Tailoring Programs for Static Analysis via Program Transformation

Anonymous Author(s)

ABSTRACT

Static analysis is a proven technique for ensuring software quality automatically catching bugs early during development. However, analysis tooling must approximate, both theoretically and in the interest of practicality. False positives are a pervading manifestation of such approximations—tool configuration and customization is therefore crucial for usability and directing analysis behavior. To suppress false positives, developers readily disable bug checks or insert comments that suppress spurious bug reports. Existing work shows that these mechanisms fall short of developer needs and present a significant pain point for using or adopting analyses. We draw on the insight that an analysis user always has one notable ability to influence analysis behavior regardless of analysis options and implementation: modifying their program. Indeed, developers find that they can effectively suppress false positives by changing their code slightly; this exercise is however manual, ad-hoc, and can introduce awkward or redundant code. We present a new technique for automated, generic, and temporary code changes that tailor to suppress spurious analysis errors. We adopt a rule-based approach where simple, declarative templates describe general syntactic changes for code patterns that are known to be problematic for the analyzer. Our technique promotes program transformation as a general primitive for improving the fidelity of analysis reports (we treat any given analyzer as a black box). Our evaluation is the first systematic study to broadly demonstrate the applicability and benefits of this technique and perspective: we evaluate using five different static analyzers supporting three different languages (C, Java, and PHP) on large, real world programs (>800KLOC). We show that our approach is effective in sidestepping long-standing and complex issues in analysis implementations.

ACM Reference Format:
Anonymous Author(s). 2020. Tailoring Programs for Static Analysis via Program Transformation. In *Proceedings of The 42nd International Conference on Software Engineering (ICSE 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Writing and maintaining high-quality, bug-free software remains a largely manual and expensive process. Static analysis has proven indispensable in software quality assurance for automatically catching bugs early in the development process [31]. A number of open challenges underlie successful adoption and integration of static analyses in practice. Analyses must approximate, both theoretically [23, 25] and in the interest of practicality [10, 12]. Developers are sensitive to whether tools integrate seamlessly into their existing workflows [22]; at minimum analyses must be fast, surface few false positives, and provide mechanisms to suppress warnings [11]. The problem is that the choices that tools make toward these ends are broadly generic, and the divergence between tool assumptions and program reality (i.e., language features or quality concerns) can lead to unhelpful, overwhelming, or incorrect output [21].

Tool configuration and customization is crucial for usability and directing analysis behavior: the inability to easily and selectively disable analysis checks and suppress warnings presents a significant pain point for analysis users [11]. Common existing mechanisms include analysis options for turning off entire bug classes (generally too coarse [22]) or adding comment-like annotations for suppressing spurious warnings at particular lines (a predominantly manual exercise that leads to code smells and is insufficiently granular [11, 22]). It is notably the *analysis author*, not the user, who has agency over the shape of these analysis knobs: which configuration options are available and how to suppress errors. It follows that it is infeasible for analysis writers to foresee and accommodate individual user preferences or analysis corner cases through a myriad of configuration options or suppression mechanisms.

An *analysis user* always has one notable ability to influence analysis behavior and output: modifying their program. For example, developers may slightly modify existing code in a way that suppresses false positives or undesired warnings. A concrete example is the following change, which was made in rsyslog[1] to suppress a Clang Static Analyzer warning:

```
- if(strcmp(rectype, "END")) {
+ // clang static analyzer work-around
+ const char *const const_END = "END";
+ if(strcmp(rectype, const_END)) {
```

The analyzer warns that a potential out-of-bounds access occurs when comparing two strings using `strcmp`. However, the warning only surfaces when complex macro expansions take place (in this case, macro expansion activates for `strcmp` because it is passed a string literal). One contributor notes that under normal circumstances these warnings are suppressed for macros, but can surface if macro preprocessing is done manually.[2] The workaround in this case extracts the string literal out of the `strcmp` call so that no macro-expansion takes place; after the change, the analyzer sees `strcmp` as a C library function and no longer emits a warning. In practice, modifying programs to avoid analyzer issues are exercised as manual, one-off changes. Changes like the one above can have the undesirable side-effect of persisting in the code solely to suppress unwanted analysis warnings. Despite these shortcomings,

ICSE 2020, 23–29 May, 2020, Seoul, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://github.com/rsyslog/rsyslog/commit/ea7497
[2] https://bugs.llvm.org/show_bug.cgi?id=20144

modification of the analysis target (the program) does, however, allow a primitive for *analysis users* to draw on their particular domain knowledge of their code to positively influence analysis behavior. For example, a developer may recognize that a particular API call is the cause of a false positive resource leak, and modify or model the call differently to suppress a false positive.[3] Our technique in particular affords agency to *analysis users* to change and suppress analysis behavior when no recourse is available in tool support (e.g., when analysis maintainers delay fixing analysis issues, or limit tool configuration options). Our insight is that the same flavor of one-off, manual code changes like the one above can apply generally and automatically to remedy analysis shortcomings. Additionally, workaround changes need not have the undesirable trait of persisting in production code, and are applied only *temporarily* while performing analysis.

We propose a rule-based approach where simple, declarative templates describe general syntactic changes for known problematic code patterns. Undesired warnings and false positives are thus removed during analysis by rewriting code fragments. Our approach can be seen as a preprocessing step that tailors programs for analysis using lightweight syntactic changes. It operates on the basis of *temporary* suppression (a desirable trait in configuring analyses [22]) and also enables catching false positives *before* they happen by rewriting problematic patterns. Since patterns can occur across projects, codifying transformations as reusable templates amortizes developer effort for suppressing false positives.

The notion that syntactic transformations abstract semantic transformations [14] underpins our intuition that manipulating syntax can achieve desirable changes in the analysis domain and implementation. Work on semantic properties of transformations emphasize the potential for improving analysis precision [26, 28], and recent work suggests that automatic bug-fixing transformations can improve analysis results in popular real-world programs [33]. Despite anecdotal and theoretical appeal for tailoring analyses via transformation, there is currently little demonstrated applicability or benefit in practice.

A key objective of our work is to demonstrate the broad feasibility, applicability, and effectiveness of these ideas for the first time in practice. To this end we evaluate on large, real-world programs written in a variety of popular languages. A significant challenge lies in recognizing and transforming syntactically-diverse languages for such an approach to work. Recent techniques in declarative multi-language transformation help to address this challenge [34] and forms the basis for operationalizing our approach. We address analysis issues broadly by considering (a) user-reported false positives across multiple active analyzers and (b) historic user commits for suppressing analysis warnings. We develop transformations that tailor programs to address shortcomings in analysis implementation and reasoning. Our contributions are as follows:

- We operationalize the process of tailoring programs for static analysis using declarative syntax rewriting.
- We show that our approach is effective in improving existing static analyses and resolves real (including yet-unresolved) false positive issues affecting analysis users.

[3] https://github.com/facebook/infer/issues/781.

```
1  function test(): void {
2    if (($file = @fopen('file', 'wb+')) === false) {
3      return;
4    }
5
6    // analyzer complains that $file may be false
7    if (\fputcsv($file, [1,2,3]) === false) {
8      \fclose($file);
9      return;
10   }
11   ...
12   \fclose($file);
13 }
```

(a) Assigning the result of `fopen` to `$file` in Line 2 confuses the analyzer. It doesn't track the effect that `$file` is not false on Line 7, and emits a false positive warning that `$file` may be false when passed to `$putcsv`.

```
1  function test(): void {
2    $file = @fopen('file', 'wb+');
3    if ($file === false) {
4      return;
5    }
6    ...
7  }
```

(b) An analysis user proposed this workaround: extract the assignment out of the conditional. This allows the analyzer to correctly track the assignment effect and does not emit a false positive. Unfortunately this approach is hard to blanket-apply automatically and diverges from developer preferences who prefer this style for readability.

Figure 2: Variable assignment inside if-statements can cause a false positive report in `PHPStan`. A workaround is to put the assignment outside of the if conditional.

- We show that our approach is efficient: transformation typically takes one to three seconds (compared to analyses that typically take in the order of minutes)
- We present empirical results of our approach on 15 real-world projects (including large ones, >100KLOC) across three languages (PHP, Java, and C) and develop 9 transformation templates for improving the analysis output of five modern and active analyzers (`Clang`, `Infer`, `PHPStan`, `SpotBugs`, and `Codesonar`).

2 MOTIVATION

Fig. 2a illustrates a past issue in `PHPStan`, a popular PHP analyzer. A file is opened in line 2, and assigned to a handle `$file` *inside* the `if`-condition on line 2. If opening the file fails, `$file` is assigned the value `false`; the condition evaluates to true and the function immediately returns (line 3). On the other hand, if execution passes the check then `$file` is guaranteed to be valid (i.e., not `false`). The problem is that `PHPStan` would not track the effect of assignments in `if`-conditions, and reports an error saying that `$file` may be `false` when passed as an argument to `fputcsv` on line 7.

The issue for this false positive stayed open and unresolved for *over a year* on GitHub, and is cross-referenced in 13 related user-reported issues.[4] One project member responded that the issue is

[4] https://github.com/phpstan/phpstan/issues/647.

important and will be fixed in the future, but that "no one has yet figured out how to implement it without rewriting major part of the (analysis) core". The fix imposed significant effort on analysis authors which delayed a resolution for months. Although some analysis authors were in favor of code that avoided assignments in conditionals, others found the style improves readability. Multiple users noted that the false positive can be avoided by a code change that extracts the assignment out of the conditional (Fig. 2b).[5] However, one user also noted that this workaround was "a bit annoying" because it introduced redundancy. The proposed workaround also imposes a burden on the user to identify and refactor all affected instances. Our approach introduces a new way to address these tensions. The intuition is that workarounds via code changes, as in Fig. 2b, *can* generalize to cater for individual user preferences and overcome long-standing analysis issues. The high-level idea is to write simple, declarative syntactic templates that can blanket-apply automatically over an entire code base. Although the code changes *could* be persisted in the code, they need not be: our approach foremost promotes code changes as a temporary suppression mechanism with respect to a particular analysis. In our approach, a *match template* specifies a pattern that is syntactically close to the problematic pattern in Fig. 2a:

```
if ((:[v] = :[fn](:[args])) === false)
```

A *rewrite template* replaces all instances of the match template, extracting the assignment out of the `if`-condition. The rewrite template is syntactically close to the pattern suggested by the user in Fig. 2b:

```
:[v] = :[fn](:[args]);
if (:[v] === false)
```

The match template matches on the syntactic pattern where variable `v` is assigned the return value of calling a function `fn` with arguments `args` (the `:[`*identifier*`]` notation binds matching syntax to a variable *identifier*). All other syntax is matched *concretely*; whitespace in the template matches all contiguous whitespace in the source code. For illustration we use this template to match on function call syntax because the analysis particularly tracks values for modeled functions like `fopen`. The *rewrite template* references variables in the match template, which are substituted with their corresponding matched syntax.

Using the above patterns we identified and rewrote 27 instances of the `if`-assign pattern in the WordPress and PHPExcel projects and removed 82 false positives due to issue #647 in `PHPStan`. Our approach has the positive effect of removing more false positives than matches, because the issue has a cascading effect of reporting false positives along multiple execution paths (we elaborate in Section 4).

Writing these declarative patterns is comparatively easy to implementing additional analysis reasoning and sufficiently general for overcoming analysis shortcomings. The format of syntactic templates is accessible to developers; indeed templates can be written in a format that is syntactically close to user-identified and user-implemented workaround transformations (as in Fig. 2b). In our experience, complex changes to an analysis implementation can have a correspondingly easy resolution via code transformation patterns; templates can be developed in minutes for issues that take days to months to resolve in an analysis implementation (or even issues that don't have any proposed solution whatsoever).[6]

Match and rewrite templates appear simple, but express nontrivial syntactic properties that go beyond the capabilities of regular expression search-and-replace. We use recent work in multi-language syntax transformation to enable this approach and apply it broadly toward our objective.

3 APPROACH

This section explains the overall process of our program tailoring approach. We introduce background on the rewrite technique in Section 3.1. Section 3.2 explains how we integrate program transformation for improving the quality of analyzer bug reports and the principles behind our approach.

3.1 Preliminaries: Lightweight Syntax Transformation

To rewrite syntax declaratively in the fashion shown, `comby` [34], a tool for declaratively rewriting syntax with templates. We give a brief overview of template syntax and matching behavior:

- The `:[hole]` syntax binds matched source code to an identifier *hole*. Holes match all characters (including whitespace) lazily up to its suffix (analogous to `.*?` in regex), but only *within* its level of balanced delimiters. For example, `{:[hole]}` will match all characters inside balanced braces. Parentheses and square brackets are also treated as balanced delimiters.
- `:[[hole]]` matches only alphanumeric characters in source code, analogous to `\w` in regex.
- Using the same identifier *hole* multiple times in a match template adds the constraint that matched values be equal.
- All non-whitespace characters are matched verbatim.
- Any contiguous whitespace (spaces, newlines, tabs) in a match template matches any contiguous whitespace in the source code. Match templates are thus sensitive to the presence of whitespace, but *not* the exact layout (the number of spaces do not need to correspond exactly between match template and source code).
- Matching is insensitive to comments in the source code; comments are treated like whitespace when matching non-hole syntax in the match template.

We additionally use rules to refine match and rewrite conditions. Rules place constraints on matched syntax; we explain rules as needed in the rest of the paper. The match template, rewrite template, and rule comprise the full input for a single transformation. Each part is passed on the command-line.

3.2 Tailoring Programs for Analysis

This section explains our approach to tailoring programs for analysis. Fig. 3 illustrates the main phases in our approach; we use it to characterize the process in detail.

[5] https://github.com/phpstan/phpstan/issues/1739

[6] See, e.g., https://github.com/spotbugs/spotbugs/issues/493

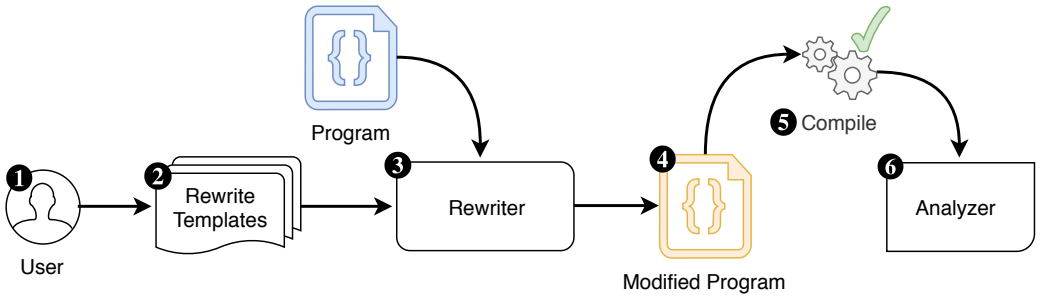

Figure 3: Overview of the program tailoring process.

Who writes templates? Our approach starts with a manual step where users ❶ write transformation templates. A key principle of our approach is to allow particularly *analysis users* to influence static analyses via program transformation. The simplicity of declarative templates make syntax transformation accessible so that user observations and workarounds (as in Fig. 2b) can be straightforwardly implemented. This primitive is not exclusive to analysis users, but also accessible to analysis writers and external contributors. For example, analysis writers can create and distribute transformation templates for issues on an interim basis (such as for Fig. 2a). Users can then apply these templates as a temporary workaround before analyzing their projects. In this paper, we (the authors) develop and evaluate transformations as external contributors—we treat analyses as a black box (as a user would) and develop transformations for extant issues in popular analyzers (as an analysis writer might).

What properties do templates have? Writing rewrite templates ❷ is a one-off exercise per transformation. Templates are generally short (we develop templates that are between 3 and 15 lines long). Our results show that there is typically a one-to-one correspondence of rewrite template to analysis issue, where an analysis issue can entail a general problem in analysis implementation, code generation, or function modeling. Formulating rewrite templates requires competitively low effort compared to implementing complex changes analysis-side.

Templates are customizable. For example, we can refine the template `if ((:[v] = :[fn](:[args])) === false)` to match on a particular concrete function, such as `fopen`, instead of `:[fn]`. Rudimentary mechanisms (e.g., C macros, comments, or assertions [17], and bug auto-patching [4]) have been used and recommended for suppressing false positives in existing analyzers [3, 6]. These mechanisms suffer from relying on language-specific features, are brittle and coarse, and built into the tool or hardcoded in the program. Our template-based mechanism is broadly accessible (it is easy to write templates), customizable, and generic across languages for manipulating syntax. It operates independent of language toolchains and does not presume the availability of language-specific features (e.g., macros) and does not prescribe any configuration for external analysis tools or compilers. Templates currently express purely syntactic properties (e.g., we do not draw on type information to inform manipulation). However, future work may incorporate richer static information.

A set of templates form a catalog of transformations that can be reused across projects using analyses. A catalog is the starting point for an automated pipeline, driving the remaining phases in Fig. 3. In this paper we present a catalog of human-written templates that directly targets known analysis issues.

What are the principles behind automated code changes tailored to analysis? The rewriter tool ❸ takes as input the set of rewrite templates and a program. It rewrites matched syntax in the original program to produce a modified program that will be analyzed. The modified program is intended to be a *temporary* representation of the original program that is discarded after analysis. In practice users can identify transformational workarounds to issues (as in Section 2), but do not want to persists those changes in their code.[7] Our solution provides the ability where users only temporarily change their code under analysis. We implement temporary program modification by running the rewriter on a version controlled project (we use `git`). After running the analysis and capturing the bug reports, the project is reverted to its original state. Syntactic rewriting is fast (less than 4 seconds to process an 800KLOC project) and introduces negligible overhead compared to analysis runtime.

In this paper we consider transformations primarily as a targeted suppression mechanism for analysis false positives. However, we observe that transformations can also be tailored to surface additional bugs (we elaborate in Section 4).

The rewriter can be seen as a preprocessor for analyses. One principle of running our approach *prelude* to analysis is that we can dually use templates to search for and detect (but not necessarily rewrite) problematic syntactic patterns. This mechanism provides an early smoke signal that may trip particular analyzers *even before the analysis runs*. We notably reuse our templates to detect false positive-inducing patterns across projects in our evaluation. Large scale efforts can similarly detect the extent of possibly affected projects when adopting a new analyzer; analysis writers may use it to prioritize analysis fixes. These capabilities are notable in contrasting our approach to existing mechanisms: syntactic templates are

[7] For example, one user user experience identifies that a certain change removes a false positive report, but they do not want to permanently change their code https://github.com/Microsoft/CodeContracts/issues/255

valuable in explicitly codifying sensitivities of analysis behavior as it relates to program structure, whereas suppression-by-comments and configuration options provide an escape hatch in anticipation of problematic analysis interactions where program structure is readily ignored.

What are the conditions for analyzing a modified program?

Automated program transformation is a powerful primitive, and applying incorrect templates can produce malformed programs. When developing templates, it is helpful to impose validation criteria for running the analysis on the modified program. Analyses generally accept only well-formed programs (in the respective language), and typically rely on building artifacts or instrumenting compiler output to perform analysis. Language allowing, we impose a validation step that all modified programs must compile ❺ when performing analysis, which provides additional assurance that the analysis will not terminate early due to malformed programs. For PHP, we rely on a valid parse of target files as the validation step. Applying transformations may violate style checkers (e.g., linters) integrated into a build manifest, or cause spurious compiler warnings (e.g., unused variables), but still allow the program to compile successfully. We allow such violations or warnings; in our results these do not directly affect the fidelity of the analysis with respect to the issue targeted by transformation. Another possibility of interference is that transformations addressing one bug class may lead to additional reports for a different bug class. This undesirable behavior depends on the analysis checks, its configuration, and template formulation. We qualify such cases in our experiments (Section 4).

In the final step we capture analyzer output for the modified program ❻. The goal of our approach is that this output provides a higher quality bug report than that obtained from running the analyzer on the original program.

4 EVALUATION

This section describes our results tailoring programs for analysis. Our evaluation emphasizes real-world applicability on large and popular programs for modern analyzers. The goal of our evaluation is to show that programs *can* be tailored generically and declaratively to improve analysis output. We focus on breadth of applicability across languages and analyzers for real-world issues. Thus, our research questions are:

RQ. 1 **Can tailoring programs improve the fidelity of static analysis reports?** Specifically: Does the approach remove false positive reports without otherwise adversely affecting the analysis results?

RQ. 2 **Does the program tailoring approach generalize?** Specifically, we evaluate generality with respect to multiple languages and analyzers, and pattern reuse across projects.

Section 4.1 describes our experimental setup. Section 4.2 describes our results applying 9 rewrite templates to 15 projects across five analyzers.

4.1 Experimental Setup

We consider five popular analyzers: `PHPStan` [7] for PHP, `SpotBugs` [8] for Java, `Clang Static Analyzer` [1] and `CodeSonar` [2] for C, and `Infer` [5] for both Java and C. All analyzers are mature, actively used, and incorporate state-of-the-art techniques. All analyzers are open source, except for `CodeSonar` which is a commercial analyzer.[8] For each analyzer we were interested in current or long-standing issues that cause spurious warnings or false positives; particularly issues that could not be easily addressed by analysis configuration or suppression mechanisms:

(1) We searched the `PHPStan`, `SpotBugs`, and `Infer` issue trackers on GitHub for reports or comments containing the words "false positive".
(2) We found related issues for the `Clang Static Analyzer` (which does not have a GitHub issue tracker), by searching for GitHub commits containing the words "clang static analyzer false positive".
(3) `CodeSonar` does not have a public issue tracker, nor did we find public commits referencing false positives. We manually inspected warnings for false positives.

The above methods influenced project selection. For (1) we identified problematic syntax patterns and user-affected projects from user reports. For (2) we identified committed code changes in an existing project and used this to develop a generic template. For (3) we selected popular C projects as representative real world projects (since we did not have sources indicating false positives in CodeSonar a priori).

Templates developed from the initial set of issues were reused to search for additional projects containing potentially false positive-inducing patterns. We searched over the top 100 most popular[9] projects for each language. However, to apply our approach generally, we require that a project (a) compiles (or parses) successfully and (b) is configured for analysis. Many of the projects identified by large-scale search presented significant manual burden to compile (e.g., due to various dependencies and platform-specific requirements like like Android, iOS, or Visual Studio) and consequently configure for analysis. We note that this burden is amplified for external users (such as ourselves) who have limited access to various platforms and who are unfamiliar with specific build configurations. We generally expect the burden to be less of an impediment for using our approach among project maintainers, who are familiar with the complexities of their own projects.

In aggregate, we evaluate our approach on 15 projects. Our selection represents a convenience sample of real world issues to substantiate our claims about tailoring programs to overcome analyzer limitations. The selection includes issues that affect real developers of large, popular codebases. We show that our approach is efficient and scales to these concerns, and that it presents reuse potential across projects.

We ran our large-scale search experiments on an Ubuntu 16.04 LTS server, with 20 Xeon E5-2699 CPU cores and 20GB of RAM. We evaluated analysis improvement on this same hardware for large projects (>100KLOC) and the `CodeSonar` analyzer. All other analysis improvement experiments were run on an Ubuntu 16.04 VM with two 2.2 GHz i7 CPU cores and 4GB RAM.

[8] We used `CodeSonar` under a free academic license.
[9] Ranked by the number of user favorites (GitHub stars).

Analyzer	Lang	Pattern	Project	KLOC	Time		Reports			#R	Ref
					Anlyz	Rewr	Bef	Cls	Δ FP		
PHPStan	PHP	if-assign-resources	WordPress	5.7	2.5s	0.3s	112	44	-44	16	647
			PHPExcel	5.1	0.7s	0.1s	113	38	-38	11	
		substr-model	dompdf	3.9	0.5s	0.2s	34	1	-1	1	1215
			matomo	0.3	0.2s	0.1s	21	1	-1	1	
			PHP_CodeSniffer	1.6	1.1s	0.1s	123	3	-3	3	
			neonizer	0.1	0.1s	0.1s	1	1	-1	1	
Infer	C	free-model	OpenSSL	402.9	17m29s	1.6s	462	22	-6	3,735	120
	Java	create-socket	Drift	50.7	2m47s	1.0s	62	1	-1	1	781
			Presto	813.0	39m20s	3.3s	822	106	-1	1	
		wrapped-resources	hazelcast-jet	103.8	5m24s	1.3s	29	7	-2	16	999
SpotBugs	Java	null-on-resources	hazelcast-jet	103.8	1m48s	2.0s	1	1	-1	45	756
			Santulator	11.1	44s	0.5s	2	2	-2	8	
Clang SA	C	const-strcmp	rsyslog	145.0	16m20s	2.9s	3	3	-3	10	ea74
CodeSonar	C	strncpy-null	swoole-src	96.8	6m40s	0.6s	117	1	-1	1	—
		snprintf-null	ioping	1.2	0m14s	0.2s	12	2	-2	2	

Table 1: Main results of our approach. Each row represents an application of a transformation Pattern on a project. Reports, Bef is the total analyzer warnings before transformation. Cls is subset of total analyzer warnings in the bug class that the transformation targets. (Δ FP) is the number of warnings removed by the transformation for the targeted bug class. Ref is the external GitHub issue or commit reference (cf. Table 3. Time, Anlyz is the time to run the analyzer on each project (given in thousands of lines of code, KLOC). Rewr is the time to process and rewrite each matching pattern across KLOC. We remove 107 false positives (Δ FP) in total, with a median of 2 per project.

4.2 Experimental Results

4.2.1 Overview. Table 1 shows our results for each analyzer. We develop transformations for 9 syntactic **Patterns** across PHP, Java, and C projects. The **Issues** column shows the total number warnings across all bug classes in the **Before** column (each analyzer is run with its default checks). We ran each analyzer on the entire project (warnings are thus for the entire project) except for `PHPStan` where the number of warnings is reported for a single file. `PHPStan` can operate at the file level, and using targeted transformation we demonstrate that our approach can isolate issues in individual files without incurring a project-wide analysis. The **Cls** column is the subset of all bugs that fall into the bug **Class** that the pattern targets. The Δ **FP** column is the number of false positives removed for that bug class by transformation. The **#R** column is the number of rewritten instances in the source code for each pattern. There may be more or fewer rewritten instances compared to removed false positives due to the effect of transformations on analyses (cf. pattern **free-model** and **if-assign-resources**); we elaborate in Section 4.2.5. The time to transform the program (**Time Rewr**) is negligible compared to analysis runtime (**Time Anlyz**). In aggregate these transformations remove 107 false positives (Δ **FP**), with a median of 2 per project.

4.2.2 Effects of transformations on analysis behavior. Providing a primitive for arbitrarily modifying programs means that our approach can hypothetically introduce adverse effects which do not exist in the original program. The negative possibilities are that a change either removes true positives in the original program, introduces more false positives in the modified program, or both. We evaluated whether our patterns cause such adverse effects by manually inspecting bug reports before and after applying a transformation for every analysis run. From our inspection, no true positives are removed for any project. No additional bug reports are introduced for any project, except OpenSSL and Presto. Infer nondeterministically reports different numbers of bugs for large Java projects in the case of Presto, irrespective of whether we perform a change. We confirmed that our change removes the false positive on five independent runs; nondeterminism for large numbers of bug reports (>800) make it difficult to conclude whether our change has any meaningful effect on other reports.

We observed that Infer reports an additional 6 potential null dereferences after applying the **free-model** pattern. This is interesting because the **free-model** pattern targets memory leak false positives, not null dereference reports. Interestingly, Infer analyzes and reports bugs in functions after transforming the program, whereas it previously short-circuits analysis and reporting for functions containing `free`-wrapper functions. We inspected these reports and could not obviously discern whether they were false positive or true positive reports.

In summary: Our approach removes false positives without adversely effecting the analysis in the majority of cases (13 out of 15 projects), while two projects are inconclusive. In general, program tailoring is effective because transformations perform small, local changes that affect only the reasoning of the analysis for that instance or bug class (as detailed in Section 4.2.5). It follows that changes which are closely semantics-preserving of the original program ought not make an analysis less precise, and our results affirm this intuition.

Lang	MLOC	Pattern	#M	Time
PHP	8.9	if-assign-resources	42	18s
		substr-model	6	20s
Java	16.2	create-socket	1	56s
		wrapped-resources	254	30s
		null-on-resources	4	35s
C	29.4	free-model	52,461	7m
		const-strcmp	91	48s
		strncpy-null	1	48s
		snprintf-null	13,763	50s

Table 2: Large-scale search results over the top 100 most popular PHP, Java, and C repositories on GitHub. MLOC is the millions of lines of code searched; #M is the number of matching instances across all projects. Search is fast and scales to millions of lines of code for rich patterns.

4.2.3 Amortizing human effort for codifying and detecting patterns. Pattern reuse is an especially appealing property of tailoring programs. We developed four patterns from user-reported issues in a single project[10] which we then used to detect and fix multiple false positive issues in six additional independent projects. These results show that patterns may generalize to benefit multiple projects, and imply that the cost and human effort of writing broadly applicable templates can be amortized across software stakeholders (i.e., both analysis users and analysis writers develop, distribute, and benefit from patterns). In contrast, existing mechanisms using comment suppression or command line options cannot likewise generalize, and consequently induce recurring developer effort.

We performed a large-scale search using each pattern to identify the additional projects above, and to quantify overall efficiency.[11] Table 2 shows our results running each pattern on the top 100 most popular GitHub repositories for PHP, Java, and C. In general search is fast and can identify potential false-positive inducing patterns before an analysis even runs.

4.2.4 Issue duration and resolution for analysis end users. Table 3 characterizes six open source issues that our patterns address.[12] Interestingly, issues are long-standing (unresolved for over a year, on average), and all but one remain unresolved. These issues generally reveal that analysis end users are subject to long delays and lack of support, having little recourse for resolving false positives using existing mechanisms. On the one hand, analysis writers may not have the time to support user- or project-specific needs, and may deprioritize less pressing requests (e.g., `infer/781` is an individual user request with no cross-references to other issues). On the other hand, an issue may affect many users (as shown by multiple cross-referenced issues for `phpstan/647`, `spotbugs/756`), but very complex to solve for analysis maintainers.

Our approach handles these issues via program transformation, and give agency to end users for implementing workarounds. Furthermore, user-reported errors can be significantly more severe than other analysis reports as they may break existing software workflows. For example, even a single false positive report in `SpotBugs` due to **null-on-resources** breaks the continuous integration (CI) build of cross-referenced projects, and caused users to disable the check wholesale for their projects across all versions of Java. Because of this profound effect, bug severity and build integration must be weighed into analysis configuration mechanisms. Moreover, although the relative size of false positive reduction is small for some reports in Table 1, the correspondence to real-world issues and extended impact on end users and software workflows make the false positives we handle more significant than others. Our results show that our approach can uniquely address such complexities.

[10] **if-assign-resources**, **substr-model**, **create-socket**, and **null-on-resources**

[11] Note that the patterns identified projects which failed to run under the analysis or compile, and hence not included in our final Table 1.

[12] The three remaining patterns, **rsyslog**, and those for CodeSonar, do not have open source issues.

Fixed?	Issue	Duration	Reported	# Refs
✓	phpstan/647	1 yr. 3 mos.	11-2017	16
✗	phpstan/1215	1 yr. 1 mo.	07-2018	1
✗	infer/120	4 yrs. 1 mo.	06-2018	4
✗	infer/781	1 yr. 9 mos.	10-2017	0
✗	infer/999	10 mos.	09-2018	0
✗	spotbugs/756	11 mos.	09-2018	41

Table 3: Summary of false positive issues in active analyzers that we address. The ✗ in the Fixed? colum indicates that the issue is still unresolved at the time of writing. Only one of the issues is currently resolved (✓). On average, issues stay unresolved for 1 year and 7 months (aggregated over Duration, as of the date the issue was Reported). # Refs indicates the number of additionally cross-referenced issues for a report (including, e.g., duplicate user reports or external tools affected by this issue).

Match template

```
if ((:[v] = @fopen(:[a])) == 0)
```

Rewrite template

```
:[v] = @fopen(:[a]);
if (:[v] === false)
```

Figure 4: The if-assign-resources pattern extracts a variable assignment out of an if-conditional (in this case for `fopen` calls).

4.2.5 Rewrite patterns. We now discuss analyzer issues in greater detail, explain what our transformation does to resolve the issue, and why it induces a positive change in analysis behavior.

Pattern: if-assign-resources. This pattern addresses the issue of variable assignment in `if`-conditionals (as introduced in Section 2). At the original time of writing `PHPStan` did not accurately track the effects of such assignments, and it took over a year to fix in the analysis implementation. False positives particularly manifest for cases where states of resources (like files) are opened. Because `PHPStan` does not track that the variable assigned cannot be false, it reports an error when the variable is passed to a function such as `fwrite` or `fclose`, saying "Parameter #1 of function `fclose` expects resource, `resource|false` given". The transformation pulls the assignment out of the if-conditional, which allows `PHPStan` to accurately track the effect (Fig. 4). An interesting result is that rewriting this pattern can remove many false positive reports because multiple functions may use a file resource along multiple paths, and each use raises

Match template

```
$:[[v]] = substr(:[a]);
if ($:[[v]] === false)
```

Rewrite template

```
$:[[v]] = substr(:[a]) || isset($maybe_false);
if ($:[[v]] === false)
```

Figure 5: The substr-model pattern informs PHPStan that a call to `substr` may return false.

Match template and rule

```
:[[f]](:[args])
```

```
where match :[args] with
| ":[_],:[_]" -> false
| ":[_]" -> true
```

Rewrite template and rule

```
:[[f]](:[args])
```

```
where rewrite :[f] with
| ":[_]free" -> "free"
```

Figure 6: The free-model pattern renames custom `XXX_free` functions to just `free`. It first matches all function calls, then filters calls that have only one argument using the `where match` rule. It then rewrites satisfying calls that contain the letters `free` using the `where rewrite` rule.

an error. For example, rewriting 16 cases in WordPress removes 44 false positives.

Pattern: substr-model. The `substr` function in PHP performs a substring operation. `PHPStan` models return value types of functions like `substr`. However, `PHPStan` did not track the fact that `substr` may return false if the length of the string is shorter than the requested substring range. `PHPStan`, thinking the value can only ever be a string, thus emits a false positive when a user's code checks whether the return value of `substr` is false, saying "comparison using `===` between string and false will always evaluate to false".

It took eight months as of the first user report for maintainers to implement the solution properly. It is particularly interesting that the maintainers were unwilling to make a simple change to the function model to reflect that `substr` can return false, because it would propagate spurious warnings when used as an argument in other contexts. However, users primarily had issues with the model when they *checked* return values, not when the value was used as an argument. Our transformation in Fig. 5 matches problematic cases that users experienced while avoiding the difficulty of changing the `substr` model wholesale. We introduce the expression `isset($maybe_false)` which lets `PHPStan` reason that the return value of an unset variable may be false. Statements that assign the result of `substr` before an `if` check are changed with an or-clause, which effectively remodels the `substr` call to possibly return a `false` value. Note that the use of `:[[v]]` in the assignment statement (line 1) and conditional expression (line 2) of the match template introduce a constraint that both matching instances must be syntactically *equal* for the rewrite rule to fire. This pattern removes six false positive reports in four PHP projects.

Pattern: free-model. `Infer` may timeout when analyzing functions and fail to summarize their effect. `Infer` reports false positive memory leaks when it fails to realize that a function frees memory—this happens particularly when the C library `free` call is wrapped

Match template

```
:[[sock]] = :[_].createSocket(:[sock], :[_], :[_], true);
```

Rewrite template

```
:[[sock]].close();
:[[sock]] = new Socket();
```

Figure 7: The create-socket pattern removes an unmodeled `createSocket` constructor that wraps a socket, and replaces it with an explicit, modeled sequence where the socket is closed and created again. The last argument in the match template (which implies the socket will be closed automatically) must be true in the original source for this rule to fire.

Match template

```
try (:[[o]] :[[v]] = new :[[c1]](new :[[c2]](:[args])))
```

Rewrite template

```
:[c2] wrapped = new :[c2](:[args]);
try {
  wrapped.close();
} catch (IOException e) {}
try (:[o] :[v] = new :[c1](wrapped))
```

Figure 8: The wrapped-resources rewrites the Java try-with-resources pattern where constructors wrap a file descriptor. The pattern closes the inner resource so that Infer stops tracking it (and stops it from reporting a leak.

inside custom free functions. The `OpenSSL` project follows the convention of wrapping free calls in functions like `OPENSSL_free`, which Infer fails to analyze. As an approximation of these custom free functions, our transformation (Fig. 6) rewrites the wrapping functions to call the plain C library `free` version. OpenSSL compiles successfully despite transforming 3,735 calls, and removes 6 false positives because Infer can then track the effect of `free` on memory for rewritten calls.

Pattern: create-socket. The `createSocket(socket, ..)` call is typically used in conjunction with the Java SSL library. It returns a server socket that wraps an existing `socket` in the first argument. The last argument, when `true`, tells the call that the underlying socket should be closed when the returned socket is closed. Infer fails to model the `createSocket` call, and a false positive report states that the underlying socket is leaked. The boolean toggle in the last argument makes this function difficult to model generically.

Interestingly, a user reports a false positive for a particular case where the last argument is always `true`. Our transformation (Fig. 7) addresses the issue by simulating the close operation early, and simply returning a fresh socket for `Infer` to track. Note that we would never persist this change in practice as it loses the implementation details of `createSocket`; however, it is sufficient for making the analysis more precise. This pattern removes two false positive reports in two Java projects; no resolution has yet been proposed in the Infer issue tracker.

Pattern: wrapped-resources. Java 7 introduced the try-with resources statement which automatically closes a resource after the block executes, preventing a leak. Infer reports a false positive leak

Match template and rule

```
try (:[x] :[[v]] = :[expr]) {
  :[body]
}:[rest]
```

```
where match :[rest] with
| " catch" -> false
| " finally" -> false
| ":[_]" -> true
```

Rewrite template

```
:[x] :[v] = :[expr];
try {
  :[body]
} finally {
  if (:[v] != null) {
    :[v].close();
  }
}
:[rest]
```

Figure 9: The null-on-resources rewrites a Java try-with-resource statement to a more traditional try-with statement. This avoids a redundant null-check injected by the compiler in Java versions 11 and 12, which leads to an analyzer warning. The match rule ensures that matches pass only if there does not already exist a `catch` or `finally` clause after a try-with-resources statement.

Match template

```
if(strcmp(:[1],":[2]"))
```

Rewrite template

```
const char *const t1 = ":[2]";
if(strcmp(:[1],t1))
```

Figure 10: The cons-strcmp pattern rewrites cases of `strcmp` so that constant strings in the second argument do not trigger macro expansion, thereby suppressing the false positive report.

when a resource constructor is nested inside another resource constructor within a try-with-resources statement (this happens, e.g., when passing a `FileInputStream` to an `InputStreamReader`). The underlying problem is similar to pattern **create-socket**: `Infer` fails to track that the wrapped resource will be closed. We use a conceptually similar transformation as in **create-socket**, but account for the syntactic variation introduced by try-with blocks (Fig. 8).

Pattern: null-on-resources. `SpotBugs` reports a redundant null check on resources inside try-with-resources statements (i.e., a resource is null checked after being previously dereferenced).[13] However, the error is only reported for code compiled with Java 11 and 12, and *not* Java 10. The reason is pernicious: the Java compiler in later versions *inserts a null check in the bytecode* which does appear to be indeed redundant. From the user's perspective, however, the report is a false positive—no null check is visible in the source code. The issue is cross-referenced by a large number of projects, and remains unresolved for over a year. Various projects have added annotations or disabled the check completely. No official solution has been proposed. The transformation in Fig. 9 converts a try-with-resources statement to a traditional try-catch-finally block. In effect, we normalize the try-with-resources syntax across Java versions to sidestep the null-check generation that only happens for Java versions 11 and 12. This suppresses three spurious bug reports in two of the projects.

Pattern: const-strcmp. The Clang Static Analyzer may report a potential out-of-bounds access when comparing strings with `strcmp`. This only happens when macro-expansion (defined in `glibc` headers) is triggered, in this case by the fact that a string literal is passed in the second argument to `strcmp`. The transformation in Fig. 10 extracts the string literal in the comparison to a string `const`,

[13]Note this bug is orthogonal to **wrapped-resources.**

Match template

```
strncpy(:[dst], :[src], :[len]);
if (:[len]) :[rest])) { :[dst])[:[idx])] = 0; }
```

Rewrite template

```
strncpy(:[dst], :[src], :[len] - 1);
:[dst][:[len] - 1] = '\0';
if (:[len] :[rest]) { :[dst][:[idx]] = 0; }
```

Figure 11: The pattern strncpy-null pattern. This transformation fires when the destination buffer `dst` has the same name as the buffer `dst` that is null-terminated subsequently, and the conditional check using `len` is dependent on the length `len` used in the call.

```
1     static char *php_ssl_cipher_get_version(const
       SSL_CIPHER *c, char *buffer, size_t max_len) {
2       const char *version = SSL_CIPHER_get_version(
       c);
3   -   strncpy(buffer, version, max_len);
4   +   strncpy(buffer, version, max_len - 1);
5   +   buffer[max_len - 1] = '\0';
6       if (max_len <= strlen(version)) {
7           buffer[max_len - 1] = 0;
8       }
```

Figure 12: A warning is emitted at line 4, where `strncpy` may not necessarily null-terminate `buffer`. It is a false positive: the buffer is always null-terminated in the case where `max_len` characters are copied. Our transformation makes the null-termination explicit to suppress the warning.

causing the analyzer to analyze the `strcmp` C library function rather than the macro expansion.

Pattern: strncpy-null The C `strncpy` function does not necessarily null-terminate its destination buffer, which can lead to memory corruption. `CodeSonar` warns about this possibility, but also notes that *if* a subsequent statement definitely null-terminates the string, then the warning can be ignored. We found that the warning was indeed a false positive in the `swoole` project: a subsequent check always null-terminates the buffer safely. However, a possible reason why `CodeSonar` conservatively reported an error is that the subsequent statement is guarded and was not considered safe (see line 10, Fig. 12). To avoid the false positive, our pattern checks whether a condition on the `strncpy` buffer length terminates that same buffer with a null character (Fig. 11). If so, the rewrite unconditionally null terminates the buffer. Although this transformation is not generally strong enough to match syntax that guarantees a null-terminated buffer, it does provide flexibility for refining analysis warnings. For example, we found exactly the same pattern using our template in the `PHP` source code, where it appears the `php_ssl_cipher...` function was borrowed from.

Pattern: snprintf-null `CodeSonar` reports an "Unterminated C String" error when a possibly unterminated string is passed to a function such as `strcat`. The analysis believes a string may not be null-terminated when, for example, space is allocated in the heap but not subsequently null-terminated. We identified two false positives where heap-allocated memory for a string *is* null-terminated, but only because we know that the buffer starts out with positive length and passes through `snprintf` which always null-terminates.

Match template

```
snprintf(:[dst], :[len], :[rest]);
```

Rewrite template

```
snprintf(:[dst], :[len], :[rest]);
if (:[len]) > 0) {
  :[dst][:[len]] = '\0';
} else {
  :[dst] = NULL;
}
```

Figure 13: The snprintf-null pattern models the possibility that the `snprintf` destination buffer may be null if the length is zero.

The analysis introduces imprecision where it believes the string can be of zero length, but does not, however, model the possibility that the buffer can subsequently be treated as a null pointer when passed to `snprintf`. Our transformation (Fig. 13) makes this result explicit, which suppresses two unterminated string warnings.

4.2.6 Discussion. We further characterize considerations and limitations of our approach.

Pattern Development. We found that developing patterns can take a few iterations to refine until they precisely match syntax of interest. For example, we iteratively added constraints in the **null-on-resources** pattern to filter out `try` statements that already contained `catch` and `finally` statements (without this constraint we would generate malformed programs). We expect users of our approach to similarly develop patterns incrementally. This process is analogous to existing practice at Google where analysis writers tune checks based on results of running over the codebase [31].

Tailoring Applicability and Usability. Analyzers on GitHub have numerous open issues related to "false positive" reports (`Infer` has 36 open issues and `PHPStan` has 32, and `SpotBugs` has 38), and our approach can fall outside the scope of these. For example, we may need type information to check whether a method or object may cause a false positive, while our approach is purely syntactic.

We note the compelling case for applying our technique in "black box" analysis settings. Users of commercial analyzers, like `CodeSonar`, do not have agency over the closed-source implementation or configuration options outside those provided by the distributor. Our approach demonstrates a new way to fine-tune results that is complementary to a black box analysis.

True positive warnings in the modified program may appear at different lines compared to the original program. As a usability concern, affected lines in the modified program should map to those in the original. This is primarily an engineering concern, as transformations keep track of precise changes in offsets so that no information is lost.

5 RELATED WORK

Program transformation has been used in various contexts to augment a procedure, technique, or system. Harman et al. introduce the idea of testability transformation [19, 20] where the goal is to transform a program to be more amenable to testing (e.g., by altering control flow) while still satisfying a chosen test adequacy criterion. Program transformation can improve fuzz testing coverage and reveal more bugs [29] and enable new crash bucketing strategies to accurately triage bugs [32]. Failure-oblivious computing [30] adds (for example) bounds checking that allow programs to execute through memory errors at runtime. We similarly develop source-to-source transformations; however, we focus on improving analysis output fidelity. Our technique also aims to improve a static procedure and thus must be fast enough to integrate into static workflows. Randomized program transformation is an approach for testing static analyzers [15] and compiler internals, and excels at finding bugs in optimization passes [16, 35]. Our approach differs generally from these in using tailored program transformations to deterministically rewrite syntax.

Program transformation on intermediate representations (IR) can improve analysis precision (e.g., by adding bounds on arrays [9, 13, 24]). Recent work formalizes the impact of program transformations on static analysis in the abstract [28]; for example 3-address code transformation can introduce analyzer imprecision [26]. These works adopt a predominantly semantic view of program transformations and their influence on analysis; Cousot and Cousot develop a language-agnostic framework for reasoning about the correspondence of syntax and semantics under transformation [14]. These ideas underlie our intuition that semantic changes can improve analysis. However, abstract representations are difficult for developers to manipulate. Our work promotes changing program syntax directly as a proxy for inducing semantic changes that enhance analysis reasoning. In practice, programs are difficult to generically transform [27]. Recent advances for transforming multi-language syntax [34] however allows us to implement our approach in an efficient and declarative manner.

In practice analyzers compromise on soundness [25] and implementation tradeoffs manifest as implicit tool assumptions that are difficult to trace and modify [12]. Existing work shows that analyzer configuration options and suppression mechanisms fall short of developer needs in practice [11, 22]. Recent work by Gorogiannis et al. [18] emphasizes the value of reducing false positives over false negatives, where the objective is to never report a false positive. In terms of this work, we introduce a new program transformational approach toward false positives while sidestepping the difficulties of modifying analyzer implementations or configurations.

6 CONCLUSION

We introduced a new approach for effecting changes in static analysis behavior via program transformation. Our approach uses human-written templates that declaratively describe syntax transformations. Transformations are tailored to suppress spurious errors and false positives that arise due to problematic patterns and limitations in analyzer reasoning. We made the observation that analysis users have little agency over the format of analysis configuration options provided to them, but that program transformation offers a fresh primitive for leveraging influence over analysis behavior. To this end we showed that manipulating concrete syntax can resolve diverse and long-standing issues in existing analyzers, where configuration and suppression mechanisms fall short. Our evaluation presents the first study for empirically validating this program transformational technique, which we evaluated on active analyzers and large real world programs.

REFERENCES

[1] 2019. Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[2] 2019. CodeSonar. https://www.grammatech.com/products/codesonar.

[3] 2019. Coverity: suppressing asserts. https://community.synopsys.com/s/question/0D534000046YuzbCAC.

[4] 2019. Error Prone: Patching. https://errorprone.info/docs/patching.

[5] 2019. Infer. https://github.com/facebook/infer.

[6] 2019. NullAway: auto-suppressing. https://github.com/uber/NullAway/wiki/Suppressing-Warnings#auto-suppressing.

[7] 2019. PHPStan. https://github.com/phpstan/phpstan.

[8] 2019. Spotbugs. https://github.com/spotbugs/spotbugs.

[9] Josh Berdine, Arlen Cox, Samin Ishtiaq, and Christoph M. Wintersteiger. 2012. Diagnosing Abstraction Failure for Separation Logic-Based Analyses. In *Computer Aided Verification (CAV '12)*. 155–173.

[10] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (NFM '15)*. 3–11.

[11] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *International Conference on Automated Software Engineering (ASE '16)*. 332–343.

[12] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2015. An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '15)*. 336–354.

[13] J. Robert M. Cornish, Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2014. Analyzing Array Manipulating Programs by Program Transformation. In *Logic-Based Program Synthesis and Transformation (LOPSTR '14)*. 3–20.

[14] Patrick Cousot and Radhia Cousot. 2002. Systematic design of program transformation frameworks by abstract interpretation. In *Symposium on Principles of Programming Languages (POPL '02)*. 178–190.

[15] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods (NFM '12)*. 120–125.

[16] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *PACMPL* 1, OOPSLA (2017), 93:1–93:29.

[17] Pär Emanuelsson and Ulf Nilsson. 2008. A Comparative Study of Industrial Static Analysis Tools. *Electr. Notes Theor. Comput. Sci.* 217 (2008), 5–21.

[18] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A true positives theorem for a static race detector. *PACMPL* 3, POPL (2019), 57:1–57:29.

[19] Mark Harman. [n.d.]. We Need a Testability Transformation Semantics. In *Software Engineering and Formal Methods (SEFM '18)*.

[20] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability Transformation. *IEEE Trans. Software Eng.* 30, 1 (2004), 3–16.

[21] Ciera Jaspan, I-Chin Chen, and Anoop Sharma. 2007. Understanding the value of program analysis tools. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07)*. 963–970.

[22] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *International Conference on Software Engineering, 2013 ('13)*. 672–681.

[23] William Landi. 1992. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (dec 1992), 323–337.

[24] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization (CGO '04)*. 75–88.

[25] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[26] Francesco Logozzo and Manuel Fähndrich. 2008. On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In *Compiler Construction (CC '08)*. 197–212.

[27] Eduardus A. T. Merks, J. Michael Dyck, and Robert D. Cameron. 1992. Language Design For Program Manipulation. *IEEE Trans. Software Eng.* 18, 1 (1992), 19–32.

[28] Kedar S. Namjoshi and Zvonimir Pavlinovic. 2018. The Impact of Program Transformations on Static Program Analysis. In *International Symposium on Static Analysis (SAS '18)*. 306–325.

[29] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*.

[30] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing.. In *OSDI*, Vol. 4. 21–21.

[31] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66.

[32] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic Crash Bucketing. In *International Conference on Automated Software Engineering (ASE '18)*. 612–622.

[33] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *International Conference on Software Engineerinng (ICSE '18)*. 151–162.

[34] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Conference on Programming language Design and Implementation (PLDI '19)*.

[35] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. [n.d.]. Finding and understanding bugs in C compilers. In *Conference on Programming Language Design and Implementation (PLDI '11)*.