Slicer4J: A Dynamic Slicer for Java

Khaled Ahmed
Univ. of British Columbia, Canada
khaledea@ece.ubc.ca

Mieszko Lis
Univ. of British Columbia, Canada
mieszko@ece.ubc.ca

Julia Rubin
Univ. of British Columbia, Canada
mjulia@ece.ubc.ca

ABSTRACT

Dynamic program slicing is used in a variety of tasks, including program debugging and security analysis. Despite being extensively studied in the literature, the only dynamic slicing solution for Java programs that is publicly available today is a tool named JavaSlicer. Unfortunately, JavaSlicer only supports programs written in Java 6 or below and does not support multithreading. To address these limitations, this paper contributes a new dynamic slicing tool for Java, named Slicer4J. Slicer4J uses low-overhead instrumentation to collect a runtime execution trace; it then constructs a thread-aware, inter-procedural dynamic control-flow graph and uses the graph to compute the slice. To support slicing through Java framework methods and native code, Slicer4J relies on a set of pre-constructed data-flow summaries of the main framework methods. It also allows the users to further customize this set, adding user-defined methods when needed. We demonstrate the applicability of Slicer4J on ten benchmark and open-source Java programs, comparing it with JavaSlicer, and discuss how to use and extend the tool.

Tool package and demo: https://github.com/resess/Slicer4J

CCS CONCEPTS

• **Theory of computation** → **Program analysis**; • **Software and its engineering** → *Maintaining software.*

KEYWORDS

Program analysis, dynamic slicing, Java

ACM Reference Format:
Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Slicer4J: A Dynamic Slicer for Java. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473123

1 INTRODUCTION

Program slicing [22] computes the set of statements that affect a particular variable or statement of interest, often referred to as a *slicing criterion.* Slicing techniques are used in a variety of tasks, e.g., program debugging, to help locate the origin of an error. Slicing can be performed either statically or dynamically [14]. While static slicing considers all possible program paths leading to the slicing criterion, dynamic slicing focuses on one concrete execution.

The main idea behind a dynamic slicing tool is to first collect an *execution trace* of a program, i.e., the set of all executed program statements. Then, the tool inspects control and data dependencies of the trace statements, identifying statements that affect the slicing criterion and omitting the rest. The produced dynamic slices are more compact than static ones, making them particularly suitable for debugging activities [1, 2].

While there is a large number of papers proposing and optimizing dynamic slicing techniques, e.g., [6, 16, 17, 20], only one dynamic slicing solution of Java programs is publicly available at the time of writing: JavaSlicer [10, 11]. JavaSlicer is an efficient tool that is optimized for low instrumentation overhead with the capability of producing compressed traces for space optimization. However, the tool has several limitations: it does not support newer language constructs from Java versions above 6, e.g., lambda expressions; it does not support multithreading, producing instead one separate slice per thread; and it cannot handle certain Java framework methods, as well as framework and custom methods implemented in native code, e.g., `java.lang.System.arraycopy(...)`.

To address these issues, we contribute a dynamic slicing tool, named Slicer4J, which supports Java version 9, multithreading, and modeling of Java framework methods. Slicer4J relies on the infrastructure we created for slicing Android applications [3], adapting it to support stand-alone Java programs. The implementation and the documentation of Slicer4J is available online [4]. We also demonstrate the applicability of Slicer4J and compare it with JavaSlicer on ten Java programs.

Researchers and tool developers can use Slicer4J as a part of their dynamic analysis systems in a wide variety of projects, such as fault localization [6], regression analysis [21], and malware detection [15]. Software developers can also utilize Slicer4J for fault localization, e.g., to narrow down the scope of statements to inspect given failed test assertions or crash sites.

2 DYNAMIC SLICING

We now define slicing for Java programs on an example in Figure 1. This program calculates the lengths of two arrays, given as the program command line arguments. Its `main` method defines and spawns two threads of type `PThread`, which will read and parse the first and the second argument, respectively (lines 2-6). The parsed arrays are stored in the `arr` field of `PThread` (line 13). After the threads terminate (the check is omitted in line 7), the `main` method reads, sums up, and outputs the length of these arrays (lines 8-9).

Each thread is initialized with the array of input arguments `args` and an index within this array `index` (lines 15-16). When the thread starts executing, it checks if the `index` is within the range of the input parameters and, if so, populates the array using a helper function `parse`, which we omit here for brevity (lines 19-20).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473123

```
 1 public class SliceMe {
 2   public static void main(String[] args) {
 3     PThread p1 = new PThread(args, 0);
 4     PThread p2 = new PThread(args, 1);
 5     new Thread(p1).start();
 6     new Thread(p2).start();
 7     ...
 8     int length = p1.arr.length + p2.arr.length;
 9     System.out.println(length);
10 }}
11 class PThread implements Runnable {
12   String[] args; int index;
13   int[] arr = null;
14   PThread(String[] args, int index){
15     this.args = args;
16     this.index = index;
17   }
18   void run() {
19     if (index < args.length){
20       this.arr = parse(args[index]);
21 }}}
```

Figure 1: SliceMe: a faulty app.

The bug in this program occurs when one or both input parameters are omitted. For example, if the second parameter is omitted, `p2.arr` is not initialized, leading to an exception when calculating `p2.arr.length` in line 8. Slicing from the statement in line 8 produces the subset of executed program statements which contribute to the failure. These are the statements highlighted in the figure. We define the slice more formally next.

For simplicity of presentation, we refer to a statement in line i of our examples as s_i, e.g., the `if` statement in line 19 is denoted by s_{19}. When a program runs, each statement can be triggered multiple times during the execution, e.g., in multiple iterations of a loop or in different instances of a thread. We refer to each individual execution of a statement as a *statement instance* and denote the k^{th} execution of a statement s_i as s_i^k. For example, assuming that `p1` is executed before `p2`, the trace will contain statement instances s_3^1, s_{13}^1, s_{15}^1, s_{16}^1, s_4^1, s_{13}^2, s_{15}^2, s_{16}^2, s_5^1, s_6^1, s_{19}^1, s_{20}^1, s_{19}^2, s_8^1. We refer to the full sequence of statement instances from a particular app run as an *execution trace.*

Dynamic control-flow dependencies represent a concrete transfer of control recorded during the program execution. We say that a statement instance s_j^m is *control-flow-dependent* on s_i^k if s_j^m is executed immediately after s_i^k in the same execution thread [5]. For the example in Figure 1, s_{16}^1 is control-flow-dependent on s_{15}^1.

A sequence of statement instances with no jump statement (i.e., conditional or method call), except at the end of the sequence, is referred to as a *basic block.* For example, s_{15}^1 and s_{16}^1 are in the same basic block, while s_{19}^1 and s_{20}^1 are not.

A statement instance s_j^m is *control-dependent* on s_i^k if s_i^k can alter the program's control and it determines whether s_j^m executes [9]. In Figure 1, s_{20}^1 is control-dependent on s_{19}^1, as the outcome of the `if` determines whether the control reaches s_{20}^1 or not.

A statement instance s_j^m is a *data-flow-dependent* on s_i^k w.r.t. the variable v used in s_j^m if and only if s_i^k defines v and no other statement redefines v between s_i^k and s_j^m in the trace [1]. In Figure 1, s_8^1 is data-flow-dependent on s_{20}^1 w.r.t. `p1.arr`, as the statement in line 20 defines the `p1.arr` variable used in line 8.

A *slicing criterion* for an execution trace is a statement instance and all variables of interest used in this statement instance [14]. For the example in Figure 1, the app throws a `NullPointerException` in line 8, making line 8 with all variables that it uses – `p1.arr` and `p2.arr` – a suitable slicing criterion for the developer interested in finding the cause of the crash.

A *backward dynamic slice* [14] is the set of all statement instances whose execution affects the slicing criterion, i.e., the set of instances on which the slicing criterion is control- or data-flow-dependent, either directly or transitively. For the example in Figure 1, the backward slice from s_8^1 w.r.t. variable `p1.arr` consist of s_8^1, s_{20}^1, s_{19}^1, s_{16}^1, s_{15}^1, and s_3^1. The backward slice from s_8^1 w.r.t. variable `p2.arr` consist of s_8^1 and s_{13}^2. The source code lines corresponding to these statement instances are highlighted in the figure.

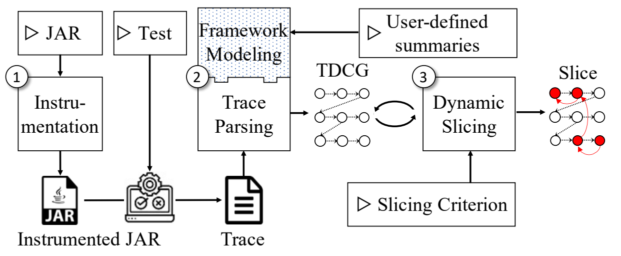

Figure 2: Slicer4J overview.

3 SLICER4J DESIGN

We now describe the main design decisions behind Slicer4J. Figure 2 gives its high-level overview. Our tool receives the following inputs: (1) a Java Archive (JAR) file of the program to slice, (2) a slicing criterion, e.g., a statement that throws an exception, (3) a test that triggers this criterion, and (4) an optional list of user-defined method summaries, as described below. It produces as output a backward dynamic slice for the provided slicing criterion.

Instrumentation. As the first step, Slicer4J runs a lightweight static analysis to instrument the JAR. Our instrumentation relies on Soot [18] and works on the Jimple intermediate code representation [19]. It records a unique identifier of each execution thread (using `Thread.getId()`) and for each statement within the thread. Slicer4J also includes an option to perform basic-block-level instrumentation, which records a unique identifier of each basic block rather than each statement within the thread [13]. Such implementation relies on the assumption that control is unlikely to switch from one thread to another within a basic block; it thus should not be used in heavily multi-threaded code.

While every local variables can be uniquely identified within its declaring method, shared variables (e.g., object fields) may have different names in the same/different methods. We thus follow Agrawal et al. [1] and record unique identifiers for fields (with `java. lang.System.identityHashCode` method), which we use to identify fields when calculating data-flow dependencies.

As Java framework code is not part of the input JAR, it cannot be instrumented by Slicer4J. To still enable slicing through framework methods, Slicer4J relies on statically-generated Java framework method summaries, which we borrow from FlowDroid [8] and StubDroid [7]. These summaries, originally designed for taint analysis, define how a taint originates from each method input (i.e., fields and parameters) and propagates to its output (i.e., fields, parameters, and return value). We recast the taint-flow summaries as assignment statements, which our analysis uses to find data-flows.

For example, the StubDroid summary for the method `java.lang.System.arraycopy(a,srcPos,b,...)` will state that a taint from the first array reaches the second array b. We use this information to represent a method as an assignment `b = a`, expressing the fact that the values of the first array are copied to the second array (directly or indirectly). Slicer4J also provides the option to augment this initial framework method summaries with a user-defined list of additional summaries, e.g., to model user-defined native methods.

An alternative solution, employed by JavaSlicer, is to instrument Java classes when they are loaded by the Java Virtual Machine (JVM). The main advantage of this solution is that it can handle framework methods and dynamically generated code. Its main disadvantage is that it cannot handle Java methods used as part of its instrumentation logic, e.g., `java.lang.String`: instrumenting these classes would cause an infinite loop when producing the trace. Moreover, such instrumentation has to be repeated every time a trace is generated, which slows things down when a program needs to be sliced multiple times and for different slicing criteria. Because of these limitations, we opt to statically instrument JARs and model framework/native methods. We plan to add handling of dynamically generated code as part of future work.

Trace Parsing. The instrumented JAR is executed (from command line or with a test script) to produce a trace. Slicer4J parses the trace and builds a Thread-aware, Inter-procedural Dynamic Control-flow Graph (TDCG), whose nodes are statement instances of each executed basic block. The graph has three types of unidirectional edges. *Control-flow edges* connect statement instances with the same thread, in their order of execution. *Thread-control-flow edges* connect statement instances from different threads, if they were executed immediately after each other in the trace. Finally, *control-dependence edges* connect statement instance s_j^m to the instance s_i^k iff s_i^k can alter the execution of s_j^m. We rely on a static control dependency graph produced by Soot and create a control-dependence edge between s_j^m and s_i^k iff s_j is control-dependent on s_i in the static control dependency graph.

As the last step, we replace each statement instance that invokes a framework method (or any other method in the user defined list of methods summaries) with its corresponding summary, which we treat as a set of assignments for slicing purposes.

Dynamic Slicing. The TDCG is given as the input to the dynamic slicing step. We follow the backward slicing algorithm by Agrawal et al. [1]. The main objective of the algorithm is to identify control and data-flow dependencies of all variables used in the slicing criterion. In a nutshell, it works in an iterative manner, initializing the working set W to include variables used in the slicing criterion. It then transitively adds all variables used in their control and data dependencies to W. The algorithm terminates when dependencies of all variables in W are identified.

For identifying the control dependency of a statement instance, we simply follow the control-dependence edges in TDCG. For data-flow dependencies, we separate between local and shared variables (i.e., object fields). The scope of local variables is within a single method/thread. Thus, to identify the definition of a local variable, we simply traverse control-flow edges until the definition of a variable with the same name is found. Shared variables can be read/written in different threads. We thus traverse both control-flow and thread-control-flow edges, to find the definition of a variable with the same memory address; when a statement instance has outgoing edges of both kinds, a thread-control-flow edge takes preference over control-flow edges, to accurately represent the sequence of statement execution in the trace. For completeness, we add the identified data-flow-dependency edges to TDCG.

Outputs. At the end of this process, Slicer4J maps each statement instance in the produced slice back to its corresponding line of source code and outputs the list of source code lines. It also outputs the raw slice representation, where each line corresponds to a statement instance and has three fields: the name of the source file and the line number within the file for the statement instance; the thread of the instance; and the statement instance itself, in Jimple format (see Figure 3). Finally, Slicer4J also makes TDCG available for further inspection of the relevant control and data-flow dependencies.

```
SliceMe:3   1   p1 = new Thread
SliceMe:3   1   p1.<init>(args, 0)
SliceMe:15  2   this.args = args
SliceMe:16  2   this.index = index
                  . . .
```

Figure 3: Slice snippet

Slicer4J is launched from the command line and provides several slicing options and configuration parameters, which are described in our tool package [4]. Information about hardware and software specification is also available online.

4 EVALUATION

We demonstrate the applicability of Slicer4J by running it on a number of subject applications.

Benchmark Programs. First, we borrowed three benchmark applications created by the JavaSlicer authors to evaluate their tool. These applications check the ability of a slicing solution to handle data-flows within a method (intra-procedural), data-flows across methods (inter-procedural), and control-dependence of exception blocks on the statements that throw exceptions.

We created four additional benchmark applications to verify capabilities introduced in Slicer4J and not supported by JavaSlicer: tracking data flows through multiple threads; Java 9 constructs, e.g., lambda expressions; native methods; and framework methods used by the JavaSlicer instrumentation. The list of the benchmarks, together with the size of each benchmark in terms of source lines of code (LoC), is given in the first two columns of Table 1. The benchmarks are small, ranging from 9 to 44 LoC.

As slices produced by JavaSlicer are at the Java bytecode level and contain both the program and framework code, for fair comparison between the tools, we map both slices to the source-code level and disregard statements within the Java framework code. The last three columns of Table 1 show the size of the expected slice, as well as the size of slices produced by JavaSlicer and Slicer4J.

The table shows that both tools produce the expected slice for the three JavaSlicer benchmarks, but JavaSlicer fails to produce the expected slice on the last four cases handled by Slicer4J due to its limitations outlined in Section 1. While Slicer4J finds the correct slice in all cases, it has an extra statement in the slice of the *Native methods* and *Instrumentation classes* benchmarks. In both cases, this is caused by an overapproximation in the framework method modeling: as this model is created by static analysis of the framework methods, it assumes that all possible execution paths

Table 1: Accuracy: Benchmarks.

Benchmark		Slice size (LoC)		
Name	LoC	exp.	JavaSlicer	Slicer4J
Intra-procedural	9	5	5	5
Inter-procedural	12	5	5	5
Exceptions	30	8	8	8
Multiple threads	44	4	3	4
Native methods	10	5	2	6
Java 9 constructs	17	5	2	5
Instrumentation classes	12	4	2	5

within the framework may be taken at runtime, which is not the case for concrete executions.

Defects4J Programs. To further demonstrate the applicability of Slicer4J and compare it with JavaSlicer on bigger examples, we selected the top three open-source projects from the Defects4J dataset [12], ordering the projects in this dataset by the number of project usages on GitHub. The list of selected projects, together with their size, is shown in the first two columns of Table 2.

Each project in the Defects4J dataset contains a list of bugs, a test case that triggers each of these bugs, and an annotated fix for each bug. We examined the bugs one-by-one, selecting the first bug on the list where at least one buggy line is executed in the trace, i.e., where fixes involve modifying or replacing a line of code. We then used the failing test assertion statement as a slicing criterion and checked whether the buggy lines indeed appears in the slice produced by both Slicer4J and JavaSlicer. The number of buggy lines in the trace per project is shown in the third column of Table 2. The size of the execution trace for the test triggering the bug is in the fourth column, while the time to produce the trace, in seconds (without any instrumentation) is in the fifth column.

Next, we show the number of buggy lines found by each tool and the size of the slice that each tool produces. While in all three cases both JavaSlicer and Slicer4J found the expected buggy lines (these are older programs, which are written in Java 6 and contain no threads), the slices produced by Slicer4J are substantially smaller in the first two cases. That is mostly because of the extra unnecessary data-flow dependencies that JavaSlicer adds due to inaccurate modeling of bytecode instructions. For example, when looking for data-flow dependencies of `y.f` in the code `y.f = x`, the variable `x` should be added to the working set W. Yet, JavaSlicer adds `y` to W as well (and further collects all it transitive dependencies).

For the last subject, JavaSlicer and Slicer4J have identical slices besides one statement missing for JavaSlicer. This statement is a definition of a static field inside a static constructor, which is correctly identified by Slicer4J. JavaSlicer does not handle static fields in static constructors correctly, which we confirm with a mini-benchmark that we made available online [4].

Finally, we measured instrumentation (Inst.), execution (Exec.), and slicing times for Slicer4J for each subject. Unlike Slicer4J, which instruments the JAR file, JavaSlicer attaches to the JVM and instruments each class as it is loaded. We thus measure Inst.+Exec. and slicing times for JavaSlicer for each subject. All measured times, for both of the tools are shown in Table 2.

The combined instrumentation and execution time for Slicer4J is lower than that of JavaSlicer in the last two subject programs but is higher in the first one. This program is relatively large (more than 46,000 LoC) but the execution trace is small (only 3,927 lines), meaning that only a small portion of the program is executed. As JavaSlicer instruments on-demand, it saves a substantial portion of instrumentation, leading to a better performance in this case.

For the other two cases, the trace is substantially larger than the code size. In such cases, execution time "dominates" the instrumentation time and thus the efficient instrumentation capabilities of Slicer4J help to bring the overall processing time down. We also do not slow down framework code by instrumenting it.

The higher slicing time of Slicer4J is the cost of the Slicer4J ability to handle data-flows across threads. Unlike JavaSlicer, Slicer4J processes the entire trace to untangle threads and create the TDCG, which is used for more accurate slicing. The overall slicing time for Slicer4J is less than five minutes, which we still find acceptable.

Limitations. Slicer4J is unable to slice classes that are dynamically generated at runtime. We plan to address this limitation by adding a JVM instrumentation component that instruments such classes on-demand. Moreover, Slicer4J's slicing time can be reduced by generating TDCG on-demand instead of pre-processing the entire trace before slicing starts. Slicer4J can map the Jimple slice to the source code only if the JAR file is compiled with debug information (preserving line numbers); otherwise, Slicer4J outputs the slice in the raw Jimple format. The interleaved writes to the same trace by multiple threads reduce parallelism as threads have to synchronize when writing to the trace, which can slow applications down. This synchronization can also "hide" concurrency bugs that occurred in the uninstrumented version of the program. Finally, Slicer4J relies on Soot to instrument the program and thus is limited to the Java 9 constructs supported by Soot.

5 CONCLUSIONS

This paper presents Slicer4J – an accurate and efficient tool for dynamic slicing of Java 9 programs. Slicer4J uses low-overhead instrumentation to collect a runtime execution trace. It then constructs a thread-aware, inter-procedural dynamic control-flow graph and uses the graph to compute the slice, while relying on pre-defined data-flow summaries of the main framework methods. We demonstrate the applicability of Slicer4J on ten benchmark and open-source Java programs, comparing it with JavaSlicer – the only other openly available slicing solution for Java 6 programs.

Table 2: Defects4J Programs.

Project					JavaSlicer				Slicer4J				
Name: Bug ID	LoC	# Buggy lines	Trace (LoC)	Exec. time (s)	# Buggy lines	Slice (LoC)	Time (s)		# Buggy lines	Slice (LoC)	Time (s)		
							Inst. + Exec.	Slicing			Inst.	Exec.	Slicing
JacksonDatabind:3	46,091	1	3,927	0.27	1	367	18.6	3.7	1	58	35.4	0.3	19.2
Gson:4	7,639	2	580,143	0.21	2	311	29.4	42.2	2	13	7.1	3.2	257.5
JacksonCore:4	15,667	1	809,814	0.21	1	32	16.8	38.7	1	33	9.6	3.4	129.3

REFERENCES

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. 1991. Dynamic Slicing in the Presence of Unconstrained Pointers. In *Proc. of the Symposium on Testing, Analysis, and Verification (TAV)*. 60–73. https://doi.org/10.1145/120807.120813

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. *ACM SIGPLAN Notices* 25, 6 (1990), 246–256. https://doi.org/10.1145/93542.93576

[3] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Mandoline: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*. 105–115. https://doi.org/10.1109/ICST49551.2021.00022

[4] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Slicer4J. https://github.com/resess/Slicer4J

[5] Frances E. Allen. 1970. Control Flow Analysis. *ACM SIGPLAN Notices* 5, 7 (1970), 1–19. https://doi.org/10.1145/800028.808479

[6] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. 2011. Fault-Localization Using Dynamic Slicing and Change Impact Analysis. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 520–523. https://doi.org/10.1109/ASE.2011.6100114

[7] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic Inference of Precise Data-flow Summaries for the Android Framework. In *Proc. of the International Conference on Software Engineering (ICSE)*. 725–735. https://doi.org/10.1145/2884781.2884816

[8] 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 *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 259–269. https://doi.org/10.1145/2666356.2594299

[9] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[10] Clemens Hammacher. 2008. Design and Implementation of an Efficient Dynamic Slicer for Java. Bachelor's Thesis.

[11] Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. 2009. Profiling Java Programs for Parallelism. In *Proc. of the ICSE Workshop on Multicore Software Engineering (IWMSE)*. 49–55. https://doi.org/10.1109/IWMSE.2009.5071383

[12] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 437–440. https://doi.org/10.1145/2610384.2628055

[13] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. 1992. Interprocedural Dynamic Slicing. In *Proc. of the International Symposium on Programming Language Implementation and Logic Programming (PLILP)*. 370–384. https://doi.org/10.1007/3-540-55844-6_148

[14] Bogdan Korel and Janusz Laski. 1988. Dynamic Program Slicing. *Inform. Process. Lett.* 29, 3 (1988), 155–163. https://doi.org/10.1016/0020-0190(88)90054-3

[15] Andrea Lanzi, Monirul I. Sharif, and Wenke Lee. 2009. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 1–16.

[16] Xiangyu Li and Alessandro Orso. 2020. More Accurate Dynamic Slicing for Better Supporting Software Debugging. In *Proc. of the International Conference on Software Testing, Validation and Verification (ICST)*. 28–38. https://doi.org/10.1109/ICST46399.2020.00014

[17] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based Statistical Fault Localization. *Journal of Systems and Software* 89 (2014), 51–62. https://doi.org/10.1016/j.jss.2013.08.031

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. 1–11.

[19] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. *Sable Technical Report* (1998).

[20] Tao Wang and Abhik Roychoudhury. 2004. Using Compressed Bytecode Traces for Slicing Java Programs. In *Proc. the International Conference on Software Engineering (ICSE)*. 512–521.

[21] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. 2010. Analyzing Concurrency Bugs Using Dual Slicing. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 253–264. https://doi.org/10.1145/1831708.1831740

[22] Mark Weiser. 1981. Program Slicing. In *Proc. of the International Conference on Software Engineering (ICSE)*. 439–449.