When Dynamic Slicing Helps and Hurts Spectrum-Based Fault Localization: Evidence from an Expanded TCAS Benchmark

Iulia Nica
Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, Austria

Franz Wotawa
Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, Austria

Abstract

Spectrum-based fault localization and dynamic program slicing are two established approaches to automated software debugging. A recent study provided a first experimental evaluation of a hybrid approach that combined both techniques, reporting that the combination improves the wasted effort metric for programs with multiple output variables. However, as stated in the study, further experiments are required for a final judgment of the introduced debugging methodology. In this paper, we review and extend that evaluation through three rounds of experiments, all carried out under a strict single-fault assumption. First, we replicate the third experiment from the original study to validate the obtained findings. Second, we expand the TCAS benchmark from 2 to 11 faulty variants by systematically generating nine new mutants spanning six distinct mutation operator families, and verify their quality through a dedicated kill-ability assessment. Eleven of the twelve mutants pass our quality filter and are used in two independent rounds of extended experiments , each evaluated with three spectrum-based fault localization coefficients (Ochiai, Tarantula, Sarhan-Beszédes) in both the plain and hybrid configurations. Our results confirm that the slicer's behavior has a decisive impact on the quality of the hybrid approach and that improvements from slicing are far from uniform across mutation types. In particular, bugs that suppress the execution of entire program paths present a fundamental challenge for dynamic slicing, providing concrete evidence for the limitations reported in the original study.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Computing methodologies → Causal reasoning and diagnostics

Keywords and phrases software fault localization, dynamic slicing, spectrum-based fault localization, mutation testing, automated debugging, TCAS

Digital Object Identifier 10.4230/OASIcs.DX.2026.17

Category Short Paper

Funding *Franz Wotawa*: The work was supported by the Austrian Science Fund (FWF) Cluster of Excellence Bilateral AI under contract number 10.55776/COE12.

1 Introduction

Automated software debugging remains one of the most practically relevant open problems in software engineering. Among the approaches proposed to support fault localization, spectrum-based fault localization (SFL) [6, 2] and dynamic program slicing [8] have each attracted substantial research attention. SFL assigns suspiciousness scores to program statements based on their co-occurrence with passing and failing test outcomes, while dynamic slicing identifies statements that causally contribute to a given output value for a specific test execution.

Schleich and Wotawa [11] recently provided the first experimental evaluation of a hybrid approach that integrates dynamic slicing into SFL, implemented within the open-source JSR framework [12]. Their study considered three sets of Java programs, including the well-known


© Iulia Nica and Franz Wotawa;
licensed under Creative Commons License CC-BY 4.0

TCAS collision avoidance benchmark. A key finding was that for programs with more than one observable output variable, the hybrid approach consistently improves the wasted effort metric compared to plain SFL, while the impact of the underlying slicer tool was identified as a significant source of variability.

However, the evaluation of the hybrid approach on TCAS programs was limited to only two faulty variants. Given that the TCAS program has a well-documented history as a mutation testing benchmark [4], a broader evaluation covering more diverse mutation operator families would be valuable for assessing the generality of the findings.

This paper makes the following contributions:

- We *replicate* the third experiment of [11], confirming that the published results are reproducible under our independent infrastructure (Section 3).
- We *construct an expanded TCAS mutation benchmark* by adding *nine new faulty variants* to the original two, spanning six distinct mutation operator families drawn from the spreadsheet mutation testing literature [1]. Each variant is exercised by a parameterized JUnit test suite of *40 test cases*, structured as eight input combinations × five output variable assertions. The full benchmark comprises 480 instrumented test executions and 2,400 individual output assertions per round of experiments (Section 4).
- We perform a *quality assessment* of the generated mutants through a dedicated kill-ability study and select eleven variants for the SFL evaluation (Section 4.4).
- We *conduct two independent rounds of experiments* on the expanded benchmark (EE1 and EE2), each evaluating three SFL coefficients (Ochiai, Tarantula, Sarhan-Beszédes) in both plain-SFL and hybrid (sliced) configurations (Section 5).
- We *identify a structural explanation* for the cases in which dynamic slicing degrades SFL accuracy: mutations that suppress execution of entire conditional blocks systematically defeat the slicer (Section 6).

Throughout all experiments we adopt the *single-fault assumption*, i.e., each faulty program variant contains exactly one intentionally introduced defect.

The remainder of this paper is structured as follows. Section 2 briefly recalls the foundations of SFL and dynamic slicing. Section 3 presents the replication of the third experiment from the original study. Section 4 describes the construction of the expanded TCAS benchmark, including mutant generation, test suite design, and kill-ability assessment. Section 5 reports the two rounds of extended experiments. Section 6 discusses the findings and their implications, Section 7 addresses threats to validity, and Section 8 concludes.

2 Background

We briefly review the foundational concepts relevant to this paper, for a more detailed discussion, we refer the reader to [11].

2.1 Spectrum-based Fault Localization

SFL uses the *program spectrum*, a matrix recording for each test case and each program statement whether the statement was executed, together with an error vector recording whether each test passed or failed. From these data, four aggregate counts $a_{nm}(i)$ are computed for each statement i, where $n \in \{0, 1\}$ indicates whether the statement was executed and $m \in \{0, 1\}$ indicates whether the test passed (0) or failed (1). These counts are combined into a suspiciousness coefficient. We evaluate three coefficients: Tarantula [6], Ochiai [2], and Sarhan-Beszédes [9].

Debugging quality is measured by the *rank* of the faulty statement and the *wasted effort* (WE), defined as

$$WE = |\{s : \text{score}(s) > \text{score}(s_f)\}| + 0.5 \cdot \big(|\{s : \text{score}(s) = \text{score}(s_f)\}| - 1\big), \tag{1}$$

where s_f is the faulty statement. Lower WE is better. We also report Hit Ratio@k (Hit@k), the fraction of programs for which the fault is ranked within the top k positions.

2.2 Dynamic Slicing and the Hybrid Approach

Dynamic slicing [8] computes, for a given execution and a chosen output variable, the set of statements that causally influenced that variable's value. The hybrid approach of [11] decomposes each test case into one sub-test per output variable and uses the dynamic slice for each output to restrict which statements are considered in the SFL computation for that sub-test. This allows the approach to exploit data-dependency information that plain SFL ignores.

The extended version of the JSR framework [10] implements this hybrid approach for Java programs, employing JaCoCo [5] for line coverage analysis and Slicer4J [3] for dynamic slicing. This implementation builds upon the original JSR framework developed by Stracke [12, 7].

3 Round 1: Replication of the Third Experiment

The third experiment of [11] used the same Java programs as the initial experiment, but increased the number of observable output variables per program. For the TCAS program, five output variables were considered: `alim`, `inhibit_biased_climb`, `non_crossing_biased_climb`, `non_crossing_biased_descend`, and `return_value`. Each input combination thus yields five assertions, and the JUnit test suite consists of 40 parameterized test cases per TCAS variant (5 assertions $\times$ 8 input combinations).

3.1 Replication Procedure

We executed the third experiment using the original `smallProject` provided with the JSR repository, retaining the two original TCAS variants (`Tcas`, `Tcas2`) alongside the remaining programs of the benchmark. The JSR pipeline was driven through the IntelliJ plugin (with *Coverage Metric: checked, Reduction Algorithm: Greedy HGS, Keep 0 coverage, Deactivate redundant test cases*) and the subsequent SFL matrix computation was triggered through the JSR command-line interface. The post-processing was performed with the original `SFL.py` script.

3.2 Replication Outcome

For the TCAS rows of *Tables 13* and *14* of [11], our independently obtained ranks and wasted-effort values coincide with those reported in the original study, confirming the reproducibility of the published experimental setup. This replication serves two purposes: it establishes the baseline against which the extended experiments in Section 5 can be compared, and it validates that the JSR tool chain behaves consistently across installations.

It is worth noting that the replication required two minor adjustments to the build environment. First, the dependency `soot-inflow-2.10.0-mandoline.jar` was not available in the standard Maven repository and had to be installed manually, with the corresponding version entry in `pom.xml` updated accordingly. Second, a compile error in `ThreadCalls.java`

had to be resolved: the method `MethodSubSignature(subsig)` is no longer present in newer versions of Soot, requiring a small adaptation of the `ThreadCall.java` source file. Neither of these adjustments affected the experimental results; they reflect the natural evolution of third-party dependencies over time and are documented here to assist future replicators of this work.

4 An Expanded TCAS Mutation Benchmark

The original study relied mostly on programs with fewer than five output variables, the single example with five outputs being the TCAS, as defined in Section 3.3 of [11]. To address this, we constructed an expanded benchmark of eleven TCAS variants by generating *nine new mutants* and adding them to the two preexisting ones. This section documents the construction process, the test suite that exercises each variant, and the quality assessment used to filter weak mutants before the SFL experiments.

4.1 Existing TCAS Variants

The JSR repository ships with two families of TCAS variants: a set of single-output programs and a set of five-output programs. We work exclusively with the five-output family (`Tcas`, `Tcas2`, `Tcas3`), which is the one relevant to the third experiment of [11]. A close inspection of the `classes` folder used by the `testall.py` script reveals two inconsistencies that must be acknowledged before any extension can be made:

1. **Inconsistent fault-marker placement.** The `//Error` comment marking the faulty line is placed in `Tcas.java` (the buggy original), but in the `Tcas2Altered` and `Tcas3Altered` files for the second and third variants. The non-annotated files in those cases are the correct versions. All three correct (non-annotated) files are textually identical; only the altered versions differ.
2. **Compound mutation in `TcasAltered`.** The first altered variant introduces two simultaneous changes to line 59: `+NOZCROSS` becomes `-NOZCROSS` (an AOR-style operator replacement) *and* a division by 1 is added. Technically, this violates the single-fault assumption. We retain only the AOR-style operator replacement in our further experiments.

For all new variants we adopt the convention that each `TcasN.java` ($N \in \{4, \ldots, 12\}$) is the correct version with no `//Error` annotation, and the corresponding `TcasNAltered.java` contains exactly one fault marked with a `//Error` comment.

4.2 Mutation Operator Selection

To ensure a systematic and reproducible mutant generation process, we adopt the mutation operator taxonomy of Abraham and Erwig [1], originally introduced for end-user spreadsheet mutation testing but directly applicable to imperative Java code. Table 1 summarizes the sixteen operators of this taxonomy.

From this taxonomy we selected six operator families for our new mutants: AOR, CRP, LCR, ROR, FRC, and UOI. The selection criterion was that the operator should be meaningfully applicable to the imperative control- and data-flow constructs of the TCAS code, so that resulting faults are realistic representatives of typical programmer mistakes (sign errors, incorrect constants, swapped predicates, miss-negations, missing operands, etc.). Range-oriented operators (CRS, NRS, CRE, NRE, RRR) are not directly applicable to the TCAS code, which does not manipulate ranges; we therefore omit them from the mutant generation.

Table 1 Mutation operator taxonomy of Abraham and Erwig [1] used as a reference for selecting diverse fault types.

Operator	Description
ABS	ABSolute value Insertion
AOR	Arithmetic Operator Replacement
CRP	Constants RePlacement
CRR	Constants for Reference RePlacement
LCR	Logical Connector Replacement
ROR	Relational Operator Replacement
RCR	Reference for Constant Replacement
FDL	Formula DeLetion
FRC	Formula Replacement with Constant
RFR	ReFerence Replacement
UOI	Unary Operator Insertion
CRS	Contiguous Range Shrinking
NRS	Non-Contiguous Range Shrinking
CRE	Contiguous Range Expansion
NRE	Non-Contiguous Range Expansion
RRR	Range Reference Replacement

4.3 Generated Mutants

Table 2 summarizes all twelve TCAS variants used in our study, including the three from the original work [11] (`TcasAltered`, `Tcas2Altered`, `Tcas3Altered`) and the nine *new* variants generated for this work (`Tcas4Altered`–`Tcas12Altered`). For each mutant the table indicates the mutation operator family, the modified source line, and the exact textual change applied.

Table 2 The twelve TCAS mutants used in this study. Mutants in italics (`Tcas4`) are excluded from the SFL experiments based on the kill-ability assessment of Section 4.4.

File	Line	Operator	Mutation detail
TcasAltered.java	59	AOR	`+ NOZCROSS` → `- NOZCROSS`
Tcas2Altered.java	89	CRP	`upward_preferred = 1` → `= 0`
Tcas3Altered.java	130	CRP	`UPWARD_RA` → `UNRESOLVED`
Tcas4Altered.java	132	CRP	`DOWNWARD_RA` → `UNRESOLVED`
Tcas5Altered.java	125	LCR	`Own_Below_Threat()` → `Own_Above_Threat()`
Tcas6Altered.java	126	LCR	`Own_Above_Threat()` → `Own_Below_Threat()`
Tcas7Altered.java	117	ROR	`High_Confidence != 0` → `== 0`
Tcas8Altered.java	117	ROR	`Own_Tracked_Alt_Rate <= OLEV` → `>=`
Tcas9Altered.java	117	ROR	`Cur_Vertical_Sep > MAXALTDIFF` → `<`
Tcas10Altered.java	59	FRC	`Up_Separation + NOZCROSS` → `Up_Separation + 50`
Tcas11Altered.java	125	UOI	`Non_Crossing_Biased_Climb()` → `!Non_Crossing_Biased_Climb()`
Tcas12Altered.java	59	UOI	`Up_Separation + NOZCROSS` → `-Up_Separation + NOZCROSS`

The nine new mutants distribute across the operator families as follows: two LCR mutations (Tcas5, Tcas6) targeting logical connectors in predicates that determine the climb/descent direction, three ROR mutations (Tcas7, Tcas8, Tcas9) targeting relational operators in the guard expression of the main resolution-advisory block, one CRP mutation (Tcas4) substituting a constant value, one FRC mutation (Tcas10) replacing a formula sub-expression with a constant literal, and two UOI mutations (Tcas11, Tcas12) intro-

ducing unary negation operators in different program locations. Together with the three inherited mutants this yields six distinct operator families across twelve fault locations, distributed over the three core decision procedures of TCAS: `Inhibit_Biased_Climb()`, `Non_Crossing_Biased_Descend()`, and `alt_sep_test()`. This represents a four-fold increase in mutant diversity over the original study on TCAS programs.

4.4 Kill-ability Assessment

A mutation testing benchmark is only useful if its mutants are actually *kill-able*, i.e., distinguishable from the correct program by at least one test case in the suite. We assessed kill-ability using the `testall.py` script from the JSR repository, which executes each test case against both the original and the altered program and counts passing and failing test cases. Table 3 reports the results for all twelve mutants.

Table 3 Kill-ability assessment of the twelve TCAS mutants. Each mutant is run against all 40 test cases of the parameterized JUnit suite. A higher number of failing test cases corresponds to a stronger (more easily killed) mutant. The kill rate is the fraction of failing test cases. Tcas4 is flagged as a weak mutant and excluded from the SFL experiments.

Variant	Operator	Passing	Failing	Kill rate
Tcas	AOR	35	5	12.5%
Tcas2	CRP	38	2	5.0%
Tcas3	CRP	38	2	5.0%
Tcas4	*CRP*	*39*	*1*	*2.5% (weak, excluded)*
Tcas5	LCR	36	4	10.0%
Tcas6	LCR	38	2	5.0%
Tcas7	ROR	29	11	27.5% (strong)
Tcas8	ROR	29	11	27.5% (strong)
Tcas9	ROR	23	17	42.5% (strong)
Tcas10	FRC	37	3	7.5%
Tcas11	UOI	37	3	7.5%
Tcas12	UOI	33	7	17.5%

4.4.1 Quality Observations

Three observations follow from Table 3. First, the three ROR mutants (Tcas7, Tcas8, Tcas9) yield the highest number of failing test cases (11, 11, and 17 respectively), which is consistent with the fact that they affect the guard expression governing the main decision branch of `alt_sep_test()`: any flipping of the guard inverts the decision behavior for a large fraction of input combinations. These are the *strongest* mutants in the benchmark. Second, Tcas4 has only one failing test case (2.5%), which is below our acceptance threshold for a useful SFL benchmark mutant - a single failing test provides very little signal for any spectrum-based ranking method. We therefore exclude Tcas4 from the subsequent SFL experiments. Third, the remaining eleven mutants exhibit failure rates between 5% and 42.5%, providing a healthy spread of difficulty levels for the SFL evaluation.

4.4.2 Final Benchmark

After applying the kill-ability filter, the final benchmark used in the extended experiments consists of the eleven variants: `Tcas`, `Tcas2`, `Tcas3`, `Tcas5`, `Tcas6`, `Tcas7`, `Tcas8`, `Tcas9`, `Tcas10`, `Tcas11`, `Tcas12`.

4.5 Test Suite Design and Scale

For each of the eleven TCAS variants we maintain a parameterized JUnit test class that exercises the five observable outputs (`alim`, `inhibit_biased_climb`, `non_crossing_biased_climb`, `non_crossing_biased_descend`, and `return_value`) over eight chosen input combinations. With five assertions per input combination this yields *40 test cases per variant*. Across the eleven-mutant benchmark, this corresponds to:

- $11 \times 40 = 440$ instrumented test executions per experimental round;
- $440 \times 3 \times 2 = 2{,}640$ ranking computations for three SFL coefficients in two configurations (sliced/not sliced) per experimental round and 5,280 across the two extended experiments EE1 and EE2.

5 Rounds 2 and 3: Extended Experiments

We performed two independent rounds of extended experiments on the eleven selected TCAS variants. The two rounds differ in how the new variants are integrated into the JSR test infrastructure: in EE1 we isolate them in a dedicated project to obtain a compact, TCAS-only spectrum matrix, while in EE2 we integrate them alongside the preexisting programs of the original `smallProject`, replicating the matrix structure used in [11]. Both rounds compute the rank and wasted effort of the faulty statement for the three SFL coefficients in both the plain-SFL and the hybrid (sliced) configurations.

All artifacts required to reproduce the extended experiments (EE1 and EE2) - the expanded TCAS benchmark, the analysis scripts, and the generated SFL matrices - are publicly available in a replication package [1].

5.1 First Extended Experiment (EE1)

5.1.1 Motivation and Setup

The original `smallProject` accumulated benchmarks from all experiments conducted in the JSR project since 2021, with the consequence that the resulting SFL coverage matrices are very large. In particular, the plain SFL coverage matrix exported by JSR for the original `smallProject` has dimensions $557 \times 1{,}752$ (statements $\times$ test cases). Such a large matrix is unnecessarily expensive to process when the goal is to evaluate only the eleven TCAS variants, and it mixes signal from unrelated programs into the SFL summary tables.

To address this, we created a dedicated project, `smallProject2026`, containing only the eleven selected TCAS variants and their corresponding JUnit test classes (the `Test` companion of each TCAS variant). The SFL coverage matrices generated by JSR for this project have dimensions 451×716, a reduction of approximately 35% in test-case rows and 60% in statement columns relative to the original project, while preserving full coverage of the eleven TCAS variants.

5.1.2 Build and Execution Procedure

The experiment was carried out in four steps:

1. **JSR rebuild against `smallProject2026`.** The JSR core was rebuilt against the new project so that the IntelliJ plugin would point to the new directory layout.

[1] `https://doi.org/10.5281/zenodo.21431782`

2. **Test Suite Reduction (TSR) through the IntelliJ plugin.** We launched the JSR IntelliJ plugin and started a TSR run with the following configuration: *Coverage Metric: checked, Reduction Algorithm: Greedy HGS, Keep 0 coverage, deactivate redundant test cases*. The plugin emits an `ERROR` log entry indicating that it fails to create the `coverageMatrix_checked_coverage` directory under the CLI output tree even when no SFL matrix was explicitly requested. The TSR itself completes successfully; we discuss this anomaly in Section 7.
3. **SFL run via CLI.** We invoked the SFL command of `JSR-CLI-1.0-SNAPSHOT.jar` with the source, classes, and test directories of `smallProject2026`, the Slicer4J location, the output directory `JSR-CLI/build/jsr/cliTest03`, and the package filter `at.tugraz`. This run completed without errors.
4. **Post-processing.** We modified the original `SFL.py` script to operate only on the eleven TCAS variants and to emit the LaTeX tables in the required format. The modified script (`SFL_new_v2.py`) produces the per-coefficient rank and WE tables, the summary Hit@k tables, and the hybrid-vs-plain comparison tables.

5.1.3 EE1 Results

Tables 4-6 present the EE1 results. Table 4 gives the aggregate Hit@1, Hit@5, and average WE across the eleven variants for both configurations. Table 5 reports per-variant ranks and WE for plain SFL. Table 6 reports per-variant ranks and WE for the hybrid sliced approach, with ↑ indicating improvement and ↓ indicating degradation relative to the corresponding plain-SFL entry.

Table 4 EE1: Aggregate summary of findings for all eleven TCAS variants. WE stands for wasted effort.

	Ochiai			**Tarantula**			**Sarhan-Beszédes**		
	Hit@1	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**
SFL	0.45455	1.0	12.77273	0.27273	1.0	17.68182	0.45455	1.0	12.22727
Hybrid	0.09091	0.72727	18.0	0.09091	0.81818	20.0	0.09091	0.90909	17.63636

Table 5 EE1: Per-variant rank and wasted effort for plain SFL (not sliced).

Variant	Ochiai		Tarantula		Sarhan-Beszédes	
	Rank	WE	Rank	WE	Rank	WE
Tcas	4	16.5	4	16.5	2	11.5
Tcas2	2	2.5	2	2.5	2	2.5
Tcas3	1	0.0	1	0.0	1	0.0
Tcas5	1	8.5	3	17.5	1	8.5
Tcas6	4	14.0	4	15.0	3	13.0
Tcas7	1	22.5	1	22.5	1	22.5
Tcas8	1	22.5	1	22.5	1	22.5
Tcas9	1	22.5	2	48.5	1	22.5
Tcas10	2	9.5	4	17.5	2	9.5
Tcas11	2	9.5	4	17.5	2	9.5
Tcas12	2	12.5	4	14.5	2	12.5

Table 6 EE1: Per-variant rank and wasted effort for the hybrid sliced approach. ↑ marks an improvement, ↓ a degradation, both with respect to Table 5.

Variant	Ochiai		Tarantula		Sarhan-Beszédes	
	Rank	WE	Rank	WE	Rank	WE
Tcas	4	4.0↑	3↑	4.0↑	4↓	4.0↑
Tcas2	9↓	43.0↓	10↓	43.0↓	4↓	43.0↓
Tcas3	1	0.0	1	0.0	1	0.0
Tcas5	2↓	10.5↓	2↑	13.5↑	3↓	11.5↓
Tcas6	6↓	15.0↓	7↓	21.0↓	4↓	10.0↑
Tcas7	3↓	34.0↓	3↓	34.0↓	3↓	34.0↓
Tcas8	3↓	34.0↓	3↓	34.0↓	3↓	34.0↓
Tcas9	8↓	47.5↓	5↓	47.5↑	8↓	47.5↓
Tcas10	2	1.5↑	2↑	1.5↑	2	1.5↑
Tcas11	2	4.5↑	4	15.5↑	2	4.5↑
Tcas12	4↓	4.0↑	2↑	6.0↑	4↓	4.0↑

5.1.4 EE1 Observations

For the eleven-variant TCAS benchmark, plain SFL achieves a Hit@1 of 0.45455 with Ochiai and Sarhan-Beszédes, while the hybrid approach collapses to a Hit@1 of only 0.09091 across all three coefficients. The average WE increases from roughly 12.8 (Ochiai) and 12.2 (Sarhan-Beszédes) for plain SFL to 18.0 and 17.6 respectively for the sliced configuration. At the per-variant level the picture is more nuanced: slicing yields a clear WE improvement for Tcas, Tcas10, Tcas11, and Tcas12, no change for Tcas3, and a substantial degradation for Tcas2 and the three strong ROR mutants Tcas7, Tcas8, and Tcas9. We return to the structural cause of these degradations in Section 6.

5.2 Second Extended Experiment (EE2)

5.2.1 Setup and Execution

For EE2 we integrate the eleven TCAS variants *directly into the original* `smallProject`. Thus, the resulting project replicates the matrix structure used in the original study while incorporating the new variants.

The plugin-driven TSR run produces the same SFL matrix path anomaly as in EE1, with the `coverageMatrix_checked_coverage` directory failing to materialize. We then ran the SFL CLI command, which completed cleanly and exported the SFL outcome matrices to the corresponding `outcomeMatrix.csv` files. Finally, we updated `SFL_new.py` to use the single set of paths corresponding to the unified `smallProject` layout and re-ran the analysis.

5.2.2 EE2 Results

Tables 7–9 report the EE2 results following the same structure as EE1.

Table 7 EE2: Aggregate summary of findings for all eleven TCAS variants. WE stands for wasted effort.

	Ochiai			**Tarantula**			**Sarhan-Beszédes**		
	Hit@1	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**
SFL	0.45455	1.0	12.77273	0.27273	1.0	17.68182	0.45455	1.0	12.22727
Hybrid	0.18182	0.72727	18.27273	0.18182	0.81818	20.18182	0.18182	0.90909	17.81818

Table 8 EE2: Per-variant rank and wasted effort for plain SFL (not sliced).

Variant	Ochiai		Tarantula		Sarhan-Beszédes	
	Rank	WE	Rank	WE	Rank	WE
Tcas	4	16.5	4	16.5	2	11.5
Tcas2	2	2.5	2	2.5	2	2.5
Tcas3	1	0.0	1	0.0	1	0.0
Tcas5	1	8.5	3	17.5	1	8.5
Tcas6	4	14.0	4	15.0	3	13.0
Tcas7	1	22.5	1	22.5	1	22.5
Tcas8	1	22.5	1	22.5	1	22.5
Tcas9	1	22.5	2	48.5	1	22.5
Tcas10	2	9.5	4	17.5	2	9.5
Tcas11	2	9.5	4	17.5	2	9.5
Tcas12	2	12.5	4	14.5	2	12.5

Table 9 EE2: Per-variant rank and wasted effort for the hybrid sliced approach. ↑ marks an improvement, ↓ a degradation, both with respect to Table 8.

Variant	Ochiai		Tarantula		Sarhan-Beszédes	
	Rank	WE	Rank	WE	Rank	WE
Tcas	4	4.0↑	3↑	4.0↑	4↓	4.0↑
Tcas2	9↓	43.0↓	10↓	43.0↓	4↓	43.0↓
Tcas3	1	0.5↓	1	0.5↓	1	0.5↓
Tcas5	2↓	10.5↓	2↑	13.5↑	3↓	11.5↓
Tcas6	6↓	16.0↓	8↓	21.0↓	4↓	10.0↑
Tcas7	3↓	34.0↓	3↓	34.0↓	3↓	34.0↓
Tcas8	3↓	34.0↓	3↓	34.0↓	3↓	34.0↓
Tcas9	8↓	47.5↓	5↓	47.5↑	8↓	47.5↓
Tcas10	1↑	1.0↑	1↑	1.0↑	1↑	1.0↑
Tcas11	2	6.5↑	4	17.5	2	6.5↑
Tcas12	4↓	4.0↑	2↑	6.0↑	4↓	4.0↑

5.2.3 EE2 Observations

The plain-SFL columns of Table 8 coincide exactly with those of EE1 (Table 5), which is expected because both experiments use the same test suite and the same fault locations. The hybrid configuration, however, behaves slightly differently between EE1 and EE2: in EE2 the sliced Hit@1 rises from 0.09091 to 0.18182 across all three coefficients. The increase is driven by Tcas10, which moves from rank 2 in EE1 to rank 1 in EE2.

6 Discussion

6.1 Plain-SFL Results are Stable across Experiments

The plain-SFL results (Tables 5 and 8) are identical between EE1 and EE2, which is expected since both experiments use the same eleven TCAS variants and the same parameterized JUnit test suite. Tcas3 achieves perfect localization (rank 1, WE 0.0) with all three coefficients, while Tcas9 stands out with an exceptionally high Tarantula WE of 48.5, reflecting the difficulty of localizing a fault whose mutation is on the guard of the main resolution-advisory block.

6.2 Slicing Degrades Results for the Majority of Variants

Contrary to the expectation raised by Finding 3 of [11] - that multiple output variables lead to WE improvements with slicing - the hybrid approach *worsens* the average WE for the eleven-variant TCAS benchmark in both EE1 and EE2. Only Tcas10, Tcas11, and Tcas12 show consistent WE improvements, while Tcas2, Tcas7, Tcas8, and Tcas9 show severe degradations of up to +34 in WE (Tcas7, Tcas8) and +47.5 in WE (Tcas9). The expanded benchmark thus reveals that the favorable picture obtained in the original study was not representative of the broader behavior of the hybrid approach.

6.3 Bug Location Explains the Slicer Failures

The most problematic variants share a common structural characteristic: their faults affect the computation of the Boolean variable `enabled` (Tcas7, Tcas8, Tcas9) or of a local control-flow variable (Tcas2). When `enabled` evaluates to `false` as a consequence of the mutation, the entire conditional block in `alt_sep_test()` is never entered, so the dynamic slice of the output variable `return_value` never reaches the faulty line. The slicer therefore *excludes the buggy statement* from the slice, and SFL subsequently assigns it a low suspiciousness score, leading to a high rank and a large WE. This generalizes the slicer-related concern raised in [11] with concrete structural evidence: dynamic slicing is fundamentally unable to reach faults that suppress the execution of the very code paths that would expose them.

6.4 EE1 vs. EE2 Comparison

The single notable difference between EE1 and EE2 is the sliced Hit@1, which improves from 0.09091 (EE1) to 0.18182 (EE2) for all three coefficients. This improvement is essentially attributable to Tcas10, which achieves rank 1 after slicing in EE2 but only rank 2 in EE1. We think that the larger coverage matrix in EE2 (which includes all original programs, including the one-output variants) provides the slicer with a richer execution context for computing slices of the five output variables, leading to slightly different slice boundaries for this particular variant.

6.5 Computational Overhead

We did not record execution times during the experiments because the objective of this study was to evaluate fault localization effectiveness rather than runtime performance. Nevertheless, the computational overhead of the hybrid approach is substantially higher than that of plain SFL, since dynamic slices must be computed for every output variable and every test execution. While this overhead was acceptable for the TCAS benchmark used in our experiments, it may become a practical concern for larger programs with extensive test suites and numerous output variables. A systematic empirical evaluation of runtime overhead therefore represents an important direction for future work.

6.6 Summary Findings

We summarize the extended evaluation in five findings that refine the conclusions of [11].

Finding 1: The slicer has a decisive impact on the hybrid approach, consistent with Finding 1 of [11]. In particular, for ROR mutations on the `enabled` guard of `alt_sep_test()`, dynamic slicing is fundamentally unable to reach the faulty line, rendering the hybrid approach ineffective regardless of the number of output variables.

Finding 2: Furthermore, the effectiveness of the hybrid approach is constrained by the limitations of the underlying slicing framework. As also stated in [11], Slicer4J uses Jimple [13], a three-address intermediate representation, as the internal representation of the program. This transformation may remove important parts of the program, preventing the slicer from always returning a correct output. Notably, dynamic slicers do not always return a correct slice, which motivated extensions such as relevant slicing [14], where a slice must contain the faulty statement.

A detailed analysis of the generated execution traces and dynamic slices was performed in the original study [11] for the two TCAS variants. For the extended evaluation presented in this paper, a similar trace- and slice-level analysis is a natural next step. Such an analysis would help determine whether the observed behaviour is an inherent consequence of dynamic slicing or is influenced by implementation-specific aspects of Slicer4J.

Finding 3: Not all mutations with multiple outputs benefit from slicing. The benefit is limited to mutations where the faulty statement lies on a data-dependency path that is actually traversed by failing test cases. This qualifies Finding 3 of [11]: having multiple outputs is a necessary but not sufficient condition for the hybrid approach to improve over plain SFL.

Finding 4: We observe that the magnitude of degradation when slicing fails (up to +43.0 WE) tends to exceed the magnitude of improvement when it succeeds (at most ∼8 WE), suggesting that identifying in advance the fault types for which slicing is beneficial is an important direction for making the hybrid approach practically robust. Can a developer know beforehand whether slicing is a good idea? Our analysis in Section 6.3 suggests that structural program characteristics may provide useful indicators. In particular, programs in which execution is dominated by a small number of highly influential control predicates may be more susceptible to the path-suppression effect observed in this study, whereas programs with richer data dependencies or more distributed control flow may benefit more consistently from slicing. Identifying predictive metrics that characterize when dynamic slicing is likely to improve or degrade fault localization represents an interesting direction for future research.

Finding 5: Based on our analysis, we hypothesize that an additional condition is that the faulty computation must remain on an executed data-dependency path in the failing execution. In such cases, dynamic slicing can exploit the output-specific data dependencies to eliminate statements that are irrelevant to the observed failure, thereby improving the discrimination of spectrum-based fault localization. Conversely, if the fault affects a control predicate whose evaluation determines whether the faulty computation is executed at all, the execution path containing the fault may be suppressed. As a result, the faulty statement is excluded from the dynamic slice, and the combined approach may perform worse than plain SFL. Thus, multiple outputs appear to be a necessary prerequisite, whereas the preservation of the faulty computation on an executed data-dependency path may be an additional condition for the hybrid approach to be beneficial. This hypothesis, however, requires validation on a broader range of benchmark programs and fault types.

7 Threats to Validity

We acknowledge the following threats to the validity of our experimental evaluation.

7.1 Tooling Anomalies

During both EE1 and EE2, we observed that the JSR IntelliJ plugin produces an `ERROR` log entry of the form `SFLMatrixCsvExporter.writeFiles() - Error while creating directory .../coverageMatrix_checked_coverage` even when the SFL export is not explicitly requested. The TSR phase nevertheless completes successfully and the subsequent CLI-based SFL run produces all expected matrices. We also observed that, regardless of the `-o` output-directory flag passed to the CLI, certain intermediate matrices are written to `cliTest01/coverageMatrix_line_coverage/outcomeMatrix.csv`. These anomalies suggest that the output-path configuration in JSR's matrix exporter is partially hard-coded and not fully override-able through the public CLI options. While neither anomaly affected our final numerical results (which are derived from the explicitly inspected matrices), they constitute a documented limitation of the current JSR release.

7.2 Benchmark Generality

Although we have increased the TCAS variant count from 2 to 11 and covered six mutation operator families, all variants share the same underlying program. Findings should therefore be interpreted as specific to the TCAS benchmark and not generalized to arbitrary Java programs without further empirical evidence.

It is also worth noting that TCAS's structural characteristics may influence the observed behaviour of the combined dynamic slicing and spectrum-based fault localization approach. In particular, TCAS relies heavily on interconnected control predicates, such that mutations affecting them can suppress entire execution paths and consequently reduce the discriminatory power of dynamic slicing, as discussed in 6. Whether this is specific to TCAS or also occurs in other software systems remains an open question. Programs with richer data dependencies, where incorrect outputs propagate through arithmetic computations rather than being dominated by a single predicate, may be less susceptible to this suppression effect, since a mutation is then more likely to alter a value along a path that still executes rather than eliminate the path entirely. Similarly, programs whose behaviour is distributed across several independent decisions, instead of a small number of dominant predicates, may be less prone to the same all-or-nothing failure. Evaluating the proposed approach on such structurally diverse benchmark suites remains a key objective for future research.

7.3 Slicer Dependence

The hybrid approach depends on the choice of dynamic slicer (Slicer4J [3] in our case). A slicer with different internal heuristics could in principle reach different conclusions about which statements are excluded from a slice when the faulty path is suppressed. Re-running our experiments with alternative slicers is left to future work.

7.4 Single-fault Assumption

Our evaluation follows the single-fault assumption commonly adopted in spectrum-based fault localization research and in the original study we replicate in the first place. This enables controlled experimentation and direct comparison with prior work. However, real-world software systems may contain multiple faults, which could interact in ways that affect both spectrum-based suspiciousness rankings and the computed dynamic slices. Evaluating the proposed approach in multi-fault scenarios is therefore an important direction for future work.

7.5 Raw Wasted Effort

Since all evaluated subjects are variants of the same TCAS benchmark, the number of executable statements is constant across experiments. Therefore, raw Wasted Effort (WE) values are directly comparable in our setting. For studies involving programs of substantially different sizes, a normalized WE metric would facilitate cross-benchmark comparisons.

8 Conclusion

In this work, we present a replication and expansion of the experimental evaluation of the hybrid dynamic-slicing SFL approach of Schleich and Wotawa [11]. The contribution comprises three rounds of experiments. In the first round, we replicated the third experiment of the original study and confirmed its reproducibility. In the second and third rounds (EE1 and EE2), we extended the TCAS benchmark from two to eleven faulty variants by generating nine new mutants across six mutation operator families, and evaluated the three SFL coefficients in both the plain-SFL and hybrid sliced configurations.

The expanded benchmark of 11 retained variants $\times$ 40 parameterized JUnit test cases $\times$ 5 output assertions per test corresponds to 2,200 individual output-value checks per SFL coefficient per configuration, or 5,280 ranking computations across the two extended experiments and three coefficients in both configurations. This represents a substantial extension of the empirical evidence base for the hybrid approach.

Our central finding is that improvements from dynamic slicing are highly dependent on the structural relationship between the fault location and the dynamic slices computed for the output variables. The Slicer4J tool fails to include the faulty statement in its output whenever the fault prevents the execution of the very code path that would reveal it - a fundamental limitation of dynamic slicing that manifests precisely for the strongest mutations (those with the highest number of failing test cases). This provides new, concrete structural evidence for the slicer-related limitations identified in the original study and suggests that fault localization approaches combining slicing and SFL should prioritize addressing slicer accuracy on control-flow-dominated faults.

Future work should investigate whether static slicing or hybrid static/dynamic slicing can overcome this limitation, whether the findings generalize to larger programs and more diverse fault types, and whether alternative relevant slicers exhibit different behavior on the same benchmark.

References

1 Robin Abraham and Martin Erwig. Mutation operators for spreadsheets. *IEEE Transactions on Software Engineering*, 35(1):94–108, 2009. doi:10.1109/TSE.2008.73.

2 Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC PART)*, pages 89–98, 2006. doi:10.1109/TAIC.PART.2006.256442.

3 Khaled Ahmed, Mieszko Lis, and Julia Rubin. Slicer4J: A dynamic slicer for Java. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

4 Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. SIR: Software infrastructure repository. https://sir.csc.ncsu.edu, 2005.

5 EclEmma Team. JaCoCo: Java Code Coverage Library. https://www.eclemma.org/jacoco/, 2025. Last accessed: April 2026.

6 James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005. doi:10.1145/1101908.1101949.

7 Roxane Koitz-Hristov, Thomas Sterner, Lukas Stracke, and Franz Wotawa. On the suitability of checked coverage and genetic parameter tuning in test suite reduction. *Journal of Software: Evolution and Process*, 36(8):e2656, 2024. doi:10.1002/smr.2656.

8 Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988. doi:10.1016/0020-0190(88)90054-3.

9 Qusay Idrees Sarhan and Arpad Beszedes. Experimental evaluation of a new ranking formula for spectrum based fault localization. In *Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 276–280, 2022. doi:10.1109/SCAM55253.2022.00038.

10 Jonas Schleich. JSR: Extended Version of the Java Test Suite Reduction Framework. https://github.com/SchleichJonas/JSR, 2025. Fork of the original JSR framework by Lukas Stracke; GitHub repository, version used in this study, accessed: 2026-05-20.

11 Jonas Schleich and Franz Wotawa. Combining dynamic slicing and spectrum-based fault localization – A first experimental evaluation. In *Proceedings of the 36th International Conference on Principles of Diagnosis and Resilient Systems (DX 2025)*, OASIcs, pages 3:1–3:18, 2025. doi:10.4230/OASIcs.DX.2025.3.

12 Lukas Stracke. JSR: Java Test Suite Reduction Framework. https://github.com/Lms24/JSR, 2024. GitHub repository, accessed: 2026-05-20.

13 Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical Report 1998-4, Sable Research Group, McGill University, July 1998. URL: https://api.semanticscholar.org/CorpusID:10529361.

14 Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault localization. In *Proceedings of the Sixth International Symposium on Automated and Analysis-Driven Debugging (AADEBUG)*, pages 33–42, 2005. doi:10.1145/1085130.1085135.