Reorganizing and Optimizing Post-Inspection on Suspicious Bug Reports in Path-Sensitive Analysis

Xutong Ma[1,3], Jiwei Yan[2], Jun Yan[1,2,3,†] and Jian Zhang[1,3]
[1]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences*
[2]*Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences*
[3]*University of Chinese Academy of Sciences*
Email: {maxt, yanjw, yanjun, zj}@ios.ac.cn

***Abstract*—To efficiently prune infeasible program paths, path-sensitive static analysis based bug detectors may utilize light-weight imprecise methods to check the satisfiability of path constraints, which leads to redundant reports and false-positives. Although the false-positives can be eliminated by the post-inspection process, which re-checks the feasibility of the paths of each bug report with precise methods, the redundant reports are inspected unnecessarily.**

In this paper, we discuss how to improve the efficiency of the post-inspection process. We categorize the uninspected reports into disjoint sets and sort the reports in each category, which helps to decrease the number of inspection attempts. Besides, we parallelize the inspection for further speedup. The experimental results indicate that about 65.20% of needless inspections are eliminated in total. With the sorted category sets, about 52.4% of attempts are additionally reduced. And compared with the sequential execution, the parallel approach further gains an average speedup of 5.74 under 8 threads.

***Keywords*-Path-Sensitive Static Analysis; Post-Inspection; Suspicious Bug Report**

I. Introduction

Static program analysis is a method that automatically analyzes the behavior of the program without executing it, which is commonly used to ensure the reliability of complex software systems. In an analyzer, path-sensitive analysis methods are often used to acquire additional precision [1], [2]. Compared with path-insensitive methods, which merge the program states from different paths, path-sensitive methods will separately trace different program paths and maintain program states for every path, which will always encounter the path explosion problem [3].

To overcome the path explosion problem during path exploration, the infeasible paths should be pruned. And to soundly prune infeasible paths, the *path constraints* (or path conditions, PC for short) should be checked when forking the program states on a branch statement. And there are mainly two kinds of methods to check the PC: the precise solvers, or the light-weight checkers.

For the precise solvers based methods [4], [5], they check the satisfiability of the PC through some precise constraint solvers, which is usually an SMT solver. As these methods may always excessively rely on these inefficient solvers to verify every branch condition, the analysis time may always be dominated by the solvers they choose [3].

For the light-weight checkers based methods [6]–[8], they use some light-weight approximated methods, such as interval arithmetic, Value-Set Analysis (VSA), *etc.*, to check the branch conditions and roughly prune most of the infeasible paths. Since the approximations may also introduce false-positives, an additional inspection process is added after the analysis to re-check these bug reports with suspicious paths (we call them suspect reports, which describe the suspicious paths that are only checked during analysis) and refute the false-positives.

Compared with the first kind of methods, the most important advantage of the second is that they can run much faster, enabling them to handle large software [9]. However, as more program paths are explored, more suspect reports will be generated for post-inspection. Although the maximum number is limited to the number of paths explored, there can still be hundreds or thousands of suspect reports generated, which makes the post-inspection process inefficient.

In addition, among the gush of suspects, there will also be a large portion of reports that describe the same bug through different trigger paths. And these circumstances happen frequently when there are multiple paths that can trigger the same bug. Since these *redundant* reports need a great effort to be inspected and have no help to the tool users, these inspections should be eliminated. However, as their trigger paths are different from each other, it is very difficult for a straight-forward deduplication method to prune them. Besides, since the redundant reports will not affect the accuracy of analysis results, it will not always be attractive for the tool developers to handle the issue, and make the tool underused in practice [10].

Facing these problems, on one hand, some works attempt to remit the problems by pruning redundant paths [11]. However, these methods require precise constraint solving during exploration, which are hard to be transplanted to the analysis methods depending on imprecise constraint solving.

[†]Corresponding author

In addition, as the visited paths may be infeasible when using light-weight checkers, but some feasible paths may be eliminated by the path redundancy pruning, these methods may also introduce false-negatives.

On the other hand, some works attempt to eliminate redundant reports [12], [13]. However, as these methods are always designed for reports generated by a precise tool, whose number is always little, the efficiency issue makes these methods also inapplicable to the circumstances when there are lots of suspect reports.

In this paper, we propose a categorization method to help the post-inspector eliminate the unnecessary inspections on the redundant reports. We categorize all the suspect reports into disjoint categories with respect to their bug types and trigger operations, and leave the trigger paths not restricted. By keeping only the first *feasible report* (the report with a feasible path verified by the post-inspection process) in each category and dropping other reports, the unnecessary inspections can be eliminated. As there may be a lot of suspects and categories to be inspected, we introduce parallel techniques to acquire additional speedups and make our inspector execute much more efficiently.

We implement these post-inspectors as the *non-categorized inspector* (the original post-inspection method), *categorized inspector* or *sequential inspector* (with categorization method) and *parallel inspector* (the parallel version of our categorized inspector). We evaluate these post-inspectors on 299,960 suspect reports extracted from our benchmark. The experimental results indicate that about 98.82% of the extracted reports are redundant, which can be separated into categories with an average of 177 reports, and our methods can eliminate 65.20% inspections in total, and eliminate 97.13% inspections for the *feasible categories* (the categories with at least one feasible report). By sorting the reports in categories with different strategies, we can reduce inspection attempts used to find the first feasible report by 52.4%. Besides, our parallel inspector reaches an average speedup rate of 5.74 under 8 threads, and 10.59 under 16 threads compared with the sequential inspector.

To sum up, the main contributions of our paper are as follows:

- We introduce a light-weight categorization method to separate the suspect reports of a program into disjoint categories and use the categories to efficiently eliminate inspections on redundant reports.
- We find an effective sorting strategy for the reports in each category, to make the inspector attempt the least times to find a feasible report.
- We introduce parallel techniques to further expedite the post-inspection process and keep it efficient even if there are numbers of suspects to be inspected.

This paper is organized as follows: Section II introduces the post-inspection based static analyzer and uses a motivating example to present the problems of the post-inspector; in section III, we describe our categorization method, sorting strategy and parallelized inspector in detail; the evaluation of our improvements is provided in section IV; and at last, the related works and conclusion are presented in section V and VI respectively.

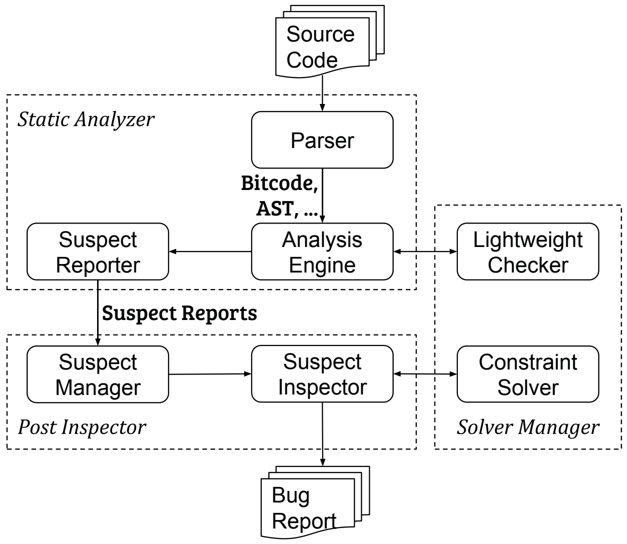


Figure 1: Structure of the LC & PI based static analyzer

II. Background

In this section, we first explain the structure of the original post-inspection based static analysis method, and separately introduce the components of the static analyzer and post-inspector. Then we use an example to illustrate the problems and requirements of the new post-inspector.

A. Post-Inspection and Path-Sensitive Static Analysis

To remit the efficiency issue of precise solver based path-sensitive analysis methods, the light-weight checker and post-inspection (LC & PI) based analysis method is proposed. In the method, some light-weight approximated methods, instead of precise solvers, are used to check the PC and roughly prune the infeasible program paths. As these light-weight PC checking methods are not precise enough, the introduced false-positives cannot be eliminated until the post-inspection process, which re-checks the PC of reports using precise solvers to refute the infeasible ones.

Figure 1 shows the general structure of the LC & PI based static analyzer. There are three parts in the system: the static analyzer, the post inspector, and the solver manager. The static analyzer processes the input source code and generates suspect reports. And the post-inspector inspects the suspects and drop the infeasible ones. During the processes, both the static analyzer and the post-inspector query the solver manager for checking or solving the PC.

When analyzing a program, the source code of the program will be first parsed and converted to the intermediate representation (bitcode, AST nodes or *etc.*, IR for short) used in the analyzer by the *parser*. Then the IR will be sent to the *analysis engine*, which will carry out the analysis and query the *constraint checker* to prune the infeasible program paths. When a bug is found, the information related to the bug (bug type, PC, trigger path trace, *etc.*) will be sent to the *suspect reporter* for generating a *suspect report*. And then the suspect report will be sent to the *suspect manager* of the post-inspector for further inspection.

When a suspect report reaches the *post-inspector*, the report will be first stored and managed by the *suspect manager*. Then when the static analyzer halts at some stage (*e.g.* if the granularity of analysis is functions, the analyzer halts when the analysis on a function is finished), the *suspect inspector* will start inspecting the suspect reports. The inspector traverses the reports stored in the *suspect manager* and queries the *constraint solver* to verify the PC of each report in batch. If the trigger path of the suspect report is feasible, the suspect report will be presented to the users as a true *bug report*. Otherwise, the report will be marked as infeasible and dropped.

B. Motivating Example

Fig. 2 shows a motivating example. Function `getbuf` is used to select the proper buffer memory with the given buffer size requirement (`size`) and a pointer to a fast buffer (`fastbuf`). When fast buffers are enabled (`usefb` is `true`), and the size requirement is not greater than the threshold `BUFFER_SIZE` (line 3), the fast buffer `fastbuf` is used as the buffer memory (line 4). Otherwise, function `malloc` is invoked to allocate memory for the buffer (line 5). Function `trigger` is the entrance of the example code. It allocates a fast buffer `fastbuf` on the stack, and then invokes function `getbuf` for buffer selection (line 10). After the buffer is selected, it will be used in a snippet of complex code with a lot of branches and sub-function invocations. In the example, we use a branch statement to represent this complex code snippet (line 12), and we suppose that all the branch conditions can be correctly judged with the light-weight PC checkers. And finally, the function checks whether fast buffers are disabled, and free the buffer memory if so (line 14).

The bug in this code snippet is simple. When fast buffers are enabled (`usefb` is `true`), and the buffer size requirement (`size`) is no less than the threshold `BUFFER_SIZE`, the allocated buffer from line 5 will not be freed on line 14, which causes a *Memory Leak* problem. Since all paths traversing from the memory allocation (line 5) to the end of the function `trigger` will lead to the problem, the analyzer may generate two suspect reports for the same bug; we call these reports *redundant reports*.

```c
1  char *getbuf(int size, char *fastbuf) {
2    char *ret = NULL;
3    if (usefb && size < BUFFER_SIZE)
4      ret = fastbuf;
5    else ret = malloc(size+1);
6    return ret;
7  }
8  void trigger(int size) {
9    char fastbuf[BUFFER_SIZE];
10   char *buf = getbuf(size, fastbuf);
11   // A snippet of complex code.
12   if (...) { ... }
13   // Deallocate the buffer.
14   if (!usefb) free(buf);
15 }
```

Figure 2: A motivating example

When the light-weight PC checkers fail to judge the conflict between line 3 and line 14, the analyzer may traverse an infeasible path that chooses the fast buffer as the buffer memory (line 4) and tries to free the buffer at the end of the function `trigger` (line 14), which describes a *Free of Memory not on the Heap* problem. Therefore, the analyzer will generate a suspect report for the "bug", we call these kinds of reports *false-positives*.

For the false-positives, since their PCs are unsatisfiable, they can be finally refuted by the precise constraint solver during the post-inspection. However, the redundant reports will not be pruned, as their PCs are satisfiable. Since all reports in one category describe exactly the same bug, it is unnecessary to inspect the other redundant reports when a feasible one is found. Besides, although only two reports are extracted from the example code, if the branch statement on line 12 is replaced with real-world complex program control flow, hundreds or thousands of suspect reports will be extracted, and the circumstances may become times worse when considering other functions calling it. Therefore, it will take a lot of time to inspect all these redundant reports. In our benchmark, the number of suspect reports of the largest category we extracted from a function together with its callers is 34,107 in total.

C. Problems of Redundant Inspection Elimination

The main problems of redundant inspection elimination can be summarized as the following three points. First, the LC & PI based static analyzer requires a post-inspection process to inspect the reports. However, it also generates too many redundant suspect reports, and these reports are difficult to be eliminated by the analyzer. Second, the state-of-the-art bug report categorization or deduplication methods cannot efficiently handle such many suspect reports. However, the feasibility of a suspect report can only be exactly known with inspections. To expedite the inspection process, one reasonable way is to eliminate inspections on redundant reports. And third, the tool developers pay little attention to the efficiency issue of the post-inspector caused

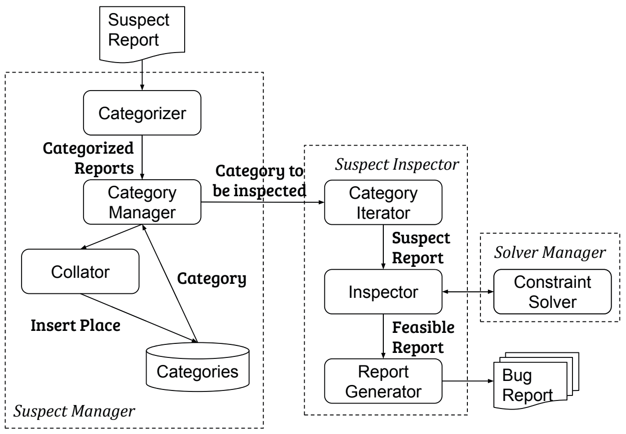


Figure 3: Structure of the categorized inspector

by redundant reports.

Because of the motivations mentioned above, we need to improve the post-inspector in the following four factors:

- We need to illustrate how many redundant reports will be generated when we do not consider the path redundancy elimination in the static analyzer.
- A light-weight report categorization method is required, which can recognize reports on the same bug with different trigger paths.
- A sorting strategy is needed for the reports in categories, which can statistically guide the inspector to use the least inspections to find a feasible report when we do not know their feasibility.
- We need to additionally expedite the inspection when there are too many categories, or some categories contain too many infeasible reports.

III. Improved Post-Inspector

In this section, we mainly discuss our improvements to the post-inspector. Since the categorization of bug reports and error messages is widely used to cluster and eliminate redundant ones [13], we also explore using categorization methods to eliminate inspections on redundant reports.

A. Categorizing the Suspect Reports

When improving the post-inspector, its inner components should also be updated. Figure 3 presents the detailed inner components of the categorized inspector. The detailed modifications on its inner components will be mentioned in the following two subsections.

1) Categorization Based Post-Inspection: In the *suspect manager*, we try to categorize the suspect reports with their *bug type* and *trigger operation*. The bug type depicts the *characteristic* of the bug, which is the main indicator to differ reports from each other; while the trigger operation refers to the corresponding statement or expression that makes the defect happen, which provides both the *condition* and the *trigger* of the bug.

Recall the example presented in figure 2, for a report describing the bug mentioned in section II-B, its bug type is the *Memory Leak* problem, and its trigger operation is the implicit `return` statement of the function `trigger` (*trigger-site*), and the allocation of the leaked memory in function `getbuf` (the *reason-site*).

The bug type and the trigger operation contain the characteristic, the reason and the trigger place of the bug. Therefore, the category of the bug can be uniquely determined based on these aspects, and all the reports in one category can also describe the same bug.

In the categorized inspector, when a suspect report is received, it will be first tagged to a category, and then stored into the corresponding category storage in the *suspect manager*. When inspecting the reports, the *suspect inspector* will iterate the categories stored in the *suspect manager*, and inspect the reports in each category one by one. When a feasible report is found for the category, the iteration on the category will be terminated immediately and continue with the next category. By dropping the rest reports in a category, unnecessary inspections are eliminated, which also reduces the total time spent on inspecting a category.

2) Sorting the Reports: Since we try to find the first feasible suspect in a category, the fewer infeasible suspects at the front of the category list, the fewer unnecessary inspections there will be, and the faster the inspector can run. Therefore, the suspects should be sorted in ascending order by the difficulty to be inspected as feasible.

It is a common sense that reports with longer *trigger paths* contain more *constraints* and *variables*, and vice versa. And a conclusion has been proved that for a set of SAT clauses, the lower the clause-variable ratio an expression has, the more likely it is satisfiable; and the higher the ratio is, the more likely it is unsatisfiable [14]. Although unfortunately there are few SMT related conclusions, in analogy with such conclusions, we can still conjecture the efficiency of constraint solvers.

Therefore, we choose four feature indicators: the *length* of the trigger path, the number of *clauses* and *variables*, and the clause-variable *ratio* as the sorting criteria. A preliminary experiment is to find out which sorting criterion can minimize the number of inspections. According to the experimental results in section IV-C, the categories sorted with *length* have the least number of inspections. Therefore, we suggest sorting the reports with their *length*s.

Based on the structure of the categorized inspector, we add a *collator* to the *suspect manager*. When a suspect report is tagged to a category, before storing it to the category storage, the collator will find a proper place for the report based on the sorting strategies. And finally, the report will be inserted into the corresponding place.

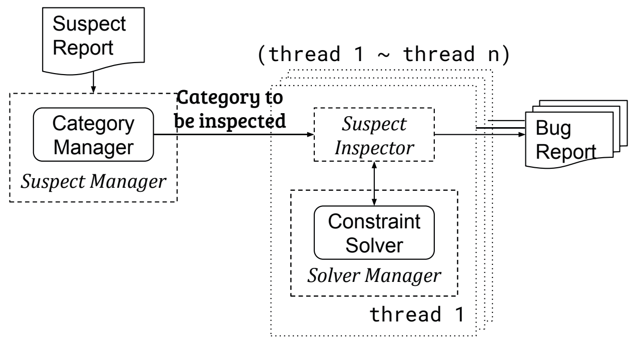


Figure 4: Structure of the parallel inspector

B. Parallel Inspector

Although the categorized inspector can greatly reduce the number of reports to be inspected, there are still two circumstances that the improved inspector still cannot handle efficiently by categorizing and sorting the reports.

The first one is that when the static analyzer generates too many categories of suspect reports. As the number of reports to be inspected will not sharply decrease with categorization, it still needs to take a lot of time to inspect these categories.

And the second one is that when there are too many infeasible categories, who contain no feasible reports in them. According to the categorization method, all the reports in the infeasible category will finally be inspected. Therefore, it will take much more time for the inspector to inspect an infeasible category than a feasible category of the same size.

In this section, we mainly discuss the challenges of parallelizing the inspection and our scheduling strategies to handle these challenges.

1) Structure of Parallel Inspector: To further expedite the inspector, especially under the above two circumstances, we introduce the parallelized optimization to the categorization method. In the parallel inspector, each working thread runs an instance of the suspect inspector together with the constraint solver. When executing the parallel inspector with only one working thread, it will act in the same way as the sequential categorized inspector. Figure 4 shows the structure of the parallel inspector.

There are two different kinds of granularity to split the inspections to different working threads: on the granularity of reports or categories. When scheduling working threads on the granularity of reports, different threads may inspect the same category simultaneously. As a smaller granularity is used for scheduling, the threads can take full advantage of the system resources. However, according to the categorized inspection method described in section III-A, when the suspect inspector running on one thread confirms a report to be feasible, all other threads working on the same category should synchronize the result, terminate their own inspections immediately and continue with the next category. The synchronization between the threads working on the same category will cause a serious data race problem, which will greatly affect the efficiency of the execution.

When scheduling working threads on the granularity of categories, different threads will separately inspect different categories. As a larger scheduling granularity is used, there will be fewer data races. However, there are mainly two challenges for the parallelization. First, as the number of reports in each category differs greatly from each other, a simple scheduling strategy will lead to great overload imbalance. Second, since the number of reports in a category has no relation with the inspections finally carried out. We are not able to know how many reports will be exactly inspected, which means that it is very hard to schedule the jobs based on their sizes.

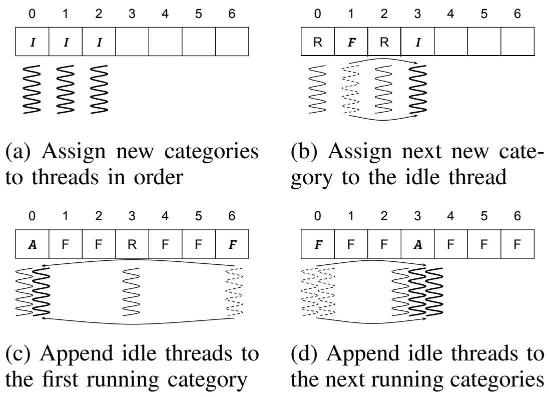
2) Scheduling Strategies of Working Threads: To solve the above challenges, the inspector instance running on each thread inherits from the sequential inspector and merges the two different kinds of scheduling granularity. The updated process of iterating categories to inspect their inner reports is presented in Algorithm 1.

Algorithm 1 Process of requesting for categories.

```
 1: function CATEGORYSCHEDULER
 2:     for all  category ∈ CategoryManager do
 3:         if  CategoryManager.Acquire(category) then
 4:             SuspectInspector.Inspect(category)
 5:         end if
 6:     end for
 7:     for all  category ∈ CategoryManager do
 8:         if ¬CategoryManager.Finished(category) then
 9:             SuspectInspector.Inspect(category)
10:         end if
11:     end for
12: end function
```

To combine the advantages of both kinds of scheduling granularity, the parallel inspection is separated into two stages. In the first stage (from line 2 to line 6), the inspector on each thread will simultaneously traverse the categories in the category manager, and apply for the categories one by one with the *Acquire* method of the category manager (line 3). In the method, the category manager will atomically check whether the category has been assigned to another thread, and assign it to the caller thread if not. As a consequence, there will be no categories inspected by more than one thread in stage one, which makes the threads scheduled on the granularity of categories. Therefore, the categories can be quickly covered.

When all the categories have been assigned, the second stage begins (from line 7 to line 11). When the inspector finishes the first traverse on the categories, it will traverse the categories again, and apply for the categories which are still under inspection with the method *Finished* (line 8). In



(a) Assign new categories to threads in order
(b) Assign next new category to the idle thread
(c) Append idle threads to the first running category
(d) Append idle threads to the next running categories

Figure 5: Schedule strategies of working threads

the method, the category manager will atomically query the category whether it has been finished by other threads, and append the caller thread to it if not. As a consequence, there will be multiple threads working on the same category in this stage, which makes the threads scheduled on the granularity of reports. Therefore, the inspector can make full use of idle threads, and make the unfinished categories can be inspected faster. To prevent one report from being inspected by more than one thread, a flag is added to mark whether it is being inspected by other threads. And the thread will atomically check and mark the flag first, before inspecting a report.

In addition, in the second stage, when a working thread finds a feasible report, it will first notify the category to record the report. Then it notifies all other threads working on the category to abort their unfinished inspection. When a thread receives a notice that a feasible report has been found for the category that it is working on, it will stop the constraint solver and abort the report. In the parallel inspector, it is possible that the generated bug report is not the first feasible report in the category. However, as its inspection spends less time, we believe that it may also be easier or the same for developers to review the report. Therefore, we choose to keep the report instead of the one with a greater chance to be feasible.

3) Example of Scheduling Strategies: Figure 5 shows the four snapshots of scheduling 3 threads to inspect 7 categories. The numbered cells represent the categories to be inspected, and the curves indicate the threads that are working on the category. An empty cell represents the category is pending for inspection, while the letter in the cell indicates the status of the inspection: a letter *I* means the category has just been assigned to a thread and the inspection is about to start at this step, a mark *R* refers to the inspection on the category is running now, a sign *A* represents the category has just been appended with new thread(s) at current state, and an *F* indicates the inspection on the category has already been finished. The Roman letter indicates the category has been in this status for a period, while the italic letter represents the category has just been transferred to the status at this moment.

Figure 5a represents that at the beginning of the inspection, the categories will be assigned to threads one by one, despite the order of the threads. When a thread finishes its inspection, the next pending category will be assigned to it (figure 5b). Figure 5c indicates that when a thread finishes its inspection, and all categories have been assigned, it will be appended to the first category that is still under inspection. And the threads will be appended to the next running category when the appended inspection is finished, which is presented in figure 5d.

IV. Evaluation

To evaluate the effectiveness of our improvements, we carry out three groups of experiments to answer the following three research questions:

- **RQ 1**: What is the ratio of redundant reports? How many inspections are eliminated and how much time is saved by the elimination?
- **RQ 2**: How many inspections are required under different sorting strategies? Which strategy needs the least number of inspections?
- **RQ 3**: Compared with the sequential inspector, how much speedup can be acquired with different numbers of working threads?

The first half of **RQ 1** is answered in section IV-B, which is used to illustrate the seriousness of the inspection redundancy problem, and persuade the tool developers to pay attention to it. The second half of **RQ 1** and the **RQ 2** are used to evaluate the effectiveness of the sequential categorized inspector and provide the preliminary experimental results mentioned in section III-A2, whose answers can be separately found in section IV-D and section IV-C. And **RQ 3** is used to present the effect of our parallel inspector compared with the sequential inspector, and is answered in section IV-E.

In this section, we will first show the implementation of our methods. Then, we introduce the system environments and the benchmark instances we used for the experiments. And at last, we present our experimental results and answers to the research questions in detail.

A. Experiment Setup

1) Implementation and Experimental Environments: We removed the post-inspector of `Canalyze` [9], a static symbolic execution based program defect checker, to extract bug reports together with the PCs of their trigger paths from the benchmark. The PCs are dumped in the syntax under the SMT-Lib v2 standard.

Our post inspection engine is separately implemented in `C++`. In our implementation, the parallelization is implemented with the `std::thread` class in `C++11`, which is

Table I: Composition of benchmark.

Program	Version	Reports	Size(KiB)
bftpd	v3.8	4,698	15,341
binutils	v2.31.1	31,020	422,767
bison	v3.2	42,756	490,754
coreutils	v8.30	17,476	182,398
curl	v7.21.1	34,300	242,382
gdb	v8.2	27,906	333,443
gzip	v1.9	2,252	12,213
libosip2	v4.0.0	23,256	425,200
lighttpd	v1.4.32	8,502	78,169
nano	v3.1	24,293	279,076
readline	v7.0	4,416	30,230
screen	v4.6.2	7,568	65,619
sqlite	v3230100	63,441	3,848,629
tar	v1.30	8,076	66,049
Total	-	299,960	6,492,270

Table II: Number of Categories and Reports.

Program	Rep.	Feas.	Cate.	Gen.	Redu.
bftpd	4,698	3,861	47	46	98.81%
binutils	31,020	14,234	299	212	98.51%
bison	42,756	11,075	57	31	99.72%
coreutils	17,476	7,165	117	60	99.16%
curl	34,300	59	36	19	67.80%
gdb	27,906	11,162	180	114	98.98%
gzip	2,252	1	4	1	0.00%
libosip2	23,256	7,300	115	97	98.67%
lighttpd	8,502	4,725	221	190	95.98%
nano	24,293	110	19	9	91.82%
readline	4,416	1,198	25	22	98.16%
screen	7,568	1,199	38	30	97.50%
sqlite	63,441	37,917	467	351	99.07%
tar	8,076	3,401	67	40	98.82%
Total	299,960	103,407	1,692	1,222	98.82%

an encapsulation of the `pthread` library; and the atomic operations are implemented with the `std::atomic` template in `C++11`. We invoke `Z3` [15] as the constraint solver to inspect the PC of bug reports.

We set up all of our experiments on a Linux server with Intel® Xeon® E5-2680 v4 CPU of 2.40 GHz and 64 GB of memory. And we use `docker`[1] to manually limit the number of CPU cores available equaling to the number of working threads when executing the parallel inspector. All the time is evaluated with the average of 5 measurements.

2) Benchmark: Table I provides the detailed information of our benchmark. The first two columns present the name and version of the benchmark instance. The third column provides the number of all suspect reports. And the size of the *trigger path constraint files* is in the last column. Since these constraint files are encoded in SMT-Lib v2 syntax, it will be more reasonable to measure these files with their file size (in KiB) instead of source lines of code.

The benchmark is composed of different kinds of open source software projects, which can be on behalf of many fields of application. As different fields may use different kinds of code styles, the corresponding patterns of the PC and percentages of false-positives may also be different. Using the benchmark composed in this way can make our benchmark more comprehensive and extensive. Besides, all benchmark instances are checked by `Canalyze` directly with the code of the release version listed in the table, and there are no additionally inserted bugs in the code.

B. Number of Categories and Inspections

Table II presents the statistics of bug reports generated by both versions. The first column is the name of the benchmark instance. The second (*Rep.*) and the forth (*Cate.*) column represent the number of bug reports and categories respectively. And these two columns are also the upper bounds of the number of bug reports that can be generated by the non-categorized inspector and the categorized inspector. The third and the fifth column represent the number of bug reports generated by the non-categorized checker (*Feas.*) and the categorized checker (*Gen.*), and these two columns are also the total number of all feasible bug reports and all feasible bug report categories. And the last column (*Redu.*) presents the reduction ratio on bug report numbers.

According to the statistics, the non-categorized inspector generates 103,407 bug reports for all the benchmark instances, while the categorized inspector generates 1,222 bug reports for them, which indicates that about 98.82% of suspect reports are redundant in total. For most of the benchmark instances, the analyzer can generate more than 90% of redundant reports according to our categorization method. The results answer the first half of **RQ 1** that the redundant reports frequently occur among the suspect reports, which makes the post-inspector waste a lot of effort on inspecting these reports.

Among the benchmark instances, there are only two exceptions, `curl` and `gzip`, which also generate a similar number of suspect reports but provides a very low redundancy percentage. The reason for this circumstance is that these two benchmark instances contain too many false-positives, which are eliminated during the inspection of both inspectors. Therefore, there is very limited space for the categorized inspector to reduce inspections.

C. Inspection Numbers of Different Suspect Report Orders

In this part, we mainly discuss the inspection numbers under different suspect report orders, which answers **RQ 2**. As mentioned in section III-A1, the order can affect the number of inspections required before the first feasible one is found, which can further affect the efficiency of the post-inspection process, hence we evaluated the number of inspections under different orders.

There are four sorting criteria mentioned in section III-A2, sorting the suspect reports with *length*, *variable*, *clause*, and *ratio*. Besides, we can also keep the reports unsorted,

[1]Enterprise Container Platform — Docker, *https://www.docker.com/*

Table III: Number of inspections under different order compared with original order.

Program	Length	Variable	Clause	Ratio	Random
bftpd	0	0	0	0	0.00
binutils	-162	-107	-45	386	-103.80
bison	-37	2,941	4,516	8,297	1,777.80
coreutils	9	38	30	252	259.80
curl	0	-6	-6	0	-2.80
gdb	-119	-64	-4	424	42.60
gzip	0	0	0	0	0.80
libosip2	-35	-1	19	24	171.80
lighttpd	10	14	24	59	114.60
nano	57	161	177	51	42.40
readline	-870	-870	-870	-551	-2.20
screen	-160	-160	-160	-160	-40.00
sqlite	-457	-337	-295	175	-567.00
tar	-159	-147	-157	-128	-124.00
Total	-1,923	1,462	3,229	8,829	1,570.00

Table IV: Reduction of inspection number.

Program	Non-Categorized		Categorized	
	All	Feasible	All	Feasible
bftpd	4,698	3,861	883	46
binutils	31,020	24,985	6,455	420
bison	42,756	36,069	7,260	573
coreutils	17,476	12,183	5,388	95
curl	34,300	75	34,250	25
gdb	27,906	24,417	3,727	238
gzip	2,252	130	2,153	31
libosip2	23,256	19,856	3,498	98
lighttpd	8,502	6,358	2,355	211
nano	24,293	307	24,054	68
readline	4,416	4,392	48	24
screen	7,568	6,853	745	30
sqlite	63,441	53,570	10,922	1,051
tar	8,076	5,481	2,653	58
Total	299,960	198,537	104,391	2,968

which preserves the reports in the time order of when they are generated. Therefore, we select the unsorted categories as the baseline to evaluate the performance of different sorting strategies. Since infeasible categories have the same number of inspections no matter how they are sorted, we only consider the result of 1,222 feasible categories.

Table III presents the results. The value in each cell indicates the number of invocations under the specific sorting strategy minus the number of invocations without sorting (*original exploring order*). And in the *random* column, the value is measured with the average of five random shuffles, to further the random order of the reports. The less the number is, the better the strategy will be.

According to the table, the *length* strategy is definitely the best choice, as the strategy is the only negative value in total. And under all benchmark instances, the strategy also has the least value among almost all sorting strategies. Besides, compared with the best *length* strategy, the unsorted category can also have an acceptable result in 6 out of 14 benchmark instances (the values no less than 0 in the *Length* column indicate the unsorted categories have the least number of inspections). For the other three sorting strategies, we do not recommend using them to sort the reports.

The result indicates that, as a consideration of simplicity and performance, we should choose the *length* strategy to sort the suspects in a category. And we also sort the suspect reports with *length* strategy by default, in all the following experiments whose inspection numbers are measured. Besides, when the sorting overhead is too high, keeping the category unsorted can also be a good choice.

D. Reduction of Number and Time Cost of Inspection

In this part, we consider the reduction of the number and the time cost of the inspections. We run the categorized inspector against the non-categorized inspector on the benchmark, and measure these two kinds of values. The experiments mainly answer the second half of **RQ 1**.

1) Reduction of Inspection Numbers: We now discuss the reduction of inspections. Since the goal of our improvements is to eliminate unnecessary inspections on redundant reports, we measure the number of inspections for each category on our categorized inspector, and compare the number on the non-categorized inspector.

Table IV presents the statistics about the number of inspections of both non-categorized and categorized inspector. The first column is the name of the benchmark instances. The following two columns indicate the number of inspections on the non-categorized inspector (***N**on-**C**ategorized*) when inspecting the suspect reports in all categories (***A**ll*) or in only feasible categories (***F**easible*) respectively; while the last two columns present the corresponding numbers of the categorized inspector (***C**ategorized*).

According to the table, the total reduction ratio of inspections is about 65.20% ($1 - C.A/NC.A$), while the ratio is about 98.51% when considering only feasible categories ($1 - C.F/NC.F$). And the total inspection number of only feasible categories also decreases sharply in most of the benchmark instances, which indicates that most of the redundant inspections are eliminated in the categorized inspector; when considering the infeasible categories, the statistics indicate that inspecting the infeasible suspects in these categories may be the most time-consuming part of the entire inspection, which indicates the circumstances of spending a lot of time inspecting infeasible categories do exist.

2) Reduction of Actual Execution Time: We now discuss the time reduction of our categorized inspector, which is also related to the number reduction of inspections. Since our goal is to efficiently eliminate unnecessary inspections, the execution time is also important to be evaluated.

Table V presents the statistics about the time cost of inspecting all categories under both inspectors. The first column provides the name of the benchmark instances. The second column (*Non-Cate.*) and the third column (*Cate.*) present the actual execution time of the non-categorized

Table V: Reduction of actual execution time.

Program	Non-Cate.	Cate.	Dec.
bftpd	102.09	6.34	93.79%
binutils	2,956.22	127.31	95.69%
bison	1,036.38	72.45	93.01%
coreutils	982.58	49.65	94.95%
curl	445.93	270.46	39.35%
gdb	3,293.20	73.72	97.76%
gzip	23.37	14.64	37.33%
libosip2	1,936.91	54.79	97.17%
lighttpd	316.57	59.79	81.11%
nano	308.87	182.42	40.94%
readline	111.49	0.63	99.43%
screen	189.64	5.46	97.12%
sqlite	1,739.91	151.82	91.27%
tar	237.84	21.12	91.12%
Average	13,680.98	1,090.59	92.03%

inspector and the categorized inspector respectively. And the last column (*Dec.*) is the reduction ratio of the execution time.

According to the table, our categorization method can decrease the execution time up to more than 90% under most circumstances. And the average reduction is 92.03% among the benchmark instances. However, there are three exceptions: `curl`, `gzip` and `nano`. The average reduction of these three benchmark instances is about 39.92%.

When we refer to table II, we can find that there are too many infeasible bug reports which generate a lot of infeasible categories that lag the inspector. According to the table, `curl` has about 99.83% of infeasible bug reports, `gzip` has about 99.96% and `nano` has about 99.55% ($1 - Feas./Rep.$ in table II). Therefore, by analogy with the result of inspection reduction, we can also conclude that the infeasible categories will affect the efficiency of the inspector. To acquire further speedup on these three instances, we need the help of the parallel inspector.

E. Efficiency of the Parallel Inspector

We mainly consider the parallel inspector in this part, which answers **RQ 3**. We evaluate the performances of the parallel inspector with different working thread numbers (2, 4, 8 and 16), and measure the corresponding execution time of inspecting all the categories. To compute the speedup of each configuration, we also compare the execution time with that of the sequential inspector.

Figure 6 presents the trend of the execution time of different configurations. Each curve in the figure represents an instance in our benchmark, together with the total value. The x axis represents the different configurations (*1 thread* for sequential execution), and the y axis indicates the execution time compared with the sequential execution.

According to the figure, the overall trend is that with the number of working threads increasing, the execution time decreases. The only exception is the benchmark instance `readline`, whose time cost of 16 threads is longer than that of 8 threads. The reason is that the total execution time is too short (about 0.63s under one thread according to table V) to be accurately measured. We manually examine all the original execution time under 4, 8 and 16 threads, and find that the difference between these data is very small. The average is 0.26 *sec.* and the standard deviation based on the entire population is 0.06. Therefore, we believe that this issue will not affect our conclusion.

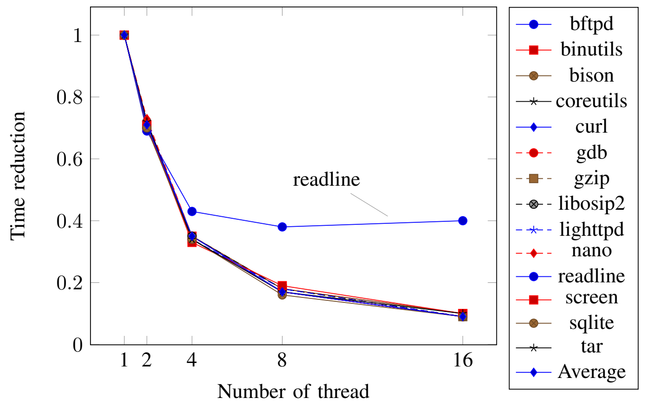


Figure 6: Execution time comparison of different configurations

Table VI presents the speedup of different configurations. We set the time of sequential execution as the baseline and compare the time under different working thread number configurations with it. The first column is the name of benchmark instances, and the following columns are the speedups of 2, 4, 8 and 16 working threads.

According to the table, the maximum speedup is 11.71 under 16 threads from benchmark instance `libosip2` and 6.09 of 8 threads from benchmark instance `sqlite`; while the minimum speedup (except benchmark instance `readline`) is 9.62 under 16 threads and 5.25 under 8 threads. The average speedup of different configurations is

Table VI: Speedup of different configurations.

Program	C-2	C-4	C-8	C-16
bftpd	1.38	2.95	5.73	10.74
binutils	1.42	2.82	5.60	9.62
bison	1.39	2.83	5.62	10.34
coreutils	1.41	2.85	5.72	10.62
curl	1.39	2.82	5.72	10.55
gdb	1.41	2.95	5.83	10.85
gzip	1.42	2.92	5.48	10.76
libosip2	1.42	2.87	5.55	11.71
lighttpd	1.41	2.93	5.66	10.73
nano	1.37	2.83	5.75	10.57
readline	1.46	2.33	2.67	2.47
screen	1.41	3.07	5.25	10.41
sqlite	1.43	2.92	6.09	11.40
tar	1.38	2.95	5.78	10.29
Average	1.40	2.86	5.74	10.59

about 0.70 times the number of threads, which indicates the average workload of threads is about 70%.

F. Threats to Validity

1) Benchmark Selection: The validity of our experiments may be subject to the threat that we selected only open source software written in C among the common fields of application. Compared with commercial software systems, the complexity of the software may sharply increase. Hence, the suspect reports extracted from these software systems will be greatly different among the aspects of complexity (path length, number of variables and clauses), the error pattern (distribution of bug types, format and code style of trigger expressions) and the number (total number of reports, ratio of false-positives). So will the software written in other languages for other fields.

2) Static Analyzer: The selection of static analyzers is also a threat to validity, since we only tested on the reports extracted from `Canalyze`. As the type and the ability of bug detection among different analyzers may also be different, the suspect reports extracted by other tools may also have differences in the aspect of characteristics, which can also threaten the performance and validity of our evaluations.

Apart from `Canalyze`, the Clang Static Analyzer (CSA), which is very similar to `Canalyze` and widely used and researched in both industry and academic, can also be used to extract bug reports in the evaluation. As interval arithmetic is also employed in the CSA to roughly prune infeasible paths, which is the same with `Canalyze`, we believe a similar conclusion can also be drawn with the uninspected reports extracted from the CSA and other LC & PI based static analyzers.

3) Sorting Strategy Evaluation: In this paper, we evaluate the sorting strategies with all our benchmark instances, and the recommended sorting strategy is used to evaluate the categorized inspector and the parallel inspector. Although our benchmark has included most of the common fields of application, which may affect the characteristics of the extracted suspect reports, the recommended sorting strategy may fail to reduce inspection attempts in some specific fields or commercial software systems.

V. Related Works

Our work is mainly related to the LC & PI based analysis method, path redundancy elimination, suspect report categorization, and parallel analysis methods. In this section, we mainly present the related works in the above aspects.

A. The LC & PI Based Analysis Method

As far as we know, the core idea of the LC & PI based analysis method was first introduced in `Canalyze` [9] with the concept of *Hybrid Solver*, which are also extended to its successors. The *Hybrid Solver* is a combination of its *Range Solver*, which is an interval arithmetic based light-weight constraint checker, and its *SMT Solver*, which is an integrated `boolector` [16] SMT solver. When using the *Hybrid Solver*, the *Range Solver* is used only during program exploration, and the suspect reports are inspected with its *SMT Solver* after the exploration.

A similar process called *SMT Refutation* has been added to Clang Static Analyzer (CSA) by Gadelha *et al.* [7]. In their work, they use the built-in constraint solver in CSA to explore the program paths, and refute the false positives by encoding the PC with SMT-Lib and invoking outside SMT solvers to inspect the reports. In their work, they integrated a lot of different state-of-the-art SMT solvers without comparing the performance of these solvers, which will confuse the users when choosing a proper SMT solver. While in our work, we only provide a generic SMT solver `Z3` [15] to inspect suspect reports.

Similar to LC & PI based analysis method, S. Ding *et al.* [17] introduced a code pattern based method to filter out infeasible paths before symbolic execution, which can reduce the number of paths inspected by the symbolic execution process. However, their light-weight pruning method is based on the syntax structures to detect unsatisfiable predicates, which may introduce false negatives due to lack of full context constraints.

B. Path Redundancy Elimination

Apart from the LC & PI based analysis method, path redundancy elimination methods are also widely used for path-sensitive analysis methods. They are originally used to solve the path-explosion problem. However, duplicated visits to buggy program points can be eliminated as redundancy. Therefore, these methods are also effective for the problem.

There have been a lot of researches [11], [18], [19] trying to eliminate redundant program paths. Q. Yi *et al.* [11] use the post-conditions to represent path suffixes and conjunct the negated post-conditions to make the path constraints of redundant suffix unsatisfiable. Since their method can only avoid redundant visits to the same branch of a condition, there will also be redundant reports generated for inputs like our motivating example presented in figure 2.

H. Wang *et al.* [18] use a dependence based symbolic value to represent path constraints, and predict the redundant paths with these dependence values. Different from [11], they use relevant path conditions based on dependences of expressions, rather than path constraints and post-constraints, to prune the program paths, and their method is more radical than [11]. D. Qi *et al.* [19] partition the program paths with inputs and outputs. If two paths generate the same output expressions, they are seen as equivalent. These two methods are much similar to our categorization method, which is based on the bug type and trigger expressions. And Junker *et al.* [8] refute infeasible paths by extracting related infeasible sub-paths and constructing observer automatons to recognize redundant infeasible program paths in model checking. However, their method is much more like a monitor to refute infeasible paths with the same sub-path, rather than a redundant elimination.

Besides, [20] and [21] also tried to use summarization techniques to reduce the search space. However, the differences are that the abstractions are built for functions to eliminate unconcerned path in sub-functions in [20], while [21] summarizes program paths and eliminates the visits on redundant paths. Compared with our work, [21] is more like our categorization strategy.

C. Suspect Report Categorization

There have been a lot of studies about bug/issue report categorization, and T. Muske *et al.* summarized eight different bug/issue report categorization methods in their summary [10]. However, the core targets of these works are different from our work. Our categorization method is used to automatically eliminate inspections on redundant suspect reports extracted from an analyzer, and expedite the post inspector. While the related works categorize the reports of defects, issues, feature enhancements or other requests, to simplify the maintenance process for developers.

Z.P. Fry *et al.* [22] formally define a bug report as a quintuple composed with the information that can be extracted from the program, including the nested functions, context, source files, bug type and so on. And using a natural language processing method to categorize the bug reports. Similar to our method, they also represent a bug report only with the information available during the analysis. However, their method requires more information and much more work to get the result. W. Le *et al.* [13] present a method to compute the correlations between bug reports, and categorize the reports with their correlations. However, their method requires an additional analysis on the program, which is repetitive to a static analyzer.

Another commonly used method to categorize reports is the Orthogonal Defect Classification (ODC) [23] concept introduced by IBM. The key idea is to extract characteristics of the development process from the reports, which is different from our method that uses the information from the report only. Many works based on this key idea have been published. F. Thung *et al.* [24] proposed a text mining method to analyze the content of the reports together with related code patches, and categorize these reports with text features.

It can also be seen as another kind of categorization to assign a bug report to a proper developer to fix it. Neelofar *et al.* [25] proposed a method that uses the Multinomial Naive Bayes text classifier to tag the bugs with their summary, while A. Tamrawi *et al.* [26] introduced a fuzzy set and cache-based modeling method. I. Chawla *et al.* [27] also presented a categorization method based on the fuzzy set theory. However, its target is to tag an issue report as a bug, a feature enhancement or a request, to improve its quality.

D. Parallel and Distributed Analysis Methods

As our improvements are also an attempt on distributed LC & PI based static analysis method, our work is also related to the parallel or distributed static analysis methods. Parallel techniques have been widely used in optimizing static analysis methods for decades [28], [29]. R. Kramer *et al.* [28] introduced a method that converts the cyclic structures (loops) of a Control Flow Graph (CFG) into acyclic structures, and then parallelizes the data flow analysis on the independent paths of the acyclic CFG. S. Bucur *et al.* [29] extends `KLEE` [4] to large shared-nothing clusters, which is the first symbolic execution engine on large clusters.

Similar to our parallel inspector, parallel SMT solving for path-sensitive analysis methods is also frequently used. Rakadjiev *et al.* [30] replace the SMT solver in `KLEE` with parallel SMT solver clusters, to expedite the engine by speeding up the constraint solving process. Different from using parallel solvers, Aigner *et al.* [31] execute a set of constraint solvers simultaneously for one query, and the fastest one that provides the result is taken. And an extension to `KLEE` has also been available in [32].

VI. Conclusion and Future Work

In this paper, we systematically improve the post-inspector in the aspect of eliminating inspections on redundant reports, which makes it an efficient, usable, complete and more essential component in the entire static analysis framework. And with our improvements, the post-inspector can be executed as a service for a lot of analyzer instances on an *inspector node* in an analyzer cluster, which can be seen as a first step in attempting to distribute the LC & PI static analysis framework.

In the future, we will continue separating and refining other components of the analysis framework and trying to build a usable distributed system of this analysis framework. Besides, some other strategies are needed to reduce unnecessary inspections on the circumstances when our categorization method can provide very limited reduction. And our categorization method also needs to be further evaluated and improved to eliminate possible false-negatives and wrongly categorized reports.

Acknowledgment

This work is supported in part by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (CAS), Grant No. *QYZDJ-SSW-JSC036*.

References

[1] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," *SIGPLAN Not.*, vol. 37, no. 5, pp. 57–68, 2002.

[2] Y. Xie, A. Chou, and D. Engler, "Archer: Using symbolic, path-sensitive analysis to detect memory access errors," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 327–336, 2003.

[3] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[4] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[5] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005.

[6] T. Kremenek, "Finding software bugs with the clang static analyzer," *Apple Inc*, 2008.

[7] M. R. Gadelha, E. Steffinlongo, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "SMT-based refutation of spurious bug reports in the clang static analyzer," *arXiv preprint arXiv:1810.12041*, 2018.

[8] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "SMT-based false positive elimination in static program analysis," in *International Conference on Formal Engineering Methods*, 2012, pp. 316–331.

[9] Z. Xu, J. Zhang, Z. Xu, and J. Wang, "Canalyze: a static bug-finding tool for c programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 425–428.

[10] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation*, 2016, pp. 157–166.

[11] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Eliminating path redundancy via postconditioned symbolic execution," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 25–43, 2018.

[12] A. Podelski, M. Schäf, and T. Wies, "Classifying bugs with interpolants," in *International Conference on Tests and Proofs*, 2016, pp. 151–168.

[13] W. Le and M. L. Soffa, "Path-based fault correlations," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 307–316.

[14] I. P. Gent and T. Walsh, "The SAT phase transition," in *ECAI*, 1994, pp. 105–109.

[15] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[16] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 174–177.

[17] S. Ding, H. Zhang, and H. Beng Kuan Tan, "Detecting infeasible branches based on code patterns," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, 2014, pp. 74–83.

[18] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.

[19] D. Qi, H. D. T. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 32:1–32:41, 2013.

[20] Z. Xu, J. Zhang, and Z. Xu, "Melton: a practical and precise memory leak detection tool for c programs," *Frontiers of Computer Science*, vol. 9, no. 1, pp. 34–54, 2015.

[21] S. Anand, C. S. Păsăreanu, and W. Visser, "Symbolic execution with abstraction," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 1, pp. 53–67, 2009.

[22] Z. P. Fry *et al.*, "Clustering static analysis defect reports to reduce maintenance costs," in *2013 20th Working Conference on Reverse Engineering*, 2013, pp. 282–291.

[23] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on software Engineering*, no. 11, pp. 943–956, 1992.

[24] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, 2012, pp. 205–214.

[25] Neelofar, M. Y. Javed, and H. Mohsin, "An automated approach for software bug classification," in *Proceedings of the 2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, 2012, pp. 414–419.

[26] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 365–375.

[27] I. Chawla and S. K. Singh, "An automated approach for bug categorization using fuzzy logic," in *Proceedings of the 8th India Software Engineering Conference*, 2015, pp. 90–99.

[28] R. Kramer, R. Gupta, and M. L. Soffa, "The combining dag: a technique for parallel data flow analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 805–813, 1994.

[29] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 183–198.

[30] E. Rakadjiev, T. Shimosawa, H. Mine, and S. Oshima, "Parallel SMT solving and concurrent symbolic execution," in *2015 IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 17–26.

[31] M. Aigner, A. Biere, C. M. Kirsch, A. Niemetz, and M. Preiner, "Analysis of portfolio-style parallel SAT solving on current multi-core architectures." *POS@ SAT*, vol. 29, pp. 28–40, 2013.

[32] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *International Conference on Computer Aided Verification*, 2013, pp. 53–68.