ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis

Austin Mordahl Zenong Zhang Dakota Soles Shiyi Wei
Department of Computer Science
The University of Texas at Dallas
Richardson, TX, USA
{austin.mordahl, zenong, dakota.soles, swei}@utdallas.edu

***Abstract*—Testing and debugging the implementation of static analysis is a challenging task, often involving significant manual effort from domain experts in a tedious and unprincipled process. In this work, we propose an approach that greatly improves the automation of this process for static analyzers with configuration options. At the core of our approach is the novel adaptation of the theoretical partial order relations that exist between these options to reason about the correctness of actual results from running the static analyzer with different configurations. This allows for automated testing of static analyzers with clearly defined oracles, followed by automated delta debugging, even in cases where ground truths are not defined over the input programs. To apply this approach to many static analysis tools, we design and implement *ECSTATIC*, an easy-to-extend, open-source framework. We have integrated four popular static analysis tools, SOOT, WALA, DOOP, and FlowDroid, into *ECSTATIC*. Our evaluation shows running *ECSTATIC* detects 74 partial order bugs in the four tools and produces reduced bug-inducing programs to assist debugging. We reported 42 bugs; in all cases where we received responses, the tool developers confirmed the reported tool behavior was unintended. So far, three bugs have been fixed and there are ongoing discussions to fix more.**

***Index Terms*—Program analysis, testing and debugging**

I. Introduction

Static analysis is a useful tool to discover software bugs. But there is no one-size-fits-all static analysis that can handle all types of target programs. Thus, many well-known static analyzers (e.g., FlowDroid [1], SOOT [2], DOOP [3], and WALA [4]) implement configuration options to allow the developers or users to tune the analysis in order to achieve the sweet spot between precision, soundness, and scalability on their target programs.

However, the resulting large configuration space also makes it more difficult to ensure the correctness of the analysis implementations. A key challenge towards automated testing of static analyzers is the lack of an easy-to-obtain *oracle*. Each configuration of a static analyzer may be expected to produce different results analyzing the same program. This means that the tool developers have to manually confirm the expected results for each configuration on every test program, which is infeasible. Indeed, a recent study [5] shows that implementations of analysis options are poorly tested, finding potential bugs in multiple configurations of FlowDroid.

These potential bugs were discovered thanks to the definition of *partial orders* of analysis options, i.e., relations between the settings of an analysis option that specify the expected behavior of the tool with regard to the true and false positives reported [5]. These partial orders make up for the lack of an oracle for various configuration options, by allowing issues to be discovered by comparing two configurations that differ only by a single option setting and finding *violations* of expected behavior, which are indicative of bugs in the analysis tool. While being a promising direction, their work is limited in improving the reliability of static analysis implementations. First, there is no evidence that their experiments using manually constructed configurations on two Android taint analysis tools can be automated and extended to other static analyzers. Second, there is no confirmation if the detected violations of partial orders are indeed bugs and if/how they can be useful for debugging. Third, their experiments require input programs with ground truths or classified results, which may not be available for many analysis clients.

In this work, we present a new general approach and framework that automatically tests and debugs configurable static analysis. Our key idea is to leverage the knowledge of tool configurations and partial orders for the *automation of test generation, bug identification, and debugging*. We propose novel partial order aware testing and delta debugging approaches, and extend the definition of a partial order and a violation to allow bug finding *without ground truths* (Section III). Our approach takes as inputs (1) a grammar that lists the tool options and their settings, (2) a specification of the partial orders, and (3) a set of programs with (e.g., DroidBench [6]) or without (e.g., DaCapo [7]) ground truths.

To detect implementation bugs in tool configurations, we propose a two-staged testing approach. First, based on a tool's default configuration, we create all partially-ordered configurations (i.e., configurations that differ only by a single option setting) that are defined in the partial order specification. These configurations are run on all the input programs to detect violations of defined partial orders. The second stage randomly and iteratively generates and tests configurations to detect new partial order bugs that only exist under certain option interactions. This stage consists of four steps: (1) A grammar-based fuzzer is used to generate seed configurations, which are randomly selected for testing. (2) The seed configuration is mutated to create partially-ordered configurations based on a set of partial orders which have not yet exhibited a violation

on any input program. (3) The mutated configurations are run on a set of input programs to detect partial order violations. (4) The violation detection results are used as feedback to remove partial orders that have exhibited violation(s). Steps (1)-(4) are repeated until a given timeout is reached. We implement and evaluate two variations of this random testing stage that differ in how they select partial orders and input programs. One, at each step, randomly samples a small number of partial orders and input programs, making it lightweight and suitable for testing many partially-ordered configurations based on different seeds. The second is a more exhaustive approach, using all partial orders and input programs at each iteration.

To assist the debugging of each partial order violation, we develop a novel violation-aware delta debugger that reduces the input programs into bug-inducing features. Our delta debugger is novel in two aspects. First, it is the first to adapt delta debugging to metamorphic testing of static analyzers. At each delta debugging iteration, our approach attempts to run two partially-ordered configurations of an analysis tool on a reduced program and detect whether the violation still exists. Second, to address the challenges in efficiency (i.e., expensive compilation and analysis passes), our delta debugger runs in two passes: first, a coarse-grained class-level reduction [8], and then hierarchical delta debugging [9] based on an abstract syntax tree.

To allow the proposed approach to be generally applied to many static analysis tools, we develop an open-source framework, *ECSTATIC* (Section IV). We design *ECSTATIC* with the goals of producing an easy-to-use, scalable, reproducible, and highly extensible framework. We achieve these goals by (1) defining clear interfaces for specifying tools and input programs such that a new tool or input program can typically be integrated by writing only dozens of lines of code, (2) structuring the code so that each phase can be run in parallel, and (3) containerizing each analysis tool inside a Docker [10] container for reproducible environments. *ECSTATIC* is available at https://doi.org/10.5281/zenodo.7577909.

We integrate four popular static analysis tools (FlowDroid, SOOT, WALA, and DOOP) and four benchmarks (DaCapo, DroidBench, CATS [11], and FossDroid [12]) into *ECSTATIC*. Running *ECSTATIC*, we detect 74 partial order bugs across the four tools. While most of these bugs were found in the first testing stage, the two variants of the random testing stage were able to collectively detect 10 new partial order bugs. The violation-aware delta debugger was able to reduce input programs to as little as 1% of their original sizes, with an average of 50% reduction on real-world programs. We also reported some of these bugs to tool developers, and in every case where the developers responded, received confirmation that our approach uncovered unexpected behavior, leading to three bug fixes in FlowDroid and ongoing discussion regarding a fix for WALA.

This paper made the following contributions:

- A new general approach that provides automated support for tool developers to test and debug static analysis, leveraging the knowledge of tool configurations and option partial orders throughout its design.

```
1  // InterproceduralConstantValuePropagator.java
2  protected void internalTransform(...) {
3    ...
4    if (removeSideEffectFreeMethods) {
5      ....
6        boolean remove = callee.getReturnType() == VoidType
       .v() && !hasSideEffectsOrReadsThis(callee);
7        - remove |= !hasSideEffectsOrCallsSink(callee);
8        + remove &= !hasSideEffectsOrCallsSink(callee);
9        if (remove) {
10           Scene.v().getCallGraph().removeEdge(edge);
```

Fig. 1: Excerpt of FlowDroid code that shows an implementation error in its *codeelimination* option, and the fix the tool developer made using *ECSTATIC*'s bug reports [13].

- An open-source, extensible framework, *ECSTATIC*, that enables easy integration of new analysis tools and benchmarks for configuration aware testing and debugging.
- The integration of four popular tools and four benchmarks in *ECSTATIC*, and an evaluation on the performance of our approach, showing its effectiveness of detecting actual bugs and producing reduced bug-inducing programs.

II. Background and Motivation

In this section, we first use an example to illustrate the challenges when testing and debugging a static analyzer. We then introduce the background of a key concept, *partial orders of analysis options*, which we adapted and improved upon to mitigate these challenges.

A. Motivating Example

We use an example from FlowDroid to illustrate the challenges of detecting and debugging bugs in static analysis tools. This bug was detected by *ECSTATIC*, reported to the tool maintainer, and is fixed in the current version of the tool [13]. The bug extracted in Figure 1 shows a logic error in the implementation of FlowDroid's *codeelimination* option. The default setting of this option, *PROPAGATECONSTS*, performs constant propagation. The *REMOVECODE* setting of this option, in addition to constant propagation, performs a pre-analysis to remove any methods from the call graph that (1) have a *void* return type, (2) do not refer to the *this* variable, (3) do not have side effects, and (4) do not call a sink method. This should be a sound optimization because methods satisfying all four conditions should not affect the detection of taint flows.

However, there was a logic error in its implementation. In line 6 of Figure 1, the boolean variable *remove* is used to determine if a method (i.e., *callee*) should be removed from the call graph. Line 6 correctly implements the logic to satisfy the first three conditions to remove a method by checking the return type and calling the method *hasSideEffectsOrReadsThis*. In line 7, it calls the method *hasSideEffectsOrCallsSink*, to decide if the fourth condition–"do not call a sink method"–is satisfied. This method also rechecks the value of *hasSideEffects* as computed by *hasSideEffectsOrCallsSink*. However, by using the OR operator `|=`, it can remove a method that only meets conditions (1), (2), (3) or only meets conditions (3) and (4).

Instead, the AND operator `&=` should be used (as shown in the fix in line 8). This bug is useful to demonstrate several challenges in testing and debugging a static analysis tool.

Challenge 1: This erroneous code may be executed only in some configurations of FlowDroid. In line 4, the field *removeSideEffectFreeMethods* guards the execution of the code that contains the logic error. This field is only set to true if the configuration sets the *codeelimination* option to *REMOVECODE* and does not set the *implicit* option to *ALL*. This suggests that the tool developers potentially need to test many configurations of a static analysis. This is almost impossible to achieve, as testing the correctness of any one configuration is a daunting task (see below).

Challenge 2: Appropriate input programs with *oracles* are not easy to obtain when testing a static analysis tool. To catch the above error, FlowDroid, using a configuration that can execute the erroneous code, needs to be run on an input program that contains a method that meets only conditions (1)-(3) or (3)-(4). Indeed, when searching DroidBench, the most popular benchmark for Android taint analysis, we found only 1 out of 190 programs (*ActivityLifecycle1*) contains a method that would allow the detection of this logic error.

Challenge 3: Debugging to understand the cause of an implementation error in a static analyzer is a challenging task. While researchers have developed some tools to support such a task [14], [15], it still involves extensive manual efforts. For the example in Figure 1, after observing that FlowDroid unexpectedly does not produce a tainted flow in an input program (e.g., *ActivityLifecycle1*), the developer is likely to find that there is a missing call graph edge to the method that calls the sink method. However, the task of determining the cause of the missing edge requires the developer to understand both FlowDroid's code and the input program, requiring the developers to inspect potentially large codebases with their domain knowledge.

B. Partial Orders of Analysis Options

We aim to mitigate the above challenges through an approach that greatly improves the automation of the testing and debugging of static analysis. At the core of our idea is the adaptation of a concept called *analysis options partial orders*. The idea of using partial orders to describe the relationship between settings within a configuration option of static analysis tools was recently presented by Mordahl and Wei [5], which they use to study the impact of configuration options on the results of two Android taint analysis tools.

The basic idea of these partial order relationships is that increasing the soundness of an analysis should not remove true positives and increasing the precision of an analysis should not introduce new false positives, compared to a less sound/precise configuration, respectively. Let C be a configuration of an analysis tool, and let $C[o_i]$ return the setting of option o_i in C. Given two configurations C_1 and C_2, We say that $C_1 \sqsubseteq_S C_2$ (read as "C_1 is at least as sound as C_2") if (1) there exists one and only one option, o_i, such that $C_1[o_i] \neq C_2[o_i]$; and (2) $C_1[o_i]$ implements an analysis that is expected to be at least as sound as that implemented by $C_2[o_i]$ (alternatively written in terms of the settings $C_1[o_i] \sqsubseteq_S C_2[o_i]$). For precision partial orders (denoted $\sqsubseteq_P$), the definition is the same except that "sound" is replaced with "precise" in (2).

Using these definitions, Mordahl and Wei found potential bugs in two Android taint analysis tools by comparing the results of different configurations. Their idea is an instantiation of *metamorphic testing* [16], [17] using these partial orders. Metamorphic testing is an approach in which a known relation between the outputs of related inputs to some piece of software is used as an oracle for testing. For our situation, let $f(C_i, p)$ be the set of results obtained by running an analysis tool with configuration C_i and input p, and let TP and FP extract the set of true and false positives, respectively. The metamorphic relations are $C_i \sqsubseteq_S C_j \rightarrow TP(f(C_j, p)) \subseteq TP(f(C_i, p))$, and $C_i \sqsubseteq_P C_j \rightarrow FP(f(C_i, p)) \subseteq FP(f(C_j, p))$. Thus, a precision bug exists if $C_i \sqsubseteq_P C_j \wedge (FP(f(C_i, p)) - FP(f(C_j, p)) \neq \emptyset)$, and a soundness bug exists if $C_i \sqsubseteq_S C_j \wedge (TP(f(C_j, p)) - TP(f(C_i, p)) \neq \emptyset)$ for any p. Furthermore, Mordahl and Wei introduced the concept of *implicit soundness partial orders* to capture the fact that changing precision is generally not expected to alter the set of true positives. As such, when there is a precision partial order $C_1 \sqsubseteq_P C_2$, they add the implicit soundness partial orders $C_1 \sqsubseteq_S C_2$ and $C_2 \sqsubseteq_S C_1$.

Despite the definition of partial orders of analysis options, Mordahl and Wei have not addressed the challenges of testing and debugging static analysis, discussed in Section II-A. First, their definition of partial orders is limited, as it requires known ground truths in the input programs or the analysis results be classified to detect violations. This requirement is often infeasible for large programs and/or for certain analyses (e.g., call graph analysis) (*Challenge 2*). Second, their goal of using partial orders is to study the robustness of two Android analysis tools; it is not clear how the violations may assist the debugging process of static analysis tools (*Challenge 3*). Moreover, they have not presented an automated and scalable approach to test the tool configurations in the study; their approach was largely manual, requiring test environments tailor-made to the tools they were evaluating (*Challenge 1*).

III. Configuration Aware Testing and Debugging

We propose a holistic approach to improve the automation of testing and debugging for static analysis, leveraging partial orders of analysis option in all of its components. An overview of our approach is depicted in Figure 2. A key design decision to make such an approach generally useful is to make reasonable assumptions about the inputs that are feasible for tool developers to obtain. Our approach expects three inputs.

First, a configuration grammar of the target tool which specifies all of its options and their settings. Second, a specification of all the precision and soundness partial orders in a tool's configuration space. This specification acts as the *oracle* of the relative expected behavior between tool configurations. We argue that formally specifying a tool's configuration space and partial orders of its options not only increases the chance for these configurations to be tested automatically; it also

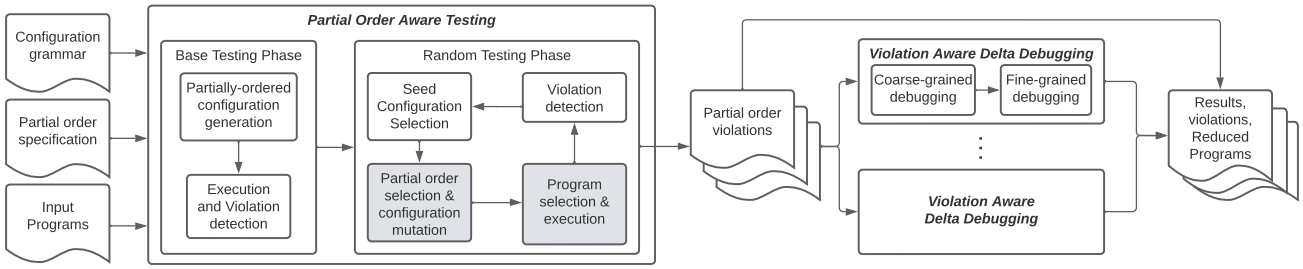

Fig. 2: Overview of configuration aware testing and debugging for static analysis. Gray boxes indicate steps for which we offer two variations in implementation and evaluation (see Section III-B).

allows the users and even the developers to better understand the capabilities of static analysis tools with less ambiguity. The third input is a set of input programs. We improved the partial order definition to make our approach flexible enough to detect bugs with or without ground truths or labeled results (Section III-A). This improvement makes most existing static analysis benchmarks suitable as input to our approach.

As shown in Figure 2, the partial order aware testing phase has two stages. The *base configuration testing* stage starts with the default configuration of the tool. It generates a set of *partially-ordered configurations* by mutating one setting from the default configuration for each setting in the partial order specification. It then runs the static analysis tool using each partially-ordered configuration on each input program. The results of these runs can be checked according to the defined partial orders to identify violations. Note that violations in some partial orders may only be observed when other options are set (i.e., there are interactions in the configuration options). To address this issue, we propose a second stage that randomly and iteratively tests partial orders on more configurations (Section III-B). The output of the testing phase is a collection of partial order bugs, each containing violations (i.e., a specific pair of configurations, the input program, and the partial order that was violated) grouped by partial order.

For each detected partial order bug, we start the debugging process. Violations are caused not only by the static analysis tool, but also by features in the input program. The goal of *violation aware delta debugging* is to reduce input programs to their violation-inducing features while still preserving the violations, in order to help the tool developers better localize the causes of the violations (Section III-C). Overall, the outputs of our approach are analysis results, partial order bugs/violations, and reduced programs.

A. Improving Partial Order Definition

We make two improvements to Mordahl and Wei's partial order definition, with the goal of detecting partial order violations in cases where ground truths in the input programs are not known. As discussed earlier, this is critical for the general applicability of our approach.

First, in addition to the implicit soundness partial orders induced by an explicit precision partial order, we add an implicit precision partial order induced by an explicit soundness partial order. Consider the soundness partial order $C_1 \sqsubseteq_S C_2$; in addition to the expectation that C_2 should never produce more true positives than C_1, we also expect that the former configuration should never produce more false positives than the latter. In other words, increasing soundness should never miss results that were previously present, whether true or false positives. Thus, for the above soundness partial order, we additionally add the implicit precision partial order $C_2 \sqsubseteq_P C_1$.

TABLE I: Violation detection with (rows 2-5) and without (rows 6-7) ground truths. $TP(C_1)$ is shorthand for $TP(f(C_1, p))$ for some p. The first column shows the set relations between two analysis results on program p, and the first row shows the defined partial orders.

	$C_1 \sqsubseteq_S C_2$ $\wedge C_2 \sqsubseteq_P C_1$	$C_1 \sqsubseteq_P C_2$ $\wedge C_1 \sqsubseteq_S C_2$ $\wedge C_2 \sqsubseteq_S C_1$
$TP(C_1) - TP(C_2) \neq \emptyset$	$\emptyset$	$\{C_2 \sqsubseteq_S C_1\}$
$TP(C_2) - TP(C_1) \neq \emptyset$	$\{C_1 \sqsubseteq_S C_2\}$	$\{C_2 \sqsubseteq_S C_1\}$
$FP(C_1) - FP(C_2) \neq \emptyset$	$\emptyset$	$\{C_1 \sqsubseteq_P C_2\}$
$FP(C_2) - FP(C_1) \neq \emptyset$	$\{C_2 \sqsubseteq_P C_1\}$	$\emptyset$
$C_1 - C_2 \neq \emptyset$	$\emptyset$	$\{C_2 \sqsubseteq_S C_1$ $\vee C_1 \sqsubseteq_P C_2\}$
$C_2 - C_1 \neq \emptyset$	$\{C_1 \sqsubseteq_S C_2$ $\vee C_2 \sqsubseteq_P C_1\}$	$\emptyset$

Second, consider the results of two configurations C_1 and C_2 on an input program p. If $f(C_1, p) \subseteq f(C_2, p)$, then without ground truths, we can ascertain $(TP(f(C_1, p)) \subseteq TP(f(C_2, p))) \vee (FP(f(C_1, p)) \subseteq FP(f(C_2, p)))$. Note that these are the conditions for satisfying the partial orders $C_1 \sqsubseteq_P C_2$ and $C_2 \sqsubseteq_S C_1$. So, although we cannot pinpoint the violation to one partial order, if both partial orders are defined, we can say that $f(C_1, p) \not\subseteq f(C_2, p)$ is a violation of either $C_1 \sqsubseteq_P C_2$ or $C_2 \sqsubseteq_S C_1$. We refer to all violations of a single partial order as one *partial order bug*. While less precise than Mordahl and Wei's original approach, this approach addresses *Challenge 2* by allowing us to detect erroneous behavior without ground truths. If we have ground truths defined for an input program, we can fall back to Mordahl and Wei's original relations. Table I shows the possible violations produced by comparing the results of two partially-ordered configurations, both with and without ground truths.

```
1  protected void onCreate(Bundle savedInstanceState) {
2    - super.onCreate(savedInstanceState);
3    - setContentView(R.layout.activity_lifecycle1);
4    TelephonyManager telephonyManager =
5      (TelephonyManager) getSystemService(Context.
       TELEPHONY_SERVICE);
6    String imei = telephonyManager.getDeviceId(); // source
7    URL = URL.concat(imei);
8  }
9  private void connect() throws IOException {
10   URL url = new URL(URL); // sink
```

Fig. 3: Excerpt of *ActivityLifecycle1* from DroidBench.

B. Partial Order Aware Testing

We propose a two-staged testing approach that iteratively constructs partially-ordered configurations, runs them on input programs, and detects violations.

First, the base configuration testing stage starts with a default configuration. This is because the default configuration and those that are slight variations of it are most likely to be what users initially try when using a static analysis tool. For each defined partial order, we mutate the default configuration to produce partially-ordered configurations. We then run the static analysis on each input program with each generated configuration, which produces a set of results. To detect violations, we compare each pair of results on the same input program from two partially-ordered configurations. We query Table I to decide if a violation exists.

After the base configuration testing stage, we offer two variations of random testing for different usage scenarios. This stage uses different configurations than the default, so that we can find partial order bugs that may be caused by the interaction of multiple non-default options. As shown in Figure 2, the random testing phase consists of four steps. First, we select a *seed* configuration from which we mutate. Second, we select some number of partial orders that have not yet exhibited violations. Third, we select some number of input programs to test on. Finally, we follow a similar workflow as the base testing phase; mutating the seed using the selected partial orders, running them on the selected input programs, and looking for violations. The two variations (which affect the gray boxes shown in Figure 2) for selecting the partial order set P and input program set I are as follows:

- *Exhaustive Testing*: P is the set of all partial orders for which we have not yet found violations, and I is the full set of input programs. This variant performs thorough testing of each selected seed configuration.
- *Non-exhaustive Testing*: P is a small random subset of partial orders and I is a small random subset of input programs. This variant favors faster iterations of the testing phase, by running only a few partial orders on a handful of input programs.

Running example: The bug discussed in Section II-A was detected by the base configuration testing phase. FlowDroid's partial order specification includes a partial order $REMOVECODE \sqsubseteq_S PROPAGATECONSTS$ for the option *codeelimination*. In the base configuration testing phase, we mutated FlowDroid's default configuration to create 35 partially-ordered configurations, including a configuration that sets *codeelimination* to *REMOVECODE*. Each configuration was run on all DroidBench programs, and the results were compared to look for partial order violations. On one program, *ActivityLifecycle1* (shown in Figure 3), the configuration that sets *REMOVECODE* misses a true positive flow, from the source on line 6 (*getDeviceId*) to the sink on line 10 (the constructor of *URL*). Not shown is the app's *onStart* callback, which calls *connect*. The default configuration does report this flow. Given the partial order $REMOVECODE \sqsubseteq_S PROPAGATECONSTS$, Table I (row 3, column 2) shows that the absence of the true positive indicates a violation. We output this partial order, the two partially ordered configurations, and this missed true flow as a violation.

C. Violation Aware Delta Debugging

The input program on which a violation is detected is a useful artifact for the debugging process. However, the part of the input program that induces a violation may be a small portion of the program. Therefore we aim to adapt delta debugging [18] to reduce input programs to violation-inducing features, which presents two primary challenges. The first challenge is adapting the delta debugging process to our approach. The second is overcoming issues in efficiency.

To address the first challenge, we adapt delta-debugging to be violation-aware. Specifically, when debugging a violation between two configurations C_1 and C_2 on program P, the delta debugger iteratively proposes a reduced program P', tries to compile P', and if it can, it runs the static analyzer under configurations C_1 and C_2 on P'. Depending on the partial orders between C_1 and C_2 (the first row in Table I) and the type of violation (the cells in Table I), we say P' preserves the violation only if the same set relation between the results of C_1 and C_2 holds as in the first column in Table I.

Most delta debugging techniques could be adapted using the above idea. However, the second challenge (efficiency) arises because each iteration of delta debugging requires two potentially expensive steps: first, we have to try to recompile the altered source code; second, we have to run the static analysis tool under two configurations if the code compiles. Both of these steps can take a long time, making this process inefficient, especially for large programs with many classes where we expect the majority of changes to result in syntactically incorrect programs.

We address this challenge by performing delta debugging at different granularities. We start with a coarse-grained approach, which we adapted from Kalhauge and Palsberg [8]. Their approach works at a class-level granularity, applying a reduction algorithm to transitive closures of the nodes in the class dependency graph (CDG) in order to avoid failed compilations due to removal of classes on which other classes depend. After performing this initial reduction, we switch to a finer grained approach based on Hierarchical Delta Debugging (HDD) [9], which can reduce at statement-level granularity. We call this

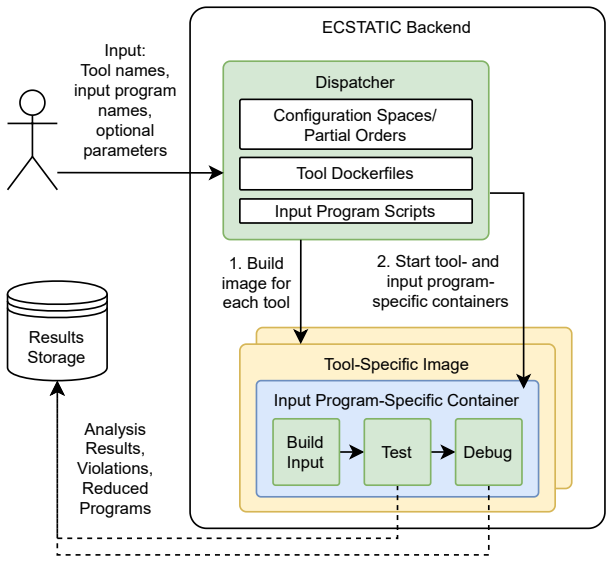

Fig. 4: The architecture of *ECSTATIC*'s backend.

approach CDG+HDD, and describe the implementation in Section IV-A.

Our delta debugging approach is, to our knowledge, the first to implement either of these ideas; namely, applying delta debugging to metamorphic testing of static analyzers, and running delta debugging in two passes in order to improve efficiency. Note that, as discussed in other delta debugging works [18], [19], if there are multiple underlying root causes in the analysis tool that could result in the same violation, our approach cannot guarantee that the root cause of the violation in the reduced program is the same as the root cause of the violation in the original. In this case, the user can iteratively apply the framework, fixing root causes until the violation is no longer detected.

Running example: Figure 3 shows an example of the delta debugger's operation on *ActivityLifecycle1*. The delta debugger was able to remove the two yellow lines in the *onCreate* method which were unnecessary for the flow to be detected, leaving only the code that is necessary to taint *URL*. In addition, the delta debugger removed two omitted lines after the sink in the method *connect*.

D. Limitations

Our approach cannot find all bugs in a static analyzer. As we detect bugs by comparing the results of two configurations, a bug that is common to all configurations cannot be found. Bugs that are not reflected in the analysis results also cannot be detected. For example, if there exists a bug in FlowDroid's implementation of the IFDS algorithm [20], it would likely affect all configurations and would not be able to be detected by our approach.

IV. *ECSTATIC* Design and Implementation

To demonstrate the generality and effectiveness of the approach presented in Section III and to allow this novel partial order aware approach to be easily adopted to test and debug many static analysis tools, we design and implement *ECSTATIC*, an open-source, extensible, easy-to-use, and reproducible framework. *ECSTATIC*'s design has two major parts: (1) a backend that executes the testing and debugging phases on the specified tools and benchmarks, and (2) a set of interfaces to allow easy integration of tools and benchmarks. This was inspired by the design of FuzzBench [21], an open-source platform for evaluating fuzzers, developed by Google.

A. ECSTATIC Backend Design

The design goals of *ECSTATIC*'s backend are that it should be (1) easy-to-use, (2) reproducible, and (3) scalable. Figure 4 shows its high-level architecture. *ECSTATIC* takes names of tools and input programs, along with optional parameters that control timeouts, random testing variant, and logging. The *dispatcher* manages all testing and debugging executions with three pieces of information of the integrated tools and input programs: the tools' partial orders and configuration spaces (specified as JSON files), Dockerfiles [10] that build images for each tool, and the scripts needed to build the input programs. For each tool name the user supplies, the dispatcher first builds a tool-specific Docker image, which downloads and builds the tool. This image inherits from *ECSTATIC*'s base image, which sets up *ECSTATIC* and its dependencies. Next, the dispatcher starts a new Docker container, responsible for the testing and debugging phases, for each specified benchmark in each tool-specific image. The outputs (i.e., analysis results, detected violations, and reduced programs) from different containers are stored on the host machine, allowing the user to access and analyze all results together.

The above architecture makes *ECSTATIC*'s backend satisfy our design goals. By containerizing analyses inside of Docker, *ECSTATIC* results are reproducible and platform independent. *ECSTATIC* is easy-to-use and automated; users only need to specify the names of the integrated tools and input programs. *ECSTATIC* is also scalable, as we allow the testing and debugging phases to be run in parallel, with the level of parallelism tunable by the user.

Implementation: The dispatcher and testing phase were implemented in 2855 lines of code in Python. In the random testing phase, we re-used the GrammarCoverageFuzzer from the Fuzzing Book [22] to generate seed configurations, and implemented the violation detector following Table I.

We implemented the violation aware delta debugger in 2123 lines of code in Java. This delta debugger can be used for debugging any static analysis tools targeting Java or Android programs. The debugger performs both class-level CDG-based debugging and AST-based hierarchical delta debugging, which can reduce statement-level granularity. We use JavaParser [23] to produce and manipulate the ASTs for both Java and Android, and JDeps [24] to construct CDGs.

B. Tool and Input Program Integration Interface

ECSTATIC exposes interfaces to integrate new tools and input programs with the goal of making the integration process

principled and easy. Using these interfaces, it typically requires only dozens of lines of code to integrate a new tool or new input programs (see Section IV-C).

Tool Integration Interface: To integrate a new tool in *ECSTATIC*, four components need to be extended. First, a new Dockerfile that sets up the analysis tool needs to be written.

Second, *ECSTATIC*'s interface called *AbstractRunner* needs to be extended. This interface exposes 14 methods that can be overridden to specify how the target tool is run. In most cases, only 4 methods need to be overridden, in order to specify the command to invoke the analysis tool, the commands to specify the inputs and output, and the command to pass a timeout to the analysis tool.

Third, *ECSTATIC*'s *AbstractResultReader* interface needs to be extended. This interface specifies how to read and compare the results of the target tool. *AbstractResultReader* requires the tool results to be stored as a collection of individual results. Typically, a common result form exists for different tools performing the same client analysis. For example, all call graph analysis results can be stored as a collection of call graph edges. Therefore, if the new tool performs an analysis that has already been integrated in *ECSTATIC*, the previously extended *AbstractResultReader* can be reused.

Fourth, the partial orders of analysis options (used by the violation detector) and configuration grammar (used by the grammar-based fuzzer) need to be specified in two JSON files.

Input Program Integration Interface: In order to integrate a new input program for testing, one simply needs to add a `build.sh` script which downloads and builds the input program. In order to perform delta debugging, one must additionally supply an *input program index* as a JSON file. This file lists, for each program, where its source code is, so that *ECSTATIC* can pass this information to the delta debugger.

ECSTATIC also supports supplying ground truths with input programs. *ECSTATIC* expects these ground truths to be specified in a format that can be read by the *AbstractResultReader*. Providing ground truths will automatically switch *ECSTATIC* from the ground-truth unaware violation detection method specified in Section III to the ground-truth aware one.

C. Integrated Tools and Benchmarks

We have integrated four configurable static analysis tools in *ECSTATIC*: FlowDroid [1], SOOT [2], WALA [4], and DOOP [3]. SOOT, WALA, and DOOP are three widely used frameworks for Java static analysis. For these three frameworks, we called provided interfaces to build call graphs. FlowDroid is the most popular static taint analyzer for Android.

Table II shows the number of analysis options and partial orders in each tool (rows 2 and 3), as well as the number of lines of code in each tool's Dockerfile, reader, and runner (rows 4-6). Overall, it takes only a few lines of code to integrate a tool into *ECSTATIC*: at most, 156 for FlowDroid.

The process of producing the configuration space specification was roughly the same for each tool. Given an analysis tool, we thoroughly explored the tool's documentation and used our domain expertise to identify options with clear soundness/precision partial orders. We also performed a cursory investigation of the code, in order to identify undocumented options and additional documentation that could help infer partial orders. This process typically took a handful of hours to complete. This approach depends on the quality of the documentation, which is often ambiguous, incomplete, or out of date. Unfortunately, the alternative, a full code review to understand the actual effect of each configuration option, is infeasible. We hope our work motivates tool developers to be more explicit about the configuration spaces in their tools, which could directly be used as input to *ECSTATIC*.

TABLE II: Configuration spaces of SOOT, WALA, DOOP, and FlowDroid, and lines of code needed to integrate them into *ECSTATIC*.

	SOOT	WALA	DOOP	FlowDroid
# Options	20	5	20	22
# Partial Orders	20	26	35	77
Dockerfile LoC	15	14	38	21
Reader LoC	12	19	26	16
Runner LoC	43	20	47	119
Total LoC	90	53	111	156

SOOT's configuration space is partitioned into "phases," which cover specific stages of the analysis. Such options are specified as `-p <phase> <option>:<setting>`. For example, configuration options for call graph construction are accessed through the `cg` phase, which exposes various subphases, such as SPARK (`cg.spark`). Each (sub)phase may implement its own options. For example, setting `-p cg.spark types-for-sites:true` causes types to be used as elements of points-to sets, rather than allocation sites. We only included configuration options that were in the `cg` phase of SOOT, since our experiments focus on detecting bugs in SOOT's call graph construction. The options in this phase had clear soundness/precision effects on the tool's output.

WALA neither provides a command line interface for call graph construction, nor does it have documentation about the configuration options that are available. Thus, we used the WALA library to implement our own call graph driver. To discover configuration options, we did a code review to identify options in WALA's call graph construction phase, and exposed those options in our driver. For example, the *cgalgo* option exposes various call graph algorithms, such as RTA [25], VTA [26], k-call-site-sensitivity [27], and k-object-sensitivity [28], [29], while *handleStaticInit* controls whether calls to static initializers are modeled.

For DOOP, we followed this methodology and were exhaustive with regard to the documented configuration space. The vast majority of DOOP's partial orders are within its *analysis* option, which exposes 32 settings, covering various heap abstractions, analysis types, and sensitivities (e.g., *context-insensitive*, *1-call-site-sensitive+heap*, *3-object-sensitive+3-heap*). Unlike other tools, DOOP does not allow the user to supply parameterized context-sensitive analyses via its command line interface.

For FlowDroid, we began from the configuration space defined by Mordahl and Wei, which was produced using a

similar methodology [5]. We modified inaccurate partial orders from this space. Specifically, we changed the partial order *aliasflowins.TRUE* $\sqsubseteq_P$ *aliasflowins.FALSE* to *aliasflowins.FALSE* $\sqsubseteq_S$ *aliasflowins.TRUE*, which we deemed congruent with the behavior of FlowDroid after performing a code review.

Some tools' configuration spaces contain conflicts between option settings. Mordahl and Wei identified one type of conflict, which they referred to as *disablement*. Given two options, o_1 and o_2, a setting $s_1 \in o_1$ *disables* a setting $s_2 \in o_2$ if $\forall p, \forall C, f(C[o_1.s_1, o_2.s_2], p) = f(C[o_1.s_1], p)$.[1] They identified 5 disablement relationships in FlowDroid. We do not explicitly handle these relationships, as they do not impact the correctness of our approach; rather, they simply result in wasted runs, as we compare the results of two configurations which invariably behave the same. We did encounter one non-disablement conflict in SOOT, wherein enabling SPARK [30] via the *cg.spark:on-fly-cg* option throws an exception when certain other options in the *cg.spark* phase are set (e.g., *cg.spark:VTA*). We thus explicitly set *cg.spark:enabled* to *false* in our *SOOTRunner*.

We have integrated four benchmarks in *ECSTATIC*: CATS [11], [31], DaCapo [7], DroidBench [6], [1], [32], and FossDroid [12], [5]. The CATS and DaCapo benchmarks consist of Java programs, while DroidBench and FossDroid consist of Android applications.

DroidBench is a popular benchmark consisting of 190 hand-crafted Android APKs for which the ground truths are known. The input programs are organized into 22 categories, each aiming to test a taint analysis' ability to handle certain elements of Java and/or Android (e.g., Aliasing, Android Lifecycle, and Reflection) [1]. This benchmark has been used in several works to evaluate Android taint analysis tools [32], [5], [1], [33], [34]. DroidBench programs range from 8 to 236 lines of code. The benchmark script for DroidBench consists of 8 lines of code.

The FossDroid benchmark consists of one real-world Android application, Alarm Klock, originally collected by Mordahl and Wei [5] from FossDroid [12], a repository for open-source Android applications.[2] True (16) and false (160) positives of FlowDroid results running this program were manually classified. This program consists of 3313 lines of code. The benchmark script for FossDroid contains 12 lines of code.

CATS consists of 112 small Java programs, ranging from 2 to 36 lines of code in size. These microbenchmarks were developed by Reif *et al.* in order to evaluate Java static analysis tools [31]. This benchmark tests 15 Java features, including Reflection, Virtual Calls, and Dynamic Proxies. The benchmark script for CATS contains 8 lines of code.

DaCapo is a benchmark of open-source, real-world Java programs [7] that has been widely used to evaluate static analysis tools [29], [35], [36]. We integrated version 2006-10-MR2, with modifications made to compile the programs with a Java 8 compiler. The 11 input programs contain an average of 85K lines of code, and the script to integrate DaCapo contains 9 lines of code.

[1] $C[o.s]$ should be understood as a configuration equivalent to C except with option o set to s.

[2] Alarm Klock was chosen to be integrated because it was the most-labeled program in the FossDroid dataset Mordahl and Wei created, suitable for using as an input program with ground truths.

V. Evaluation

In this section, we set up the experiments and present evaluation results of our partial order aware testing and debugging on the integrated tools in *ECSTATIC*.

A. Experimental Setup

Research questions: Our evaluation aims to answer three research questions.

RQ1: *How effective is ECSTATIC's partial order aware testing?* To answer RQ1, we measured the number of partial order bugs detected on each tool. We discuss how the characteristics of the input programs affect *ECSTATIC*'s ability to detect bugs, and analyze the effectiveness of each stage of the partial order aware testing.

RQ2: *How effective is ECSTATIC's violation aware delta debugging?* To answer RQ2, we compared the sizes of each input program before and after running the delta debugger to measure the reduction rate, and we compared CDG+HDD with a baseline that performs only hierarchical delta debugging on the ASTs (HDD-only).

RQ3: *Are ECSTATIC's outputs useful for tool developers?* To answer RQ3, we discuss the bugs we reported to tool developers and their responses, including fixes that have been made.

Inputs to ECSTATIC: In our experiments, we ran all three Java static analysis tools (SOOT, DOOP, and WALA) on both Java benchmarks (DaCapo and CATS), and FlowDroid on both of the Android benchmarks, DroidBench and FossDroid. The timeouts were determined through preliminary experiments on the performance of each tool. In the base configuration testing stage and the delta debugging phase, for SOOT, WALA, and FlowDroid, we used a timeout of 15 minutes for each microbenchmark program (CATS and DroidBench) and 30 minutes for each real-world program (DaCapo and FossDroid). DOOP executions took significantly more time and memory; we used a 30-minute timeout for each CATS program and a 45-minute timeout for each DaCapo program in the base configuration testing stage. We ran both variants of the random testing phase, each for 24 hours with 4 CPU cores and a smaller timeout for each program. For non-exhaustive testing, we randomly selected at most 4 programs and 2 partial orders in each iteration. All programs and partial orders were sampled at least once during non-exhaustive testing. Each random testing phase was run twice; because we observed little variance between the results of the two trials, we report the results of the first trial. We ran the delta debugger with a 6-hour timeout for each violation. We did not run the random testing and delta debugging phases with DOOP due to its large memory footprint.

TABLE III: Partial order bugs detected in each tool by dataset. The bar in each cell differentiates bugs detected in the base testing phase (left) and bugs detected only in the random testing phase (right).

	SOOT	WALA	DOOP	FlowDroid	Total
Microbenchmark	3 | 0	0 | 0	0 | 0	26 | 2	29 | 2
Real-world	18 | 0	6 | 3	12 | 0	2 | 7	38 | 10
Total	18 | 0	6 | 3	12 | 0	28 | 7	64 | 10

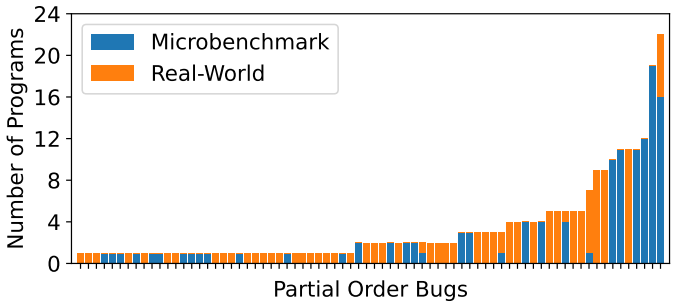

Fig. 5: Number of programs that each partial order bug appeared in.

Experimental environment: Experiments were conducted within Docker containers based on a Ubuntu 20.04 image. These containers were deployed across three machines. All experiments for the testing phases of SOOT, WALA, and DOOP, as well as random testing for FlowDroid were conducted on a server with 376GB of RAM and 2 Intel Xeon Gold 5218 16-core CPUs @ 2.30GHz running Ubuntu 18.04. Base configuration testing for FlowDroid was run on a workstation with 32GB of RAM and an Intel Core i7-9800X CPU @ 3.8GHz running Ubuntu 20.04. Delta debugging was conducted on a server with 141GB of RAM and 2 Intel Xeon Silver 4116 12-core CPUs @ 2.10GHz running Ubuntu 16.04.

B. RQ1: Performance of Partial Order Aware Testing

Table III shows the number of partial order bugs detected by *ECSTATIC* for each tool and benchmark. For Java tools, microbenchmark refers to CATS; for FlowDroid, it refers to DroidBench. Real-world refers to DaCapo for Java, and FossDroid for FlowDroid.

Overall, *ECSTATIC* detected 74 partial order bugs in the four tools. Even without ground truths in the input programs, we were able to detect violations in SOOT, WALA, and DOOP. The random testing phase was able to detect 10 additional bugs in WALA and FlowDroid that were not detected in the base testing phase; these are bugs that only appear under certain option interactions not in the default configuration.

Over 24 hours, both exhaustive and non-exhaustive random testing produced similar results. Exhaustive testing found 9 new bugs, and non-exhaustive testing found 8 new bugs, with an intersection of 7. The exhaustive variant alone was able to find violations of the partial orders *cgalgo.RTA* $\sqsubseteq_P$ *cgalgo.CHA* and *staticmode.NONE* $\sqsubseteq_P$ *staticmode.DEFAULT* in FlowDroid. The non-exhaustive variant alone was able to detect a violation in WALA of the partial order *cgalgo.1-CFA* $\sqsubseteq_P$ *cgalgo.0-1-CFA*. That the two approaches found different bugs in 24 hours

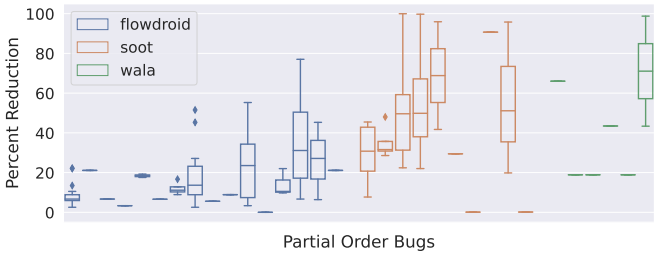

Fig. 6: Reduction rate on partial order bugs.

Fig. 7: A comparison showing the performance of CDG+HDD compared to HDD-only on real-world benchmarks.

indicates the best strategy may be to run both approaches concurrently. We confirmed this by running the random testing experiments for 48 hours, and found cases in which a 24-hour exhaustive testing run found violations that a 48-hour non-exhaustive run missed, and vice-versa.

Figure 5 shows the number of programs each partial order bug was detected in. 35 out of the 74 partial order bugs were detected in only one program. Only 5 partial order bugs appeared in programs in more than one dataset. This illustrates the necessity of having large, diverse sets of programs on which to test static analysis tools.

C. RQ2: Performance of Violation Aware Delta Debugging

Overall, the two-staged delta debugger (CDG+HDD) was able to reduce programs by an average of 26% (7167 LoC), as opposed to 14% when run with only hierarchical delta debugging (HDD-only). This difference was even more dramatic for real-world programs; CDG+HDD reduced real-world programs by an average of 50% (29187 LoC), compared to 6% by HDD-only.

Figure 6 shows the reduction rate per partial order bug, colored by tool. We can see that the reduction rates for programs that triggered violations in SOOT and WALA are higher on average; this is due to all but one of FlowDroid's inputs being a microbenchmark. CDG+HDD did not have much impact over HDD-only for microbenchmarks. This is expected, as microbenchmarks usually have only a single class. Figure 7 compares reduction on real-world programs between HDD-only and CDG+HDD. At most, the two-stage delta debugger was able to reduce a program, `hsqldb`, by 99%, from 65487 lines to 29 (on this same case, HDD-only removed only 285 lines). This illustrates the utility of CDG+HDD for reducing input program sizes to assist debugging, especially for large real-world programs.

```
1 // AnonymousClass1.apk
2 LocationListener loLi = new LocationListener() {...};
3 locationManager.requestLocationUpdates(..., loLi);
4
5 // FlowDroid
6 for (Type possibleType : possibleTypes) {
7   if (possibleType instanceof AnySubType)
8     targetType = ((AnySubType) possibleType).getBase().
       getSootClass(); }
```

Fig. 8: Code from DroidBench's *AnonymousClass1.apk*, and code from FlowDroid which failed to soundly model a callback registration.

In terms of time, CDG+HDD delta debugging hit the 6 hour timeout for all but two real-world violations. These were a violation on SOOT from `hsqldb`, which took 4.6 hours, and a violation on FlowDroid from Alarm Klock, which took 42 minutes. The delta debugger never timed out on microbenchmarks, taking a mean of 15 minutes and a maximum of 57 minutes.

D. RQ3: Usefulness of ECSTATIC for Developers to Debug

We reported a subset of the violations detected in the base testing phase to tool developers, so as not to flood developers with many bug reports at once. Specifically, we reported all the partial order bugs detected from the base testing phase from the microbenchmarks for SOOT and FlowDroid (3 and 26, respectively), all 12 bugs for DOOP, and 1 bug for WALA, totaling 42. For each bug, we provided developers our expectation of the tool behavior, the unexpected differences (i.e., the violation), and the associated input programs.

As of time of writing, we have heard back from developers of FlowDroid, WALA, and DOOP. For FlowDroid, we have received confirmation that four of the issues we raised were real (with three having been fixed) and discussion on a fifth is ongoing. For WALA, it was confirmed that we found unexpected behavior, and we are still in communication with the developer to try to find all of the root causes. Finally, for DOOP, the developer acknowledged that the behavior was unexpected, and confirmed it may be caused by a bug, but that it could also be caused by internal heuristics DOOP uses to control its performance. So far, no developer has communicated with us that behavior we reported is intended or is otherwise not indicative of a bug. Figure 1 shows part of the bugfix of a FlowDroid bug; fixing this bug in the *REMOVECODE* setting involved modifying 48 lines of code, contributed by the first author of this paper and the FlowDroid developer.

Another bug we found in FlowDroid was in the implementation of CHA [37]. Figure 8 shows code from DroidBench's *AnonymousClass1.apk*, which creates an instance of an anonymous subtype of *LocationListener* (line 2), and then registers it as a callback (line 3). Lines 6-8 show code from the *analyzeMethodForCallbackRegistrations* method of FlowDroid, in which a method call that is known to register callbacks is processed to find callback registrations. Normally, the value of *possibleTypes* on line 6 is determined through a points-to analysis; however, when CHA is activated, the points-to analysis is replaced with a dummy implementation which uses the *AnySubType* type to indicate that *loLi* may be any subtype of *LocationListener*. As shown on line 8, instead of iterating through these potential subtypes, FlowDroid unsoundly treats the variable as if it could only be of type *LocationListener*. This bug has been fixed thanks to a bug report we submitted, and now *AnySubTypes* are correctly iterated through [38].

For the bug we reported in WALA, we were able to uncover unsoundness in WALA's modeling of reflection via a violation of the partial order $reflection.STRING\_ONLY \sqsubseteq_S reflection.NONE$. One violation of this bug was detected in a DaCapo program, `hsqldb`. Our delta debugger on this violation reduced `hsqldb` from 65487 to 818 lines of code, which we provided to the developer to reproduce the bug. Specifically, the problem is that $STRING\_ONLY$ enables logic to insert a synthetic target representing the runtime target of a reflective call if the parameter to the reflective call is a string constant (e.g., `class.forName("java.lang.String")`) that WALA can resolve. However, when a configuration with $STRING\_ONLY$ resolved a string, WALA did not model any exceptions arising from the synthetic target. *ECSTATIC* detected the missing edges out of catch blocks in runs with *STRING_ONLY*.

VI. Threats to Validity

There are several potential threats to the validity. First, while the metrics we used to measure the effectiveness of *ECSTATIC* have been adopted in previous work (e.g., input size reduced by delta debugger [9]), these metrics may not directly indicate a reduction in the manual debugging efforts of tool developers. To mitigate this, we reported bugs to tool developers using *ECSTATIC* outputs and show that they are useful for fixing real bugs. Second, we only integrated tools targeting Java-based languages. There could be unforeseen hurdles to integrate analysis tools and benchmarks across other programming languages. Third, our results may not fully account for the randomness in the random testing phase. We ran two trials and observed the variance was low. Fourth, the partial order specification we used in the evaluation may not be correct. In all the bugs we reported, no developer has reported that our partial orders were incorrect. Furthermore, we hope this work can inspire tool developers to be more explicit in defining the expected behavior of different configuration options.

VII. Related Work

Evaluation of Static Analysis Configurations: Mordahl and Wei were the first to use partial orders to study bugs in two Android taint analysis tools [5]. They defined partial orders and found violations in these tools, indicating the necessity of configuration aware testing. However, their work did not address the challenges in testing and debugging static analysis, as discussed in Section II-B. Our work is inspired by their idea and proposes an automated process, adding partial order aware testing that can exercise option interactions that do not occur in the default configuration. Furthermore, we present violation aware delta debugging to aid developers in fixing bugs.

Smaragdakis *et al.* formally modeled the design space of object-sensitive analyses and evaluated the influences of different object-sensitive analyses on precision and performance [29]. Lhoták and Hendren empirically evaluated the precision of context-sensitive analyses within SOOT [35]. Wei *et al.* developed a Java numeric analysis based on WALA and evaluated 216 configurations of their tool [36]. All three of these works used the DaCapo benchmark for evaluation. Reif *et al.* presented CATS to systematically evaluate the unsoundness of call graph construction algorithms [31]. Other works focus on evaluating FlowDroid and other Android taint analysis tools against each other [1], [32], [34], [33], using the DroidBench benchmark. We tested and debugged many tool configurations evaluated in the above works and detected partial order bugs. This result demonstrates that while past evaluations are useful for comparing performances between tool configurations, they lack the ability to test the correctness of tool implementations. Furthermore, our goal in this work is not to compare tools, but to provide a flexible framework to test and debug configurable static analysis tools.

Testing and Debugging Static Analysis and Compilers: Do *et al.* introduced VISUFLOW, a visual debugging environment for FlowDroid [14]. Our approach complements their work in that it provides specific diagnostic information for a bug. Andreasen and Møller diagnosed JavaScript analyses that suffered from imprecision and high memory usage when analyzing jQuery [39], using a combination of JS Delta [40] and the TAJS inspector [15], [41]. We similarly used delta debugging to help reduce inputs, but our approach focuses on addressing the lack of oracles when testing static analysis and can be used to detect and help debug both precision and soundness issues. Wei *et al.* presented an approach to diagnose sources of imprecision in JavaScript analyses by monitoring an analysis' execution [42]. Our approach does not monitor the analysis process, but rather, generates test cases to find bugs by comparing the results of multiple configurations. As a result, our approach is less intrusive and more general.

Our approach is also related to compiler testing, as static analyses are often implemented within compilers. Metamorphic testing is a common approach for testing compilers [43]. These approaches generally construct two programs that are equivalent, and then compile them with the same compiler to ensure the two executables behave the same. For example, Le *et al.* propose an approach for generating equivalent programs by inserting and deleting code in dead regions [44], [45]. Sun *et al.* propose a more general approach to generating equivalent variants of a program within their tool, *Hermes*, which allows mutation of both dead and live regions of code [46]. While all of these works consider some equivalence relation, our work uses a subset-based metamorphic relation in order to find bugs. Furthermore, our work is the first to apply such a relation to static analysis in order to find bugs.

Configuration Testing: Combinatorial interaction testing (CIT) is a common technique for testing configurable software [47], [48], [49], [5], [50]. The goal of CIT is to generate various configurations and execute those configurations in order to maximize coverage of the software's features. While these approaches test configurations to find bugs, they require an oracle to determine whether a test passed or failed. Our work treats the partial order specification as the test oracle.

Several configuration fuzzing techniques have been developed. such as ConfigFuzz, which expands the program input with configurations and fuzzes configurations using coverage feedback [51]. Lee *et al.* presented a 2-stage fuzzer, POWER, to explore configurations [52]. Fuzzing configurations is a research direction we are interested in pursuing in the future.

VIII. CONCLUSIONS

In this work, we presented *ECSTATIC*, an easy-to-use, extensible, and scalable open-source framework for automated testing and debugging of configurable static analysis tools. *ECSTATIC* exposes a simple frontend, allowing addition of new tools and benchmarks using feasible-to-obtain inputs: configuration grammar, partial order specification, and scripts that typically are only dozens of lines of code. Given these inputs, *ECSTATIC* leverages partial order relationships between configuration options to iteratively test for bugs in configurable static analysis, even without a ground-truth benchmark. *ECSTATIC* then performs violation-aware delta debugging, in order to produce reduced programs that exhibit bugs on analysis tools, which are useful artifacts for debugging. We integrated four popular static analysis tools–SOOT, DOOP, WALA, and FlowDroid–as well as four benchmarks into *ECSTATIC*.

Using *ECSTATIC*, we found 74 partial order bugs across all four tools, out of 158 defined partial orders, using both real-world and microbenchmarks. We reported a subset of these bugs to tool developers, leading to three bug fixes in FlowDroid and ongoing discussions about potential bugs in WALA, FlowDroid, and DOOP. *ECSTATIC*'s violation-aware delta debugging was able to reduce real-world programs to an average of 50% of their original size, with a maximum observed reduction of 99%. In addition to showing the efficacy of *ECSTATIC*'s automated testing approach, our results demonstrate the necessity of large, diverse benchmarks for testing static analysis tools.

In the future, we plan to integrate more tools into *ECSTATIC*, covering more target languages and types of analysis. We plan to extend the core technique with more fuzzing strategies in order to allow more sophisticated exploration of tools' configuration spaces. We plan to research the potential of comparing intermediate analysis states in addition to final analysis results to allow detecting more bugs (Section III-D). Additionally, we plan to add support for constraints in the configuration space to handle conflicts between configuration options. We also plan to extend support for the task of debugging configurable static analysis past delta debugging (e.g., by integrating fault localization techniques to help users find bugs faster).

ACKNOWLEDGMENT

This work was partly supported by NSF grants CCF-2047682 and CCF-2008905, the NSF graduate research fellowship program, and Eugene McDermott Graduate Fellowship 202006.

References

[1] 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," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[2] 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*, ser. CASCON '10. USA: IBM Corp., 2010, p. 214–224. [Online]. Available: https://doi.org/10.1145/1925805.1925818

[3] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *SIGPLAN Not.*, vol. 44, no. 10, p. 243–262, oct 2009. [Online]. Available: https://doi.org/10.1145/1639949.1640108

[4] "Wala," https://github.com/wala/WALA, 2022.

[5] A. Mordahl and S. Wei, "The impact of tool configuration spaces on the evaluation of configurable taint analysis for android," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 466–477. [Online]. Available: https://doi.org/10.1145/3460319.3464823

[6] "DroidBench 3.0," https://github.com/FoelliX/ReproDroid, 2021.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[8] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 556–566.

[9] G. Misherghi and Z. Su, "Hdd: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 142–151. [Online]. Available: https://doi.org/10.1145/1134285.1134307

[10] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[11] "The call-graph assessment & test suite," https://bitbucket.org/delors/cats/src/master/, 2022.

[12] "FossDroid," https://fossdroid.com, 2022.

[13] "Flowdroid," https://github.com/secure-software-engineering/FlowDroid/issues/496, 2022, issue #496.

[14] L. N. Q. Do, S. Krüger, P. Hill, K. Ali, and E. Bodden, "Debugging static analysis," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 697–709, 2020.

[15] "Tajs," https://github.com/cs-au-dk/TAJS, 2022.

[16] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," *arXiv preprint arXiv:2002.12543*, 2020.

[17] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, jan 2018. [Online]. Available: https://doi.org/10.1145/3143561

[18] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[19] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 184–191.

[20] 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: Association for Computing Machinery, 1995, p. 49–61. [Online]. Available: https://doi.org/10.1145/199448.199462

[21] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: An open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

[22] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Fuzzing with grammars," in *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2022, retrieved 2022-01-12 14:39:50+01:00. [Online]. Available: https://www.fuzzingbook.org/html/Grammars.html

[23] "Javaparser," https://javaparser.org, 2022.

[24] "jdeps," https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jdeps.html, 2022.

[25] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 324–341. [Online]. Available: https://doi.org/10.1145/236337.236371

[26] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 264–280. [Online]. Available: https://doi.org/10.1145/353171.353189

[27] M. Sharir, A. Pnueli *et al.*, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ..., 1978.

[28] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, p. 1–41, jan 2005. [Online]. Available: https://doi.org/10.1145/1044834.1044835

[29] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 17–30. [Online]. Available: https://doi.org/10.1145/1926385.1926390

[30] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *International Conference on Compiler Construction*. Springer, 2003, pp. 153–169.

[31] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, *Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs*. New York, NY, USA: Association for Computing Machinery, 2019, p. 251–261. [Online]. Available: https://doi.org/10.1145/3293882.3330555

[32] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 331–341. [Online]. Available: https://doi.org/10.1145/3236024.3236029

[33] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[34] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 176–186. [Online]. Available: https://doi.org/10.1145/3213846.3213873

[35] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, oct 2008. [Online]. Available: https://doi-org.libproxy.utdallas.edu/10.1145/1391984.1391987

[36] S. Wei, P. Mardziel, A. Ruef, J. S. Foster, and M. Hicks, "Evaluating design tradeoffs in numeric static analysis for java," in *Programming Languages and Systems*. Springer International Publishing, 2018, pp. 653–682. [Online]. Available: https://doi.org/10.1007%2F978-3-319-89884-1_23

[37] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th*

European Conference on Object-Oriented Programming, ser. ECOOP '95. Berlin, Heidelberg: Springer-Verlag, 1995, p. 77–101.

[38] "Flowdroid," https://github.com/secure-software-engineering/FlowDroid/issues/503, 2022, issue #503.

[39] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," in *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2014.

[40] "Js delta," https://github.com/wala/jsdelta, 2022.

[41] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Static Analysis*, J. Palsberg and Z. Su, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 238–255.

[42] S. Wei, O. Tripp, B. G. Ryder, and J. Dolby, "Revamping javascript static analysis via localization and remediation of root causes of imprecision," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 487–498. [Online]. Available: https://doi.org/10.1145/2950290.2950338

[43] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, feb 2020. [Online]. Available: https://doi.org/10.1145/3363562

[44] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *SIGPLAN Not.*, vol. 49, no. 6, p. 216–226, jun 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594334

[45] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 386–399. [Online]. Available: https://doi.org/10.1145/2814270.2814319

[46] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," *SIGPLAN Not.*, vol. 51, no. 10, p. 849–863, oct 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984038

[47] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 75–86. [Online]. Available: https://doi.org/10.1145/1390630.1390641

[48] M. Cohen, P. Gibbons, W. Mugridge, and C. Colbourn, "Constructing test suites for interaction testing," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 38–48.

[49] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, feb 2011. [Online]. Available: https://doi.org/10.1145/1883612.1883618

[50] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across configurations: Implications for combinatorial testing," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 6, p. 1–9, nov 2006. [Online]. Available: https://doi.org/10.1145/1218776.1218785

[51] Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei, "Registered report: Fuzzing configurations of program options." San Diego, CA, USA: International Fuzzing Workshop (FUZZING) 2022, April 2022. [Online]. Available: https://dx.doi.org/10.14722/fuzzing.2022.23008

[52] A. Lee, I. Ariq, Y. Kim, and M. Kim, "Power: Program option-aware fuzzerfor high bug detection ability." 15th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2022, April 2022.

[53] O. Tange, "Gnu parallel 20211222 ('støjberg')," Dec. 2021, GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them. [Online]. Available: https://doi.org/10.5281/zenodo.5797028