ECSTATIC: Automatic Configuration-Aware Testing and Debugging of Static Analysis Tools

Austin Mordahl
austin.mordahl@utdallas.edu
The University of Texas at Dallas
Richardson, TX, USA

Dakota Soles
dakota.soles@utdallas.edu
The University of Texas at Dallas
Richardson, TX, USA

Miao Miao
mmiao@utdallas.edu
The University of Texas at Dallas
Richardson, TX, USA

Zenong Zhang
zenong@utdallas.edu
The University of Texas at Dallas
Richardson, TX, USA

Shiyi Wei
swei@utdallas.edu
The University of Texas at Dallas
Richardson, TX, USA

ABSTRACT

Static analyses are powerful tools that can serve as a complement to dynamic approaches such as testing. In order to ensure generality, many static analysis tools are configurable. However, these configurations can make testing and debugging more difficult. To address this issue, we introduce a new tool, ECSTATIC, which leverages partial order relations between analysis configuration options to automatically test and debug static analyzers, even without ground truths. ECSTATIC's results are reproducible by virtue of running within Docker containers, and ECSTATIC provides clear extension interfaces for users to add their own tools and input programs. We evaluated ECSTATIC on four popular dataflow analysis tools, and found 74 bugs in all four tools. We also found that ECSTATIC's novel two-staged delta debugging was able to reduce real-world programs by 50%, compared to a baseline of 6%.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Automated static analysis**.

KEYWORDS

static analysis, testing, debugging, configurable software

ACM Reference Format:
Austin Mordahl, Dakota Soles, Miao Miao, Zenong Zhang, and Shiyi Wei. 2023. ECSTATIC: Automatic Configuration-Aware Testing and Debugging of Static Analysis Tools. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604918

1 INTRODUCTION

Static analysis is a powerful technique that can be used to find bugs, perform code optimizations, or prove myriad properties about programs. The power of static analysis lies in the fact that it models a program's execution without running it. For the task of finding bugs, specifically, static analysis can serve as a complement to testing approaches, finding bugs in parts of the code that may not be executed by a test suite. However, because static analysis is an undecidable problem, there is no one-size-fits-all analysis. An analysis designer needs to balance the needs for *precision* (i.e., the quality of not over-approximating a program's behavior, typically measured through false positives), *soundness* (i.e., the quality of not under-approximating a program's behavior, typically measured through true positives), and termination. Any individual use case and target program can require a different set of static analysis algorithms in order to be analyzed with an optimal tradeoffs.

These tradeoffs are often implemented with configuration options. While configuration makes analysis tools more general, it can also lead to bugs that only exist under certain configurations. Even without configuration, testing static analysis tools is difficult, because the test cases are real programs about which some property is known. These test cases are not trivial to create, and adding a dimension of configuration means that the expected output of the analysis can be different in different configurations. Thus, exhaustively testing the configurations of an analysis is infeasible.

In this work, we propose ECSTATIC, a new tool that enables analysis designers and users to automatically test and debug configurable static analyses. The key idea that ECSTATIC leverages is that of *partial orders between analysis options* – these are theoretical relations between different analysis algorithms that provide expected relations between the algorithms' outputs. For example, consider object sensitivity [13, 18], a form of context sensitivity [17] for object-oriented programs; object sensitivity uses objects' allocation sites as contexts, which allows an analysis to analyze method calls differently for different sequences of receivers. The algorithm is parameterized with a natural number, k, that indicates the maximum context depth. All other things being equal, an analysis with a higher k should be at least as precise as an analysis with a lower k. In other words, an operator $\sqsubseteq_P$ (read as "is at least as precise as") induces a partial ordering over the values of object sensitivity. We can also define soundness partial orders using the $\sqsubseteq_S$ operator.

ECSTATIC uses these partial orders to perform automatic testing and debugging of configurable static analysis tools. Given a tool's partial order specification, a configuration grammar, and a set of input programs, ECSTATIC automatically constructs test cases to exercise the partial orders, and reports potential bugs when partial orders are violated. ECSTATIC then uses a new two-staged

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3604918

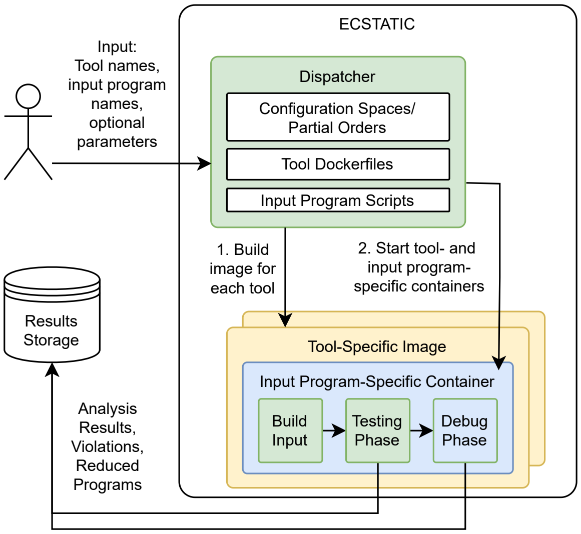

Figure 1: An overview of the ECSTATIC approach. The overall operation of ECSTATIC is shown in the input program-specific container. Note that the debugging phase is optional.

partial-order aware delta debugging technique to reduce the input programs to their failure-inducing features. To achieve the design goals of *extensibility* and *reproducibility*, ECSTATIC provides clear, well-documented extension interfaces to add new tools and program sets, and runs all of its analyses inside Docker containers.

We have integrated and tested 4 tools in ECSTATIC (SOOT, WALA, DOOP, and FlowDroid). ECSTATIC found 74 bugs across these four tools. We have so far reported 42 of them to developers, who have confirmed that we found actual misbehaviors in the analysis tools. We have made the implementation of ECSTATIC available.[1]. We have also recorded a video demonstrating how to use ECSTATIC[2]. Additionally, we have published a technical paper describing ECSTATIC's approach and evaluation in detail [15].

2 ECSTATIC OVERVIEW

Figure 1 shows the architecture and process of ECSTATIC. ECSTATIC requires three inputs from the end user: the tool they wish to test, the input program set they wish to test, and other optional parameters, such as timeouts or number of processes to run in parallel. These inputs assume that the user has integrated the desired tool and input program set into ECSTATIC; the interfaces ECSTATIC exposes to allow integration of tools and input programs are described in Section 3. Given these inputs, ECSTATIC runs in two phases. In the testing phase, ECSTATIC iterates through the partial orders in the analysis tool's specification. For each partial order, ECSTATIC constructs two configurations that differ only in the option specified by the partial order; it then runs both configurations on each input program. ECSTATIC compares the outputs of these configurations against the expectations encoded by partial orders, and reports any violations. There are two stages of the testing phase. In the *base testing stage*, ECSTATIC uses the default configuration as its base to create new configurations. This means that every configuration run in the base testing stage will be, at most, one setting different from the default. In the *random testing stage*, ECSTATIC uses a grammar-based fuzzer to generate a random base configuration, which is then mutated to produce partially-ordered configurations. The base testing stage allows us to find bugs that exist in any option individually, while the random testing stage allows us to find bugs associated with more complex option interactions.

In the debugging phase, ECSTATIC applies a novel two-staged, violation-aware delta debugging algorithm to reduce input programs to their failure-inducing features. The two-staged approach addresses an important challenge in delta debugging static analysis inputs. Specifically, both compiling an input program and running static analysis on it can be slow, yet delta debugging relies on the ability to iteratively make and check many changes. Our two-staged delta debugger first applies a coarse strategy that makes modifications on the class level. Then, it applies more traditional hierarchical delta debugging at the statement level. The debugging is violation-aware as it uses the continued presence of a partial order violation as its criteria for accepting a change to the target program. Full details of these two phases are available in our technical paper [15].

As shown in Figure 1, the architecture of ECSTATIC is split into two levels. The backend of ECSTATIC consists of analyses run inside Docker [12] containers. This gives ECSTATIC the benefits of portability and reproducibility, since the analysis is containerized in an environment that is consistent across different machines. The end user interacts with the *dispatcher*, which is a lightweight, standalone program installed on the host machine. For each tool name that the user supplies, the dispatcher uses the tool's Dockerfile (see Section 3.1) to build a tool-specific image that contains an executable version of the tool. Then, for each benchmark that is specified, the dispatcher creates a new Docker container; within this container, ECSTATIC uses the benchmark's `build.sh` script (see Section 3.2) to build the benchmark. The dispatcher mounts a common result directory from the host machine onto each container. ECSTATIC's testing phase was implemented with 2646 lines of Python code, and its delta debugger, which can reduce Java applications and Android APKs, was implemented in 2123 lines of Java code. Instructions on running ECSTATIC are available at https://github.com/UTD-FAST-Lab/ECSTATIC.

3 TOOL AND BENCHMARK INTEGRATION

ECSTATIC exposes interfaces to integrate new tools and benchmarks, with the goal of making the integration process principled and easy. Using these interfaces, we have integrated 4 analyzers and 4 benchmarks in ECSTATIC (see Section 4). It typically requires only dozens of lines of code to integrate a new tool or benchmark.

3.1 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. This Dockerfile is responsible for downloading and setting up the analysis tool, including any of its dependencies.

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; however, most tools will only need to implement four of these methods, as shown in Figure 2. These methods specify the input program, the output file, the timeout, and the command to invoke the analyzer. Figure 2 shows the

[1] https://github.com/UTD-FAST-Lab/ECSTATIC
[2] https://youtu.be/siHp4PwUjbQ

```python
def get_input_option(self, benchmark_record: BenchmarkRecord) -> List[str]:
    output = f"--process-dir {benchmark_record.name}"
    if len(benchmark_record.depends_on) > 0:
        output = output + " " + f"--soot-class-path {':'.join(benchmark_record.depends_on)}"
    return output.split(" ")

def get_output_option(self, output_file: str) -> List[str]:
    return f'--callgraph-output {os.path.abspath(output_file)}'.split(" ")

def get_timeout_option(self) -> List[str]:
    if self.timeout is None: return []
    else:
        return f"--timeout {self.timeout*60*1000}".split(" ")

def get_base_command(self) -> List[str]:
    return "java -jar /SootInterface/target/SootInterface-1.0-SNAPSHOT-jar-with-dependencies.jar -pp -w -p cg.spark on-fly-cg:false,enabled:true".split(" ")
```

Figure 2: ECSTATIC's *AbstractRunner* interface and an implementation on SOOT.

```python
def import_file(self, file: str) -> Iterable[Flow]:
    return [Flow(f) for f in ElementTree.parse(file).getroot().find('flows').findall('flow')]
```

Figure 3: ECSTATIC's *AbstractReader* interface and an implementation on FlowDroid.

```
1 "name": "codeelimination",
2 "default": "PROPAGATECONSTS",
3 "levels": ["NONE", "PROPAGATECONSTS", "REMOVECODE"],
4 "orders": [{"left": "REMOVECODE", "order": "Precision", "right":
       "PROPAGATECONSTS"}]
```

Figure 4: An example of the partial order specification.

implementation of this interface for SOOT. Implementing the runner is fairly straightforward for most tools, as long as they accept inputs, outputs, and other options via the command line.

Third, ECSTATIC's interface called *AbstractReader* needs to be extended. This interface specifies how to read and compare the results of the target tool. Figure 3 shows an implementation of the interface for FlowDroid. 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 *AbstractReader* can be reused.

Fourth, the partial orders of analysis options (used to detect partial order violations) and configuration grammar (used by the grammar-based fuzzer) need to be specified in two JSON files. An example of the partial order specification is shown in Figure 4; this file defines the precision partial order *codeelimination.REMOVECODE* $\sqsubseteq_P$ *codeelimination.PROPAGATECONSTS* in FlowDroid (line 6).

3.2 Benchmark Integration Interface

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

Table 1: Configuration spaces of the 8 integrated tools and the lines of code it took to integrate them into ECSTATIC.

	SOOT	WALA	DOOP	FlowDroid
# Options	20	5	20	22
# Partial Orders	20	26	35	77
Integration LoC	90	53	111	156

Table 2: Partial order bugs detected in each tool by program set. The bar in each cell differentiates bugs detected in the base testing stage (left) and bugs detected only in the random testing stage (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

ECSTATIC also supports supplying ground truths with benchmarks. ECSTATIC expects these ground truths to be specified in a format that can be read by the *AbstractReader*. Providing ground truths will automatically switch ECSTATIC to a more precise ground-truth aware strategy.

4 EVALUATION

We have integrated 4 widely used static analysis tools for different programming languages into ECSTATIC: DOOP [8], SOOT [19], WALA [4], and FlowDroid [6]. Table 1 shows the lines of code required to integrate each tool. The number of lines of code to integrate a tool was on average 103 and at most 156 (for FlowDroid). Given that these analyzers have different input/output formats, this demonstrates it is easy to integrate a tool into ECSTATIC.

DOOP, SOOT, and WALA are all Java analysis frameworks. We chose call graph construction as our client analysis. We used two input program sets for these tools. First, the CATS microbenchmark [2], which contains 112 small Java programs that were constructed to test the soundness of Java call graph analyses, and DaCapo [7], which is a set of 11 real-world Java programs that are widely used to evaluate static analyses [11, 18, 20]. While ground truths are known for CATS, they were not in a format we could integrate into ECSTATIC. DaCapo does not have known ground truths. Therefore, we ran both program sets without ground truths.

For FlowDroid, we used two input program sets with ground truths. We used DroidBench [1], which is a microbenchmark consisting of 190 programs used to test taint analyzers. The ground truths were available in AQL-Answer [16] format, which is what FlowDroid's reader consumes. Second, we used an app, Alarm Klock, from the FossDroid [3] benchmark contributed by Mordahl and Wei [14], which contains manually classified ground truths.

4.1 Results

Table 2 shows the number of bugs detected by ECSTATIC's testing phase in the four analyzers we tested, broken down by program set. ECSTATIC identified 74 potential bugs; 10 were only found by the random testing stage. We have reported 42 bugs to developers, and are reporting the rest. Every developer response so far has confirmed that we found aberrant behavior. So far, 3 bugs in FlowDroid have been fixed, and fixes for WALA are in progress.

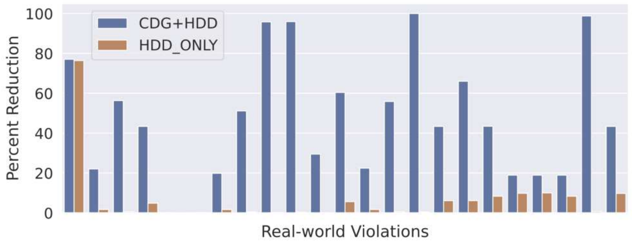

Figure 5: Our two-staged delta debugging (CDG+HDD) compared to hierarchical delta debugging (HDD-only) on real-world benchmarks.

Figure 5 shows the reduction achieved by our delta debugger on violations on real-world programs (i.e., FossDroid and DaCapo) with a 6-hour timeout. We did not run delta debugging on DOOP, because of memory and time requirements. Our two-staged delta debugging approach (CDG+HDD) was able to achieve significant improvements over hierarchical delta debugging (HDD_ONLY). The average reduction rate on real-world programs with our two-staged approach was 50%, as opposed to 6% when we only used HDD. At most, our two-stage delta debugger was able to reduce a real program by 99%, while HDD achieved <1% reduction.

4.2 Related Work and Limitations

There are two broad categories of related work. First, our work is related to those have focused on testing static analysis tools, e.g., [5, 21]. These works each focus on a single analysis, while ours proposes and implements an approach that can test any configurable analyzer. Another tool, Visuflow [10], aims to assist users debug static analysis. It gives general, bug-agnostic information (e.g., showing the IR and the call graph), while ECSTATIC provides artifacts (i.e., violations and reduced programs) that are specific to bugs that it detected. Second, various approaches apply metamorphic testing to compilers [9]; however, these works use equivalence relations (i.e., ensuring that a compiler produces equivalent executables for equivalent source code), while we utilize partial order relations. ECSTATIC is limited to finding bugs that exist in certain configurations; bugs that exist globally cannot be detected by our approach. Similarly, tools without algorithmic configurations cannot be tested by ECSTATIC.

5 CONCLUSIONS AND FUTURE WORK

ECSTATIC is a new framework for performing automatic testing and debugging of configurable static analysis tools. ECSTATIC leverages the precision and soundness partial orders between configuration option settings to perform its work, even without ground truths. We designed ECSTATIC with the goals of making it extensible and reproducible; ECSTATIC defines clear integration interfaces and runs all of its analyses inside Docker containers. ECSTATIC found 74 bugs across four popular static analysis frameworks for Java and Android. We plan to continue reporting bugs to developers, and to integrate more tools and benchmarks into ECSTATIC.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2021. DroidBench 3.0. https://github.com/FoelliX/ReproDroid.
[2] 2022. The Call-graph Assessment & Test Suite. https://bitbucket.org/delors/cats/src/master/.
[3] 2022. FossDroid. https://fossdroid.com.
[4] 2022. WALA. https://github.com/wala/WALA.
[5] Esben Andreasen and Anders Møller. 2014. Determinacy in Static Analysis for jQuery. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. 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 (PLDI '14)*. 259–269.
[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. 2006. 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*. 169–190.
[8] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. *SIGPLAN Not.* 44, 10 (oct 2009), 243–262. https://doi.org/10.1145/1639949.1640108
[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (feb 2020), 36 pages. https://doi.org/10.1145/3363562
[10] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2020. Debugging Static Analysis. *IEEE Transactions on Software Engineering* 46, 7 (2020), 697–709. https://doi.org/10.1109/TSE.2018.2868349
[11] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-Sensitive Points-to Analysis Using a BDD-Based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (oct 2008), 53 pages. https://doi.org/10.1145/1391984.1391987
[12] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.
[13] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (jan 2005), 1–41. https://doi.org/10.1145/1044834.1044835
[14] Austin Mordahl and Shiyi Wei. 2021. 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* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 466–477. https://doi.org/10.1145/3460319.3464823
[15] Austin Mordahl, Zenong Zhang, Dakota Soles, and Shiyi Wei. 2023. ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis. In *45th International Conference on Software Engineering (ICSE)*.
[16] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3236024.3236029
[17] Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ….
[18] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390
[19] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818
[20] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. 2018. Evaluating Design Tradeoffs in Numeric Static Analysis for Java. In *Programming Languages and Systems*. Springer International Publishing, 653–682. https://doi.org/10.1007/978-3-319-89884-1_23
[21] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. 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* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 487–498. https://doi.org/10.1145/2950290.2950338

Received 2023-05-18; accepted 2023-06-08