise, T_{tr} becomes the new T_{og} and is subjected to further transformations (not shown in the figure). After a configurable number of such iterations, a fresh T_{og} is generated, and the process repeats.

AST. The AST used by TaintCC is a custom, restricted abstract syntax tree tailored for object-oriented languages. In particular, it abstracts over language-specific syntax while retaining key constructs such as classes, methods, fields, statements, and expressions. Because the AST defines a restricted set of such constructs with predetermined scoping and typing rules, generating a valid tree does not require handling the full Java or Python grammar, and every generated AST is well formed by construction. During generation, the AST is augmented with randomly inserted sources and sinks to introduce information flows.

The design goal of this AST is not to approximate real-world code structure, but to exercise the taint-propagation logic of analyzers in a systematic and language-agnostic manner. Static taint analyzers abstract away most concrete language idioms; accordingly, the AST is designed to cover the core operations that drive taint flow—such as assignments, field and collection updates, control flow, and interprocedural calls—while avoiding irrelevant syntactic variation. The AST supports straightforward translation into both Java and Python, the languages supported by the taint analyzers we test.

Example. Fig. 2 shows a program P_{og} obtained by translating a randomly generated AST T_{og} into Java for testing FlowDroid. What the user considers to be source and sink functions is specified in the configuration provided to FlowDroid. In this example, for ease of presentation, they are simply named `source` and `sink`. On line 10, the value returned by `source` is added to `taint_list`, which is then passed to `sink` on line 15. Despite this flow from source to sink, FlowDroid does not report any flows, so A_{og} is empty. However, because we do not know the ground truth for this randomly generated program, TaintCC cannot yet identify this soundness issue.

Based on A_{og}, the AST-transformation component then applies a sequence of stacked EMT transformations to T_{og}, that is, to the AST of the program in Fig. 2. Since A_{og} is empty, these transformations ensure that no flows are added to the program, in other words, they preserve taint. In this particular case, they replace a constant and remove an unused statement. The generated, transformed AST T_{tr} is translated into P_{tr}, which is obtained from the code in Fig. 2 by removing line 13 and replacing the constant `-1` on line 12 with `0`. When running FlowDroid on P_{tr}, the flow from `source` to `sink` is detected, and it is included in A_{tr}. (Note that assigning any non-negative

```
1  package com.example.testapp;
2  import androidx.appcompat.app.AppCompatActivity;
3  import android.os.Bundle;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  public class MainActivity extends AppCompatActivity {
8    public void onCreate(Bundle savedInstanceState) {
9      List<String> taint_list = new ArrayList<String>();
10     taint_list.add(this.source());
11
12     Integer dummy_counter = -1;
13     List<Boolean> unused_list = new ArrayList<Boolean>();
14
15     this.sink(taint_list);
16
17     System.out.println(dummy_counter);
18   }
19
20   public String source() {
21     return "Secret";
22   }
23
24   public void sink(List<String> arg) { }
25 }
```

Fig. 2. Program generated by TaintCC exposing a soundness issue in FlowDroid.

integer to `dummy_counter` on line 12 has the same effect.) This discrepancy between A_{og} and A_{tr} indicates an issue in FlowDroid, and manual inspection confirms that it is soundness related.

The root cause of this bug was a faulty implementation of the alias analysis in FlowDroid's intermediate representation (IR), called Jimple. The developers provided a fix, but it only partially resolved the problem. After the first patch, `dummy_counter` was no longer critical, yet `unused_list` continued to interfere with the analysis results. In particular, both `taint_list` and `unused_list` were assigned to the same register in the Jimple IR, and the second assignment (to `unused_list`) incorrectly removed the taint from `taint_list`. A subsequent patch fully corrected this issue.

3 Approach

In this section, we present our approach to metamorphic testing for taint analyzers in detail. We begin by formalizing the notion of EMT and then describe the classes of transformations supported in TaintCC. All transformations operate on the AST representation of Sect. 2.

3.1 Equivalence Modulo Taint

Metamorphic transformations for testing analyzers come in different forms. Some preserve program semantics entirely, ensuring that the original and transformed programs behave identically on all inputs (e.g., [14, 35]). Others deliberately change program semantics, but in a predictable way that still allows the expected analyzer response to be derived (e.g., [23, 25, 28]). Equivalence Modulo Inputs (EMI) [21] represents another category, where equivalence is defined only with respect to a specific input, and code that does not affect the semantics of the corresponding trace may be freely mutated. In contrast, Equivalence Modulo Taint (EMT), introduced in this work, preserves only the taint flows in a program. *Transformations under EMT may change other aspects of program semantics unpredictably*, but they guarantee that the flows between sources and sinks remain the same.

How are these flows obtained? From the analyzer under test. In other words, EMT preserves the taint flows in a program *as they are determined by the analyzer itself.* By using the analyzer's

results to guide the transformations, we aim to drive the analyzer into a contradiction: if two equivalent-modulo-taint programs produce different results, then an issue is exposed. Such an EMT violation is evidence of an unexpected analyzer behavior that merits investigation; it does not by itself determine whether the root cause is an implementation defect or a limitation of the analysis.

But what exactly are these analysis results? Without loss of generality, we assume that taint analyzers emit witness paths showing how data flows from a source to a corresponding sink. In practice, these witness paths may over-approximate the true flows. Some analyzers may report only source-sink pairs, in which case the witness path can be understood as any path connecting the two. Others may report only sinks, in which case the witness path is any path leading to the reported sink. Overall, EMT is not restricted by the form of results produced by each taint analyzer.

Definition (Equivalence Modulo Taint). *Let P be a program and let $A(P)$ denote the set of taint flows reported by an analyzer A on input P. Two programs P and P' are* equivalent modulo taint *with respect to A iff*

$$A(P) = A(P').$$

According to the definition, P and P' are equivalent modulo taint if the analyzer reports the same set of taint flows for both programs, regardless of any other semantic differences between them. This also means that EMT is defined relative to a specific analyzer A, in particular its reported analysis results, while remaining agnostic to the analyzer's internal design and implementation.

EMT vs. EMI. EMT differs from EMI at the level of the testing oracle. EMI defines equivalence with respect to concrete executions on a chosen input: program variants may differ outside the executed trace, but they must preserve the same observable behavior on that input. EMT instead defines equivalence with respect to the analyzer's reported source-sink flows: program variants may change concrete behavior, and even full program semantics, as long as the reported flows are preserved. As a result, EMT induces a different transformation space from EMI: some transformations allowed under EMT are disallowed under EMI, and vice versa. In particular, EMT does not rely on dead code, concrete inputs, or execution-equivalent traces, but on preservation of the information-flow property of interest.

3.2 Metamorphic Transformations

We distinguish three categories of EMT transformations supported by TaintCC:

- **Semantically equivalent transformations**, which preserve the semantics of the program;
- **Taint-oblivious transformations**, which may alter program semantics but not in ways that affect taint flows; and
- **Taint-aware transformations**, which explicitly use the analyzer's reported witness paths to guide the mutation.

In the following, we describe the key idea behind each category and illustrate it with example transformations implemented in TaintCC. We conclude the section with a discussion of the rationale underlying our transformation design.

Semantically equivalent transformations. Given a program P, a semantically equivalent transformation produces a program P' such that P and P' are identical in behavior; that is, for all inputs, they produce the same observable behavior. Because EMT transformations only require that P and P' yield the same taint-analysis results, any two semantically equivalent programs are trivially equivalent modulo taint. Since semantically equivalent transformations have already been extensively explored in prior work, TaintCC includes transformations of this category primarily to increase the diversity of generated programs and to exercise different dimensions of taint analyzers (e.g., interprocedural dataflow, control flow). Below, we illustrate the idea with representative examples in Python.

```
1 s = source()
2 t = 0
3 if x < 42:
4   t = s
5 sink(t)
```

```
1 s = source()
2 t = 0
3 if x < 42:
4   t = s
5   u = 777 + x # inserted
6 sink(t)
```

Fig. 3. Example of the DeclareNewVar transformation. The original program (left) is transformed into the variant (right) by inserting a fresh variable `u` initialized with a pure expression. Since this insertion does not affect taint propagation from `source` to `sink`, the two programs are equivalent modulo taint.

DeclareNewVar. This transformation introduces a fresh local or global variable and initializes it with a pure random expression that does not contain any sinks. For instance, in the example of Fig. 3, the original program on the left is transformed (right) by inserting a new variable `u` on line 5.

AddNestedStmt. This transformation inserts a new pure nested statement, such as a conditional or loop, that does not affect program semantics. In the example of Fig. 4a, the original program (left in Fig. 3) is transformed by adding an extra conditional statement with a body that does nothing when executed (lines 3–4).

ReplaceWithId. The transformation ReplaceWithId replaces an expression E with $id(E)$, where id is the identity function. For instance, the constant expression `42` on the left of Fig. 3 is replaced with `Id(42)` to obtain the code in Fig. 4b.

Other examples of semantically equivalent transformations implemented in TaintCC include replacing attribute reads and writes with getter and setter calls (UseGetter, UseSetter), function inlining (InlineFnc), and mutating certain analyzer configurations (MutAnalysisConfig). All these transformations preserve program semantics and, consequently, taint flows, resulting in program variants that are equivalent modulo taint.

Taint-oblivious transformations. The second category comprises taint-oblivious transformations. These transformations do not depend on the analysis results of the original program. Instead, they may alter the program's structure or behavior, but never in ways that affect taint flows. In other words, a taint-oblivious transformation remains equivalent modulo taint regardless of any witness paths reported by the analyzer. Below, we again illustrate the idea with representative examples in Python.

AddConstant. This transformation replaces an expression E with $E + C$, where C is a random constant of a compatible type; that is, the resulting expression must type-check whenever the original expression E does. The transformation is equivalent modulo taint because E and $E + C$ are either both tainted or both untainted. It is taint oblivious because its correctness does not depend

```
1 s = source()
2 t = 0
3 if x < 100: # inserted
4   pass      # inserted
5 if x < 42:
6   t = s
7 sink(t)
```

(a)

```
1 s = source()
2 t = 0
3 if x < Id(42): # transformed
4   t = s
5 sink(t)
```

(b)

Fig. 4. (a) Example of the AddNestedStmt transformation. The original program (left in Fig. 3) is transformed into this variant by inserting an additional conditional with a body that does nothing when executed. Since this insertion does not change program semantics, the two programs are trivially equivalent modulo taint. (b) Example of the ReplaceWithId transformation. The original program (left in Fig. 3) is transformed into this variant by replacing the constant expression `42` with `Id(42)`. Since the identity function preserves semantics, the two programs remain equivalent modulo taint.

```
s = source()
t = 0
if x < 42:
  t = s + 777 # transformed
sink(t)
```

(a)

```
s = source()
t = 0
if x < 42:
  t = s
  sink("foo") # inserted
sink(t)
```

(b)

Fig. 5. (a) Example of the AddConstant transformation. The original program (left in Fig. 3) is transformed into this variant by adding the constant integer `777` to variable `s`. Since both expressions are either tainted or untainted, the two programs are equivalent modulo taint. (b) Example of the AddSinkWithConstant transformation. The original program (left in Fig. 3) is transformed into this variant by inserting a sink call with the constant argument `"foo"`. Since constants are untainted, the additional sink does not affect the reported flows, and the two programs are equivalent modulo taint.

on whether E is actually tainted. In the snippet of Fig. 5a, the constant integer `777` is added to variable `s`. Conceptually, this transformation can be generalized in two ways: by replacing the constant with any expression generated to be untainted by construction, or by replacing addition with any operation whose result is tainted if and only if at least one argument is tainted.

AddSinkWithConstant. This transformation inserts a sink call with a random constant argument. Since the argument is a constant, it is guaranteed to be untainted and therefore does not affect the analysis results. For example, in the snippet of Fig. 5b, an additional sink call with the argument `"foo"` is inserted. Conceptually, this transformation can be generalized from a constant argument to any expression that is untainted by construction.

ReplaceConstant. This transformation replaces a constant expression E with another random constant C of a compatible type. For instance, in the snippet of Fig. 6a, the integer constant `42` is replaced with `43`. In theory, such a change could affect the outcome of a dynamic taint analysis (e.g., if the analysis is run with input `x = 42`). However, our focus is on static taint analyzers, which use coarse abstractions for constants. Intuitively, static taint analyzers cannot distinguish between E and C, and therefore, their reported flows remain unchanged. Conceptually, this transformation can be generalized from constants to arbitrary expressions that are untainted by construction.

Other taint-oblivious transformations implemented in TaintCC include AppendConstantToList, which appends a random constant element to a list. Since the appended element is untainted by construction, this transformation cannot affect taint flows, and therefore, preserves equivalence modulo taint. MoveSinkInCond creates an auxiliary copy of a variable passed to a sink, replaces the sink argument with this copy, and relocates the sink inside a conditional guarded by an equivalence check between the original and the copy (which always evaluates to true).

```
s = source()
t = 0
if x < 43: # transformed
  t = s
sink(t)
```

(a)

```
s = 43 # transformed
t = 0
sink(t)
if x < 42:
  t = s
```

(b)

Fig. 6. (a) Example of the ReplaceConstant transformation. The original program (left in Fig. 3) is transformed into this variant by replacing `42` with `43`. Since static taint analyzers abstract constants coarsely, this change does not affect the reported flows, and the programs remain equivalent modulo taint. (b) Example of the RemoveSourceCall transformation. The original program (left in Fig. 7) is transformed into this variant by replacing `source()` with `43`. Since the source does not flow into any sink, this replacement preserves the reported flows, and the programs remain equivalent modulo taint.

```
s = source()
t = 0
sink(t)
if x < 42:
  t = s
```

```
s = source()
t = 0
u = t   # inserted
sink(t)
if x < 42:
  t = s
sink(u) # inserted
```

Fig. 7. Example of the AddNewSink transformation. The original program (left) is transformed into the variant (right) by introducing a fresh variable u to store the argument of the first sink and adding a new sink call with u as its argument. Since both sink arguments are equivalent modulo taint, the two programs remain equivalent modulo taint.

Because the condition does not influence whether the sink argument is tainted, this transformation preserves reported flows. Its purpose is to test whether the analyzer consistently recognizes the sink across syntactically different control-flow contexts, including those with potential aliasing. Finally, InsertPrint inserts a print statement.

Taint-aware transformations. The last category is taint-aware transformations. These transformations rely on the results of the taint analysis for the original program. For instance, any expression E can be replaced with another expression E' that is equivalent modulo taint. To avoid introducing unintended side effects, E and E' should generally be pure expressions. The following two cases illustrate the idea:

- **Untainted expressions.** If E is untainted, then it can be replaced by any other untainted expression E'. Examples include constants or computations over local variables that cannot be influenced by sources.
- **Tainted expressions.** If E is tainted, then it can be replaced by any other expression E' that is also tainted by the same source. For example, a call to `source` could be replaced by another expression derived from `source`.

To enable such replacements, we introduce several sources and sinks into the program (a step performed during AST generation, as mentioned in Sect. 2). These allow us to probe the analyzer's classification of expressions as tainted or untainted, and the results can then be used to guide subsequent transformations. Importantly, *this intuition generalizes from expressions to entire programs*: transformations that operate locally on expressions can be composed to yield equivalent-modulo-taint program variants.

Note that the results of the taint analysis may themselves be incorrect, for example, by unintentionally reporting an additional taint flow or by missing one. Interestingly, this is not a limitation for our use case: even if the analysis results are incorrect, a taint-aware transformation can still reveal inconsistencies. For instance, if the analyzer mistakenly reports that a variable is tainted (due to a bug in its implementation), our EMT transformations might replace the variable with another expression derived from the same supposed source. Both variants should then produce the same flows. If they do not, the analyzer contradicts itself, and an issue is revealed.

AddNewSink. Suppose there is a sink S with argument E that is classified as untainted by the taint analysis. This transformation inserts a new sink S' with an argument E' that is intended to be equivalent to E. Concretely, the transformation introduces a fresh auxiliary variable V immediately before S to store the argument E, and then inserts S' with argument V. The new sink S' is placed so that it is reachable from S in the control-flow graph, and its lexical scope is either the same as or a child of S's scope. In the example of Fig. 7, the additional sink call `sink(u)` is inserted, where variable `u` refers to the argument of the original sink.

RemoveSourceCall. Suppose there is a source S that, according to the taint analysis, does not flow into any sink. This transformation replaces such a source with a random constant expression C

```
s = source()
t = 0
if x < 42:
  t = s
sink(t)
u = x + s
sink(u)
```

```
s = source()
t = 0
if x < 42:
  t = s
v = t   # inserted
sink(t)
u = x + s
sink(v) # transformed
```

Fig. 8. Example of the SwitchSinkArgs transformation. The original program (left) is transformed into the variant (right) by introducing an auxiliary variable `v` to store the argument of the first sink and then redirecting the second sink to use `v`. Since both sinks have the same taint classification, the two programs remain equivalent modulo taint.

of a compatible type. For example, in the snippet of Fig. 6b, the call to `source` in the original program, shown on the left of Fig. 7, is replaced with the constant `43`. Conceptually, this transformation can be generalized from constants to arbitrary expressions that do not contain nested sources or sinks.

SwitchSinkArgs. Suppose there are two sinks S_1 and S_2 such that (1) S_2 is reachable from S_1 in the control-flow graph, (2) S_2's lexical scope is either the same as or a child of S_1's scope, and (3) according to the taint analysis, both sinks have the same taint classification (either both tainted from the same source or both untainted). This transformation replaces the argument of S_2 with the argument of S_1. To ensure correctness, the argument of S_1 is first assigned to a freshly declared auxiliary variable in case subsequent statements modify the footprint of the original expression. In the example of Fig. 8, an auxiliary variable `v` is introduced to hold the argument of the first sink, and the second sink is then redirected to use `v`.

RemoveStmt. This transformation removes a random statement that is not on any witness path according to the analysis, while ensuring that the transformed program P' still compiles if the original program P does. In Fig. 9, the statement `t = t + 1` is eliminated because it does not lie on the path from source to sink. Conceptually, this transformation can be generalized from removing statements to modifying statements, as long as the modified statements do not introduce new flows.

Transformation-design rationale. Our transformations are derived from dimensions along which static taint analyzers frequently exhibit imprecision or unsoundness in practice. Each transformation systematically perturbs one or more recurring dimensions of taint analysis while preserving EMT. These dimensions include propagation through fields and collections, which often exposes imprecision in heap abstractions (e.g., AppendConstantToList); interprocedural dataflow, including taint propagation across method boundaries (e.g., InlineFnc, UseGetter); control-flow structure, where syntactic variation should not affect taint behavior (e.g., AddNestedStmt); source-sink handling, including the introduction, removal, or redirection of sources and sinks (e.g., AddNewSink, RemoveSourceCall); and taint stability under irrelevant perturbations that should not influence taint classification (e.g., DeclareNewVar, ReplaceConstant). Together, these transformations stress-test the analyzer's taint-propagation logic rather than superficial syntactic features.

```
s = source()
t = 0
if x < 42:
  t = s
sink(t)
t = t + 1 # removed
```

```
s = source()
t = 0
if x < 42:
  t = s
sink(t)
```

Fig. 9. Example of the RemoveStmt transformation. The original program (left) is transformed into the variant (right) by removing the statement `t = t + 1`, which does not lie on the witness path from source to sink. Since this removal does not affect taint propagation, the two programs remain equivalent modulo taint.

4 Experimental Evaluation

We now evaluate TaintCC to assess its effectiveness in testing static taint analyzers. Our evaluation is guided by the following research questions:

RQ1: Can TaintCC uncover real soundness and precision issues in mature taint analyzers?
RQ2: What kinds of issues does TaintCC reveal across different analyzers?
RQ3: How efficient is TaintCC?
RQ4: Which transformations participate in issue discovery?
RQ5: How does TaintCC compare to differential testing?

Together, these questions examine whether TaintCC is not only capable of finding issues, but also practical and broadly applicable across diverse analyzers.

4.1 Setup

We next describe our experimental configurations, the analyzers we test, and the hardware.

Configurations. We evaluate TaintCC under three configurations described in the following.

Issue finding. For assessing TaintCC's ability to uncover issues in RQ1, we configure it so that each generated abstract syntax tree (T_{og}) contains multiple classes and functions. During AST generation, a random number of source functions (between 1 and 10) and sink functions (also between 1 and 10) are added. In each iteration, TaintCC applies a random sequence of stacked metamorphic transformations (between 1 and 32) to T_{og}, after which the resulting program is analyzed and checked against the EMT oracle. As described earlier, the transformed tree (T_{tr}) then becomes the new original AST (T_{og}) and is subjected to further transformations. To increase diversity, every ten iterations TaintCC discards the current AST and generates a fresh random T_{og}.

Time to EMT violation. This configuration measures the efficiency of TaintCC in an issue re-finding scenario (RQ3). Ideally, such experiments would begin from a random program and continue until an issue is discovered, with the issue confirmed by re-running a fixed version of the analyzer on the buggy program and verifying that it disappears. In practice, only five of the issues we discovered have been fixed (although all were confirmed by developers), which we considered too few to support meaningful conclusions. Without fixed versions, we cannot reliably confirm the identity of a re-found issue. Instead, we approximate this setup by starting from a program that exhibits a known issue (the same one included in our public bug report) and the most recent version of the analyzer that still exhibits it, and measuring how quickly TaintCC "loses" the issue—that is, how quickly the EMT oracle breaks. If the oracle breaks, the transformed program no longer reveals the issue. Concretely, we apply a single iteration of metamorphic transformations, analyze the transformed program, and check the EMT oracle. The transformed program is then discarded, and the process repeats until the issue is "lost" ten times in total (i.e., consistently) or the 24-hour time budget is exceeded.

Controlled testing. In RQ4–5, we run TaintCC in its default configuration, where a fresh random T_{og} is generated after every ten iterations. Each testing campaign performs a total of 1000 iterations. To examine the impact of program size, we consider three categories: *small* (8 statements), *medium* (32 statements), and *large* (128 statements) freshly generated ASTs. For each size, we execute five independent campaigns with different numeric random seeds.

Issue de-duplication. During testing, TaintCC may generate many failing program variants that expose the same underlying analyzer defect. Before reporting issues to developers, we therefore de-duplicate failures via manual root-cause analysis. Concretely, we group failures that exhibit the same analyzer behavior and correspond to the same defect (e.g., in alias handling, interprocedural propagation, or source-sink modeling) as a single issue. We then confirm the uniqueness of each issue with the respective developers. All issues reported in Tab. 1 for RQ1 were confirmed by

Tab. 1. Issues uncovered by TaintCC. Each entry is identified by a unique issue ID that links to an anonymized GitHub report. The table records the affected analyzer, the current status (confirmed or fixed), and the issue type (soundness or precision).

Issue ID	Taint Analyzer	Status	Type
FD763	FlowDroid	fixed	precision
FD764	FlowDroid	confirmed	soundness
FD767	FlowDroid	fixed	soundness
FD779	FlowDroid	fixed	precision
FD781	FlowDroid	fixed	soundness
MT172	Mariana Trench	confirmed	precision
MT173	Mariana Trench	confirmed	precision
MT174	Mariana Trench	fixed	soundness
MT176	Mariana Trench	confirmed	soundness
MT179	Mariana Trench	confirmed	precision
PY923	Pysa	confirmed	soundness
PY964	Pysa	confirmed	soundness
PY977	Pysa	confirmed	precision
PY1002	Pysa	confirmed	soundness
SGPY11257	Semgrep (Python)	confirmed	precision
SGPY11376	Semgrep (Python)	confirmed	precision

developers and are unique. The counts reported elsewhere in the evaluation (i.e., in RQ4 and RQ5) are raw outcomes and are not de-duplicated.

Selection of taint analyzers. We evaluate TaintCC on four mature, actively maintained, and popular (with thousands of stars and hundreds of forks on GitHub) static taint analyzers: FlowDroid, an influential academic tool for Android and Java taint analysis; Mariana Trench, developed at Meta and used for large-scale security scanning of Android applications; Pysa, also from Meta, which analyzes large Python codebases in production (e.g., Instagram); and Semgrep, a widely used rule-based static analysis framework that supports taint tracking across multiple languages. These analyzers span both academic and industrial settings, support different languages (e.g., Java and Python), and are under active development, making them representative targets.

We intentionally focus on open-source analyzers to enable transparent experimentation, reproducibility, and interaction with developers during bug reporting. As a result, we do not include proprietary tools such as CodeQL, whose taint analysis is not publicly accessible.

Hardware. We performed all experiments on a machine with two AMD EPYC 9654 CPUs @ 2.40GHz and 1.5TB of memory, running Debian GNU/Linux 12 (bookworm).

4.2 Results

We now turn to the results guided by the research questions introduced earlier.

RQ1: Effectiveness. Tab. 1 summarizes the *unique* issues uncovered by TaintCC using the issue-finding configuration (see Sect. 4.1). Each issue is listed with an identifier and links to an anonymous GitHub report. The table also records which analyzer is affected, whether the issue is related to soundness or precision, and its current status (fixed or confirmed by the developers). Across FlowDroid, Mariana Trench, Pysa, and Semgrep, TaintCC uncovered both soundness and precision issues, several of which have already been patched by the developers, while others were acknowledged as genuine limitations. *In total, TaintCC uncovered 16 issues across all analyzers we tested. Of these, eight are soundness issues and eight are precision issues. Every issue was confirmed by the respective developers, and five have already been fixed.*

To understand why not all confirmed issues were fixed, we contacted the development teams of the analyzers. Across all tools, the common theme was *prioritization*: developers emphasized

that they must balance fixing issues against competing demands such as supporting customers or improving core functionality. For FlowDroid, priority is given to issues that block paying customers, or that have obvious or proposed fixes. For Pysa and Mariana Trench, security impact largely drives prioritization. Issues that correspond to security issues previously found manually (e.g., through bug-bounty programs or audits) but missed by the analyzers are prioritized. Lower priority is assigned to issues that involve uncommon code patterns, produce false positives, or hinge on alias analysis—a known hard problem with acknowledged limitations.

Overall, what varies across the reported issues is not whether the contradictory analyzer behavior itself is real, but how developers currently characterize it. Some issues were fixed as bugs, whereas others were treated as limitations of the analyses. In the latter cases, the limitations were undocumented and therefore still appear to users as unexpected analyzer behavior. This distinction reflects current developer prioritization rather than a difference in whether the reported issues matter in practice.

RQ2: Representative issues. In this research question, we provide an overview of representative issues that TaintCC uncovered. Fig. 10a illustrates FD779, a precision issue in FlowDroid. The code creates a list `lvar`, adds both the result of `source` and a benign Boolean value `bvar` to the list, and later passes `bvar` to `sink`. Since `bvar` is not tainted, the sink should not be flagged. However, FlowDroid incorrectly reports a leak. Interestingly, the false positive is not generated when reversing the order of lines 4 and 5. The root cause lies in FlowDroid's alias analysis: a recently introduced feature interfered with the default over-approximation of aliases, even when the feature was disabled. As a result, taint from `source` was imprecisely propagated to `bvar`, yielding a false positive. The issue has since been fixed. FD767 was already described in Sect. 2. The remaining FlowDroid issues primarily stem from misoptimizations and issues in the alias analysis.

Fig. 10b shows a precision issue in Mariana Trench. The class `MyClass` contains a field `myField`, which is initialized to the empty string. An instance of the class is created, and the field is first read into a local variable `myString`. Afterwards, `myField` is reassigned with the result of `source`, and `myString` is passed to `sink`. Since `myString` was obtained before the reassignment, it should remain untainted and the sink should not be flagged. However, Mariana Trench reports a leak. The root cause is in its alias analysis: when `myField` is reassigned, the analyzer fails to remove `myString` from the set of potential aliases, leading to spurious propagation of taint.

Fig. 10c illustrates a soundness issue in Mariana Trench. The class `MyClass` declares a field `myField` that is initialized with the result of `source` in a static initializer. In `MainActivity.onCreate`, the value of `myField` is first read into a local variable `myString`, after which `myField` is overwritten with an empty string. Finally, `myString` is passed to `sink`. Since `myField` was originally tainted by `source`, `myString` should also be tainted, and the sink should be flagged. However, Mariana Trench fails to report this leak because taint introduced through static initializers was not handled. The developers confirmed the issue and have since released a fix.

MT176 is similar to MT174, except that `myField` is directly assigned the result of `source` rather than via a static initializer. The tainted value is copied into `myString`, then `myField` is overwritten with an empty string. Although `myString` remains tainted, Mariana Trench fails to report the leak because the alias analysis incorrectly assumes the taint has been removed. The remaining issues in Mariana Trench likewise stem from alias-analysis limitations or from taint not being correctly removed across procedure boundaries.

Fig. 10d illustrates soundness issue PY1002 in Pysa. The method `my_function` creates a list using `MyClass.init`, appends the result of `source`, and then passes the first element of the list to `sink`. Since the list contains tainted data, the sink should be reported. However, Pysa fails to flag the leak when `init` lacks a return type annotation. Without the annotation, Pysa cannot infer the type of `lst`. As a result, when analyzing `lst.append(self.source())`, it does not know which function is

```java
List<Boolean> lvar =
    new ArrayList<Boolean>();
Boolean bvar = true;
lvar.add(source());
lvar.add(bvar);
sink(bvar); // false positive
```

(a) Precision issue FD779 in FlowDroid.

```java
MyClass myInstance = new MyClass();
String myString = myInstance.myField;
myInstance.myField = source();
sink(myString); // false positive
```

(b) Precision issue MT173 in Mariana Trench.

```java
public class MainActivity
      extends AppCompatActivity {
  public void onCreate(
        Bundle savedInstanceState) {
    MyClass myInstance = new MyClass();
    String myString = myInstance.myField;
    myInstance.myField = "";
    sink(myString); // false negative
  }

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

  public void sink(String param) { }
}

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

(c) Soundness issue MT174 in Mariana Trench.

```python
class MyClass:
  def my_function(self):
    lst = MyClass.init([])
    lst.append(self.source())
    # false negative
    self.sink(lst[0])

  def source(self):
    return "Secret"

  def sink(self, param: str):
    return param

  @staticmethod
  def init(listi: list[str]):
    return listi
```

(d) Soundness issue PY1002 in Pysa.

```python
original: list[str] = []
copy: list[str] = []
copy = original
copy.append(source())
# false negative
sink(original[0])
sink(copy[0])
```

(e) Soundness issue PY923 in Pysa.

```python
my_instance = MyClass()
my_instance.attribute = source()
sanitize(my_instance)
# false positive
sink(my_instance.attribute)
```

(f) Precision issue PY977 in Pysa.

Fig. 10. Representative issues uncovered by TaintCC (RQ2).

called and falls back to a heuristic: for unknown calls, it assumes taint on arguments is returned, but not that taint is assigned to `self`. This heuristic leads to the false negative. Adding an explicit return type annotation to `init` enables Pysa to correctly infer the list type and detect the issue. Issue PY964 likewise stems from limitations in type inference.

Fig. 10e shows soundness issue PY923 in Pysa. The snippet creates a list `original`, creates the alias `copy` by copying the reference, and appends tainted data through the alias. Since both `original` and `copy` point to the same list, the sinks in both calls should be reported. Pysa, however, only flags the sink through `copy`, producing a soundness issue. The root cause is an alias-handling gap for collection elements: taint added via `copy.append(source())` propagates to reads through `copy`, but not to reads through the original reference `original`.

Fig. 10f shows precision issue PY977 in Pysa. The snippet first creates an object and assigns a tainted value from `source` to its `attribute`. It then calls a function `sanitize`, which simply overwrites the attribute with an empty string. Finally, the attribute is passed to `sink`. In this case,

Tab. 2. Results of the time-to-EMT-violation experiment. For each issue, we report how often TaintCC was able to "lose" the issue, the median time until the EMT-oracle violation, the total number of iterations performed, and the total campaign duration.

Issue ID	# Lost	Median time	# Iters	Total time
FD763	10	00h 00m 22s	27	00h 03m 28s
FD764	3	01h 01m 52s	11023	24h 00m 19s
FD767	10	00h 00m 17s	32	00h 03m 40s
FD779	10	00h 01m 58s	268	00h 32m 00s
FD781	10	00h 00m 43s	91	00h 10m 18s
MT172	10	00h 27m 27s	118	03h 45m 15s
MT173	10	00h 04m 30s	35	01h 05m 02s
MT174	10	00h 04m 07s	34	00h 48m 29s
MT176	10	00h 08m 02s	79	02h 25m 36s
MT179	10	00h 04m 32s	23	00h 43m 12s
PY923	10	00h 15m 16s	54	02h 32m 59s
PY964	10	00h 14m 52s	67	03h 03m 54s
PY977	10	00h 08m 05s	56	02h 34m 09s
PY1002	10	00h 16m 19s	70	03h 13m 10s
SGPY11257	10	00h 00m 11s	55	00h 02m 20s
SGPY11376	10	00h 00m 05s	26	00h 01m 08s

Pysa incorrectly reports a leak even though the taint was removed. Interestingly, if the sanitizer is inlined directly (i.e., replacing the call with `my_instance.attribute = ""`), Pysa does not report the leak. This discrepancy indicates that Pysa's modeling of procedure calls during taint removal is imprecise: while overwriting is recognized when done in place, overwriting through a helper function fails to suppress the taint, leading to a false positive.

Semgrep exhibits two precision issues related to taint removal and propagation. In SGPY11257, Semgrep reports a false positive when analyzing `try-except-finally` constructs in Python: if a tainted variable is reassigned a constant in both the `try` and `except` branches, and the `finally` block is empty, the reassignment should remove the taint. However, when the `try` block contains at least one method call preceding the reassignment, Semgrep fails to clear the taint. A second precision issue, SGPY11376, arises because the analyzer does not update the taint source when a tainted variable is reassigned another tainted value. While both issues were uncovered in Python, SGPY11376 also manifests in Java.

RQ3: Efficiency. We evaluate the efficiency of TaintCC by measuring how frequently (in number of iterations) and how quickly (in running time) it can violate the EMT oracle on programs for which TaintCC had previously uncovered issues. To this end, we use the time-to-EMT-violation configuration (Sect. 4.1).

Tab. 2 summarizes the results. It shows, for each issue, how often TaintCC was able to "lose" the issue before the termination condition, the median time until a violation, the total number of iterations, and duration of the campaign. Overall, TaintCC consistently violated the EMT oracle within 24 hours for 15 out of 16 issues. For FlowDroid, violations typically occurred within 2 minutes, except for FD764, which only appeared under a certain flag configuration and required over 1 hour. For Mariana Trench, violations occurred in median under 28 minutes, for Pysa under 17 minutes, and for Semgrep under 11 seconds. The iteration counts confirm this picture: when issues were lost quickly, only a handful of iterations were needed, while harder-to-lose issues required many more attempts.

On average across all issues, TaintCC executed 6.62 iterations per minute. Importantly, between 79% (FlowDroid) and 99.9% (Pysa) of runtime was spent inside the analyzers, not in TaintCC itself. This shows that throughput is primarily bounded by analysis speed rather than our tool.

RQ4: Transformation participation in issue discovery. In this research question, we examine how individual transformations participate in issue discovery. Our goal is not to attribute issues to single transformations in isolation, but to understand whether certain transformations are systematically involved in exposing issues. We use the controlled-testing configuration (see Sect. 4.1). For each transformation, we measure its likelihood of appearing in an iteration that leads to an EMT-oracle violation versus one that does not. Concretely, we record all transformations applied during issue-revealing and non-issue-revealing iterations across all campaigns and analyzers. We then compute, for each transformation, the probability of occurrence conditioned on an iteration being issue-revealing (respectively, non-issue-revealing), defined as the fraction of that transformation's occurrences among all transformations observed in issue-revealing (respectively, non-issue-revealing) iterations.

Fig. 11 plots these probabilities. Overall, we find that transformations are equally likely to appear in issue-revealing and non-issue-revealing iterations, indicating that no subset of transformations dominates issue discovery. Instead, EMT-oracle violations arise from diverse transformation sequences, and all transformations contribute. This suggests that issues in taint analyzers are not triggered by isolated syntactic changes, but by interactions across multiple dimensions of taint propagation.

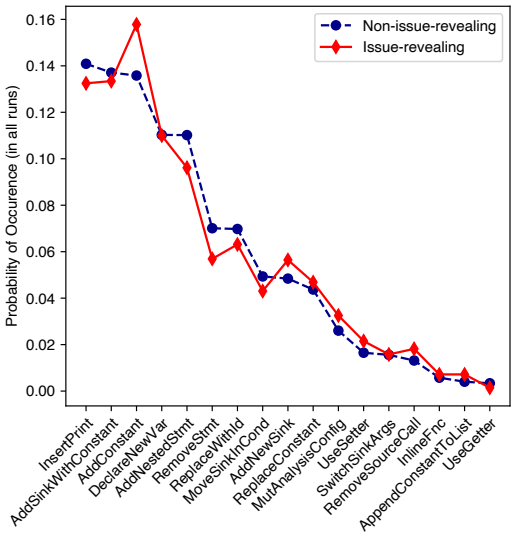


Fig. 11. Probability of each transformation occurring in issue-revealing vs. non-issue-revealing iterations across all campaigns and analyzers, with similar frequencies in both settings.

More specifically, across all analyzers, differences in transformation frequency between issue-revealing and non-issue-revealing iterations are below 2%. Some transformations nonetheless occur slightly more often in issue-revealing iterations, most notably AddConstant and AddNewSink. Importantly, these patterns are analyzer-specific: AddConstant is most associated with issues in Mariana Trench and Pysa, DeclareNewVar and AddNewSink in Semgrep, and no transformation stands out for FlowDroid. By contrast, transformations with narrow applicability constraints (e.g., AppendConstantToList, which requires a list variable) appear less frequently overall, while transformations with broad applicability but limited semantic impact (e.g., InsertPrint) occur frequently yet show little association with issue discovery.

We also perform an ablation study to quantify how many issues are found by different transformation groups, thereby complementing the per-transformation participation analysis above. We compare five settings: TaintCC with all EMT transformations enabled; TaintCC restricted to semantically equivalent transformations (SEQ); TaintCC restricted to taint-oblivious transformations (TO); TaintCC restricted to taint-aware transformations (TA); and TaintCC using taint-oblivious and taint-aware transformations together (TO+TA).

Using TaintCC's controlled-testing configuration (see Sect. 4.1), we measure the average number of EMT-oracle violations across five runs per setting. Tab. 3 reports the results for each analyzer and AST size. The upper rows report the TaintCC configurations above. (The lower rows report Statfier configurations [35] on our Java analyzers, which we discuss separately below.) Across all analyzers and AST sizes, no single transformation group dominates uniformly. The strongest results

Tab. 3. For each AST size (S, M, L), the table reports the average number of EMT-oracle violations across five campaigns of 1000 tests under eight configurations. For TaintCC, we report all transformations enabled (ALL), only semantically equivalent transformations (SEQ), only taint-oblivious transformations (TO), only taint-aware transformations (TA), and taint-oblivious plus taint-aware transformations (TO+TA). For Statfier, we report random locations (RND), guided locations (GUI), and guided locations with `CFWrapperWithIfFalse` disabled (GUI-noIFF). Because the implemented Statfier artifact only supports Java analyzers, its results are reported only for FlowDroid, Mariana Trench, and Semgrep (Java); entries marked with – are not applicable. Boldface indicates the highest number of violations for each AST size.

Config.	FlowDroid			Mariana Trench			Pysa			Semgrep (Python)			Semgrep (Java)		
	S	**M**	**L**	**S**	**M**	**L**	**S**	**M**	**L**	**S**	**M**	**L**	**S**	**M**	**L**
TaintCC															
ALL	0.2	1.6	7.0	2.8	3.6	13.8	0.0	**0.6**	2.0	**0.2**	**0.4**	0.2	**0.4**	**0.6**	2.6
SEQ	0.0	1.2	4.4	0.0	0.2	0.2	**0.2**	0.0	1.0	0.0	0.2	1.6	0.0	0.2	2.0
TO	0.0	**2.4**	**8.4**	0.0	0.0	0.0	0.0	0.2	**3.8**	**0.2**	0.2	2.0	0.2	0.2	2.0
TA	0.0	0.4	3.6	0.0	0.0	0.0	0.0	0.0	0.2	**0.2**	**0.4**	**4.0**	0.2	0.4	**3.8**
TO+TA	**0.4**	0.6	7.8	**4.0**	**7.8**	**27.6**	**0.2**	0.2	2.0	0.0	**0.4**	0.0	0.0	0.4	1.2
Statfier															
RND	0.0	0.0	0.0	0.0	0.0	0.0	–	–	–	–	–	–	0.0	0.0	0.0
GUI	0.0	0.0	0.0	0.0	0.0	0.0	–	–	–	–	–	–	**99.8**	**100.0**	**99.6**
GUI-noIFF	0.0	0.0	0.0	0.0	0.0	0.0	–	–	–	–	–	–	0.0	0.0	0.0

are obtained by the full EMT setting and by taint-specific transformations (TO, TA, or TO+TA), depending on the analyzer and program size. In contrast, SEQ is best only in one case (tied with TO+TA for Pysa on small ASTs) and is otherwise consistently outperformed. These results show that many issues exposed by TaintCC require taint-specific transformations. In some cases, TO or TA alone is sufficient, while in others their combination (TO+TA) or the full EMT setting is strongest. By contrast, semantically equivalent transformations alone are rarely the most effective.

An exhaustive ablation over all subsets of transformations would be infeasible due to the combinatorial explosion and would poorly reflect practice, as issues typically arise from interactions across transformations rather than isolated effects.

To connect this result more directly to prior work, we also compare against Statfier [35], which applies semantically equivalent transformations to test static analyzers, although it does not target taint analyzers. Since Statfier's artifact only supports Java analyzers, we restrict this comparison to our Java analyzers. Moreover, because Statfier requires seed files as input whereas TaintCC generates fresh programs, we use TaintCC-generated Java programs as seeds in this comparison. We align Statfier's configuration with the same controlled-testing settings used for the TaintCC configurations above: we disable bug de-duplication, use search depth 10, and use random locations by default. Here, search depth 10 means that Statfier applies up to 10 successive transformations to a seed before restarting from a fresh one, and random locations means that mutation sites are selected uniformly rather than biased toward locations mentioned in analyzer reports. We also exclude six transformations from the artifact, which we found do not preserve semantic equivalence in general: `CFWrapperWithDoWhile`, `CFWrapperWithForTrue1`, `CFWrapperWithForTrue2`, `CFWrapperWithWhileTrue`, `LoopConversion1`, and `LoopConversion2`. The corresponding averaged results are reported in the lower rows of Tab. 3.

As shown there, under random locations (RND), Statfier found no issues for FlowDroid, Mariana Trench, or Semgrep (Java). We then enabled Statfier's guided-location heuristic (GUI), which biases mutation toward locations mentioned in the analyzer's reports and their backward slices. In that setting, Statfier repeatedly exposed the same Semgrep precision issue, always via `CFWrapperWithIfFalse`. This transformation copies a statement and wraps the copy

Tab. 4. Comparison of differential and metamorphic testing. For each AST size (S, M, L), the table reports the number of oracle violations across five campaigns and the average throughput. Differential testing (second row) compares analyzers jointly, and metamorphic testing in TaintCC (other rows) tests analyzers individually.

	Reset after	# Reports			Throughput		
		S	M	L	S	M	L
Differential testing	1	146.2	427	829.8	0.16	0.16	0.15
FlowDroid	1	0.2	2.4	9.6	4.72	4.71	4.47
Mariana Trench	1	3.8	4	17.2	0.42	0.42	0.42
Pysa	1	0.6	0.4	1.2	0.96	0.97	0.83
Semgrep (Python)	1	0	0.8	0.6	14.65	14.43	14.25
Semgrep (Java)	1	0	0.4	0.8	14.55	14.40	13.95
FlowDroid	10	0.2	1.6	7	7.56	7.45	7.16
Mariana Trench	10	2.8	3.6	13.8	0.79	0.79	0.78
Pysa	10	0	0.6	2	0.43	0.43	0.42
Semgrep (Python)	10	0.2	0.4	0.2	24.14	23.74	23.30
Semgrep (Java)	10	0.4	0.6	2.6	23.51	24.23	22.78

in `if (false)`. When the copied statement is a sink to which taint flows, Semgrep reports a false positive for the unreachable copy. All discovered results corresponded to this single issue, which the developers closed as not planned[1]. This issue is limited in what it tells us in our setting: any control-flow-insensitive analyzer would exhibit it, whereas the other analyzers compile the unreachable code away. Disabling `CFWrapperWithIfFalse` (GUI-noIFF) reduced the number of issues that Statfier uncovered to zero. These results reinforce the conclusion from Tab. 3: semantic-preserving transformations alone are not enough to expose many of the issues that TaintCC finds.

RQ5: Comparison to differential testing. In this research question, we compare metamorphic and differential testing. For differential testing, we generate a random AST, translate it into both Python and Java using TaintCC, and then run all analyzers on the resulting programs. Although we evaluate four analyzers overall, we treat Semgrep's Python and Java analyses separately in this experiment, since their analysis results may differ across languages. If the analyzer outputs disagree, a potential issue is reported. Such disagreements may indicate genuine defects, but they can also arise from intentional differences in modeling choices or precision between analyzers. For TaintCC, we use the controlled-testing configuration (see Sect. 4.1). To ensure a fair comparison, we run TaintCC under both its default setting—generating a fresh T_{og} every ten iterations—and an alternative setting that generates a new T_{og} in every iteration, mirroring differential testing where each test starts from a fresh random program.

Tab. 4 summarizes the average test throughput and the number of potential soundness and precision issues reported by differential and metamorphic testing across five campaigns for each initial AST size. The second row reports differential-testing results obtained by jointly running all analyzers on the same programs. The remaining rows report TaintCC results for each analyzer tested individually. Rows 8–12 correspond to TaintCC's default configuration, where a fresh T_{og} is generated after every ten iterations, whereas rows 3–7 correspond to an alternative configuration in which a new T_{og} is generated in every iteration, mirroring differential testing.

Our results show that differential testing predictably reports far more potential issues, many of which reflect differences in modeling choices or precision across analyzers and are therefore typically classified as "won't fix" by developers. These require significantly more manual effort

[1] https://github.com/semgrep/semgrep/issues/11661

to review than the issues uncovered by TaintCC. EMT violations, by comparison, reflect self-inconsistencies within a single analyzer: the analyzer produces different results for two equivalent-modulo-taint programs. This substantially reduces triage effort and increases the likelihood that a reported violation corresponds to a genuine analyzer issue rather than an intentional design choice. Differential testing is also slow: throughput is limited by the slowest analyzer and on average is only 0.16 tests per minute. In contrast, TaintCC achieves up to 150x higher throughput on Semgrep compared to testing jointly with the other analyzers.

Finally, we observed that TaintCC's default setting—generating a new T_{og} every ten iterations—achieves higher throughput than resetting the AST in every iteration. The reason is that the default setting allows reusing analysis results: in ten iterations, the analyzer is run eleven times, whereas generating a fresh T_{og} each time requires two runs per iteration (for P_{og} and P_{tr}).

4.3 Threats to Validity

Our results are subject to the following threats to validity. First, we evaluated TaintCC on four analyzers—FlowDroid, Mariana Trench, Pysa, and Semgrep—across Python and Java. While these tools are mature and diverse, the findings may not generalize to all taint analyzers or programming languages. Second, our classification of issues as related to soundness or precision, as well as our understanding of their root causes, could be mistaken. We mitigated this risk by confirming all cases with developers. Third, our transformations were designed to target recurring difficulty dimensions in static taint analysis, as discussed in Sect. 3.2. However, they are not guaranteed to be comprehensive: although they cover a broad range of taint-analysis behaviors, additional transformations may expose further issues. Finally, our experiments rely on randomness in AST generation and transformation sequences; to reduce this threat, we ran long campaigns, used different numeric random seeds, and reported aggregated results.

5 Related Work

Testing taint analyzers. Bonett et al. proposed μSE [5], which evaluates taint analyzers by injecting synthetic leaks into real Android applications and checking whether these leaks are detected. This approach is effective for exposing soundness issues in Android-specific settings, particularly those related to lifecycle modeling. In contrast, TaintCC does not inject new leaks but instead tests analyzers for internal consistency under EMT. This enables the detection of both soundness and precision issues and allows TaintCC to apply uniformly across languages and analyzers beyond Android. The two approaches are complementary: μSE emphasizes realism through domain-specific mutation, whereas TaintCC emphasizes generality through a taint-specific metamorphic oracle.

Testing compilers and program analyzers. In Sect. 1, we already discuss the most closely related work on fuzzing for compilers and program analyzers. Recent work has also focused on detecting faults in internal program representations of static-analysis frameworks [37]. These efforts highlight the value of automated testing for complex analysis infrastructure but differ in their choice of oracles.

The key novelty of TaintCC lies in its use of EMT. Prior work on metamorphic testing has largely focused on transformations that preserve full program semantics, or—less commonly—on transformations that induce a subset relation between program outputs O and O' (e.g., $O \subseteq O'$ or $O' \subseteq O$). For example, Statfier [35] applies semantically equivalent transformations to test static analyzers, while queryFuzz [23] explores transformations that preserve, expand, and contract program outputs to uncover issues in Datalog engines. These oracles are well suited to their respective settings, but they do not directly capture the source-sink flow property that taint analyzers compute. This property admits a much weaker, taint-specific equivalence notion, captured by EMT.

Equivalence modulo inputs (EMI) [21] introduces a different oracle by comparing two compiled programs only with respect to a specific input, thereby enabling the discovery of compiler miscompilations. While powerful in its domain, EMI does not transfer naturally to static analyzers, which are designed to reason over all possible inputs.

In contrast, EMT makes the analyzer central to the oracle: two programs are considered equivalent if the tested analyzer reports the same set of source-sink flows. TaintCC demonstrates how EMT can systematically uncover both soundness and precision issues in static taint analyzers. Exploring whether a similar oracle can benefit other analyses (e.g., typestate) is a promising future direction.

TaintCC draws inspiration from interrogation testing [17, 18]. Its taint-aware transformations use the analysis results for the original program P to construct transformed programs P'. This design enables TaintCC to reveal issues in an analyzer's reasoning by checking whether two equivalent-modulo-taint programs yield inconsistent results.

6 Conclusion

We presented TaintCC, the first testing framework for static taint analyzers based on a domain-specific metamorphic oracle. Its key idea is *equivalence modulo taint*: two programs are considered equivalent if the analyzer reports the same source-sink flows, regardless of other semantic differences. By preserving only the property of interest for taint analysis, EMT provides the weakest equivalence sufficient for systematic testing. Using EMT as an oracle, TaintCC generates program variants via semantically equivalent, taint-oblivious, and taint-aware transformations, and exposes issues when analyzer results diverge.

Our evaluation on four mature and widely used analyzers—FlowDroid, Mariana Trench, Pysa, and Semgrep—demonstrates that TaintCC is effective and practical. In total, it uncovered 16 previously unknown issues (eight soundness, eight precision), all confirmed by the developers, with several already fixed. Beyond issue finding, our experiments showed that TaintCC can consistently expose contradictions within seconds to hours, with throughput largely determined by the analyzers themselves rather than our framework.

These results show that even industrial-strength taint analyzers remain vulnerable to subtle soundness and precision issues, and that EMT provides a lightweight yet powerful oracle for exposing such inconsistencies.

Data Availability

Our tool is publicly available at https://github.com/Rigorous-Software-Engineering/TaintCC and archived at https://doi.org/10.5281/zenodo.21266656.

Acknowledgments

We thank the static taint analyzer developers for their valuable work and assistance. This work was supported by Maria Christakis' ERC Starting grant 101076510.

References

[1] [n. d.]. Mariana Trench. https://mariana-tren.ch.
[2] [n. d.]. Pysa. https://pyre-check.org/docs/pysa-basics.
[3] [n. d.]. Semgrep. https://semgrep.dev.
[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *PLDI*. ACM, 259–269.
[5] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering Flaws in Security-Focused Static Analysis Tools for Android Using Systematic Mutation. In *Security*. USENIX, 1263–1280.
[6] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. ACM, 1459–1470.

[7] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.
[8] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surv.* 53 (2020), 4:1–4:36. Issue 1.
[9] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST–CS98–01. HKUST.
[10] Maria Christakis, Hasan Ferit Eniser, Jörg Hoffmann, Adish Singla, and Valentin Wüstholz. 2023. Specifying and Testing k-Safety Properties for Machine-Learning Models. In *IJCAI*. ijcai.org, 4748–4757.
[11] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *CACM* 20 (1977), 504–513. Issue 7.
[12] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ISSTA*. ACM, 1219–1231.
[13] Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Constraint-Based Test Oracles for Program Analyzers. In *ASE*. ACM, 344–355.
[14] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *PACMSE* 1 (2024), 1656–1678. Issue FSE.
[15] Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2025. Fuzzing Processing Pipelines for Zero-Knowledge Circuits. In *CCS*. ACM, 783–797.
[16] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamaric, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (experience paper). In *ISSTA*. ACM, 556–567.
[17] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Interrogation Testing of Program Analyzers for Soundness and Precision Issues. In *ASE*. ACM, 319–330.
[18] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2026. Interrogation Testing of CHC Solvers. In *FSE*. ACM. To appear.
[19] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *ASE*. IEEE Computer Society, 590–600.
[20] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.
[21] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*. ACM, 216–226.
[22] Francesco Logozzo, Manuel Fähndrich, Ibrahim Mosaad, and Pieter Hooimeijer. 2019. Zoncolan: How Facebook Uses Static Analysis to Detect and Prevent Security Issues. https://engineering.fb.com/2019/08/15/security/zoncolan.
[23] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic Testing of Datalog Engines. In *ESEC/FSE*. ACM, 639–650.
[24] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE*. ACM, 701–712.
[25] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA*. ACM, 236–247.
[26] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.
[27] Jan Midtgaard and Anders Møller. 2017. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.* 27 (2017). Issue 6.
[28] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *ICSE*. IEEE Computer Society, 550–562.
[29] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *PACMPL* 5 (2021), 1–19. Issue OOPSLA.
[30] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *TSE* 42 (2016), 805–824. Issue 9.
[31] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.
[32] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *PACMPL* 4 (2020), 193:1–193:25. Issue OOPSLA.
[33] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI*. ACM, 718–730.
[34] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.
[35] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *ESEC/FSE*. ACM, 237–249.

[36] Huaien Zhang, Yu Pei, Shuyun Liang, and Shin Hwei Tan. 2024. Understanding and Detecting Annotation-Induced Faults of Static Analyzers. *PACMSE* 1 (2024), 722–744. Issue FSE.
[37] Huaien Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, and Shin Hwei Tan. 2024. Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks. In *ISSTA*. ACM, 1772–1784.

Received 2026-01-29; accepted 2026-06-25