Large-Scale Analysis of Non-Termination Bugs in Real-World OSS Projects

Xiuhan Shi
College of Intelligence and Computing, Tianjin University
Tianjin, China
shixiuhan@tju.edu.cn

Xiaofei Xie
Singapore Management University
Singapore, Singapore
xfxie@smu.edu.sg

Yi Li
School of Computer Science and Engineering, Nanyang Technological University
Singapore, Singapore
yi_li@ntu.edu.sg

Yao Zhang
College of Intelligence and Computing, Tianjin University
Tianjin, China
zzyy@tju.edu.cn

Sen Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

Xiaohong Li*
College of Intelligence and Computing, Tianjin University
Tianjin, China
xiaohongli@tju.edu.cn

ABSTRACT

Termination is a crucial program property. Non-termination bugs can be subtle to detect and may remain hidden for long before they take effect. Many real-world programs still suffer from vast consequences (*e.g.*, no response) caused by non-termination bugs. As a classic problem, termination proving has been studied for many years. Many termination checking tools and techniques have been developed and demonstrated effectiveness on existing well-established benchmarks. However, the capability of these tools in finding practical non-termination bugs has yet to be tested on real-world projects. To fill in this gap, in this paper, we conducted the first large-scale empirical study of non-termination bugs in real-world OSS projects. Specifically, we first devoted substantial manual efforts in collecting and analyzing 445 non-termination bugs from 3,142 GitHub commits and provided a systematic classification of the bugs based on their root causes. We constructed a new benchmark set characterizing the real-world bugs with simplified programs, including a non-termination dataset with 56 real and reproducible non-termination bugs and a termination dataset with 58 fixed programs. With the constructed benchmark, we evaluated five state-of-the-art termination analysis tools. The results show that the capabilities of the tested tools to make correct verdicts have obviously dropped compared with the existing benchmarks. Meanwhile, we identified the challenges and limitations that these tools face by analyzing the root causes of their unhandled bugs. Finally, we summarized the challenges and future research directions for detecting non-termination bugs in real-world projects.

*Sen Chen and Xiaohong Li are the corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549129

CCS CONCEPTS

• **Theory of computation** → **Program analysis**; • **Software and its engineering** → *Software post-development issues.*

KEYWORDS

Non-termination Bug, Benchmarking, Empirical Study

ACM Reference Format:
Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. 2022. Large-Scale Analysis of Non-Termination Bugs in Real-World OSS Projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549129

1 INTRODUCTION

Termination concerns the *liveness* of a program, which is crucial to software quality. A program is non-terminating if there exist some inputs that cause the program to execute indefinitely. Non-termination of programs may have vast consequences, especially when employed in safety-critical environments, *e.g.*, aerospace software. For example, software with non-termination bugs can become unresponsive [12], leading to degraded user experiences and sometimes denial-of-service attacks [15]. The current attempts to this problem focus on proving termination [4, 7, 10, 20, 24, 46]. Yet, determining program termination is shown to be an undecidable problem [23], and a failure to prove termination does not indicate that the program can always terminate. On the other hand, it is also challenging to show that a program is non-terminating. The challenge lies in the fact that the violation witnesses of a liveness property are infinite traces, therefore, one cannot come up with a finite oracle as in the case of a *safety* property.

In recent years, many advanced algorithms and techniques [5, 6, 38, 56] have been proposed to either prove termination or demonstrate non-termination of programs. Generally, they are able to achieve good performance on standard benchmarks, such as SV-COMP [21] and TermCOMP [52]. These benchmarks often include manually crafted programs, with significantly simplified language features and execution environments, to make the evaluation of various algorithms [14, 39, 56] easier. This brings concerns to whether the evaluation results can truly reflect the performance of the techniques on real-world non-termination bugs. For example, our study shows that while UAutomizer [34] correctly handles 71.5% (1,581/2,212) programs in SV-COMP 2021, but it cannot be directly applied on real OSS projects, and it only successfully handles 47% of the real-world non-terminating programs after necessary simplifications have been made. This indicates that the existing benchmarks may not be ideal evaluation subjects when the practical values of the non-termination checking techniques are concerned.

There have been a number of studies done on *real-world software bugs* [26, 28, 30, 36, 42, 59], which play significant roles in raising awareness and bringing new insights to software quality assurance [35, 53, 54]. Similarly, we believe that a deep analysis of real-world non-termination bugs will provide useful insights to developers and guide the development of new non-termination detection techniques. To the best of our knowledge, there is still no such study on non-termination bugs, which motivates our work. In particular, we would like to find out the answers to the following questions. *How common do non-termination bugs appear in real-world programs? What are the root causes of these bugs? How difficult is it to find these bugs? How effective are the state-of-the-art techniques in detecting non-termination bugs in real-world OSS projects?*

In this paper, we aim to bridge this gap by conducting a large-scale empirical study of non-termination bugs in real-world OSS projects. We face three main challenges in this study. **First,** there is no existing dataset on the non-termination bugs from real-world OSS projects. It is difficult to establish reasonable criteria and collect representative non-termination bugs to build the dataset. **Second,** the root cause analysis of non-termination bugs is difficult. Due to the complexity of the real-world program logic (*e.g.*, complex data structure, nested loops, and recursive function) and the lack of a test oracle, it is non-trivial to understand whether they are real non-termination bugs and why the programs do not terminate. **Third,** the state-of-the-art tools cannot be directly applied to real-world OSS projects since many complex features are not well supported, such as dependencies and complex data structures, making it difficult to evaluate these tools on OSS projects.

To overcome these challenges, we first collect 3,142 commits from 1,600 C/C++ projects, which are related to non-termination bugs. Note that we selected C/C++ in our work because the popular termination analysis tools (e.g., AProVE, CBMC, and UAutomizer) only support C/C++ programs. We dedicated substantial efforts to manually investigate these commits and finally identified 445 non-termination bugs from 199 projects. Through further in-depth analysis on the root causes of these bugs, we systematically built a hierarchical taxonomy containing 24 categories. To evaluate the state-of-the-art tools (*i.e.*, UAutomizer [34], CPAChecker [9], 2LS [48], AProVE [31], and T2 [11]) on real-world programs, we built a new benchmark including 56 non-terminating programs and 58 fixed versions, which are extracted from the real-world non-termination bugs. We evaluated their effectiveness and summarized the common reasons for their failures. In general, we aim to answer the following research questions:

- **RQ1:** What are the root causes of non-termination bugs in real-world OSS projects?
- **RQ2:** How effective are the state-of-the-art tools in proving the non-termination of real-world programs?
- **RQ3:** What are the potential root causes for the failures of the studied tools?

By answering these questions, we characterize the real-world non-termination bugs and provide useful insights for developers and researchers. For example, our results reveal that infinite loops can be caused by 10 types of common logical faults (*e.g.*, missing iterator update, using erroneous condition) as well as 6 other bug types related to general programming features (*e.g.*, overflow, type conversion). Infinite recursions can be caused by three incorrect recursion designs (*e.g.*, incorrect return) and 5 types of unexpected recursion (*e.g.*, misusing method overloading). Our study on the existing tools shows that they are almost completely inapplicable to real projects. Compared to the existing benchmarks, the performance of existing tools drop significantly on the extracted benchmarks, indicating that they are still far from effective in discovering real-world non-termination bugs. We also discuss and summarize key challenges that should be addressed in future research. More details can be found on our website.[1]

In summary, this paper makes the following contributions:

- To the best of our knowledge, we conducted the first comprehensive study on analyzing root causes of real-world non-termination bugs. We constructed a systematic taxonomy of 24 bug categories and highlight their characteristics including the distributions, root causes, fix strategies, *etc.*
- We constructed a new benchmark that is extracted from different categories of real-world non-termination bugs, including the non-termination versions and the corresponding fixed versions. The benchmark is public available and will be expanded continuously.
- We evaluated the state-of-the-art termination analysis tools on the extracted benchmarks and identified their weaknesses. We summarized the main challenges and provided future research directions for detecting non-termination bugs in real-world projects.

2 RELATED WORK

2.1 Termination Analysis

The general approach to prove termination is to search for *ranking functions* [4, 7, 10, 20, 24, 40, 46, 55, 58]. which map a program state to an element of some well-funded ordered set. Most termination analysis approaches rely on static analysis and constraint solving to synthesize ranking functions. Podelski *et al.* [46] proposed an automated method for proving the termination of an unnested loop by synthesizing linear ranking functions. Cousot *et al.* [24] expressed program semantics in polynomial form and automatized the Floyd/-Naur/Hoare proof method to verify semialgebraic programs. Chen *et al.* [20] reduced non-linear ranking function inference for polynomial programs to semi-algebraic system solving problems. Xie *et al.* [57] propose the path dependency automaton to capture the

[1]https://sites.google.com/view/non-termbug/home

dependencies among the multiple paths in a loop. Ultimate Automizer [34] covers the whole set of program executions by taking the union of the languages of several automata, each of which is proved to be terminating by exhibiting an appropriate ranking function. For more complex programs, more complex ranking functions [4, 7, 10] are proposed to prove termination.

Although a lot of termination-proving techniques are proposed, most of them are incomplete. The failure of proving termination does not indicate that the program is non-terminating. Hence, this paper mainly focuses on detecting non-termination bugs, which cannot be directly solved by the proposed techniques. We also aim to study real-world non-termination bugs and evaluate the state-of-the-art tools on proving non-termination.

2.2 Non-Termination Analysis

The general approach to prove non-termination is to search for *recurrent sets*. Gupta *et al.* [33] developed a non-termination prover "TNT", which proves non-termination by dynamically enumerating lasso-shaped candidate paths to search for counterexamples to termination and search for a recurrent set for each lasso. Giesl *et al.* [31] used constraint solving to find a recurrent set in a given loop to prove non-termination of the loop. Cook *et al.* [22] proved non-termination by using abstract interpretation to over-approximate nonlinear programs and inferring linear recurrent sets. Le *et al.* [39] proved non-termination by iteratively collecting executions traces and dynamically learning conditions to refine recurrent sets. For proving the non-termination of non-deterministic programs, closed recurrent sets are proposed, which is a stronger notion than recurrent sets. Chen *et al.* [14] used a safety prover to eliminate terminating paths iteratively until it finds a closed recurrent set in the remaining paths. Larraz *et al.* [38] proved non-termination by Max-SMT-based invariant generation.

In addition, there are other techniques that tend to identify infinite states [12, 13, 16, 45]. Carbin *et al.* [12] used dynamical detection to record the program state at the start of each loop iteration, and proved non-termination when two consecutive loop iterations produced the same state. Menendez *et al.* [45] proposed a methodology to detect non-termination issues with a suite of peephole optimizations. Chatterjee *et al.* [13] proved the non-termination of non-deterministic integer programs by relying on a purely syntactic reversal of the program's transition system. Xie *et al.* [56] reduced the non-termination analysis to a reachability problem, *i.e.*, to find a counterexample that reaches an infinite state.

Some efforts have been recently made to take low-level programming features (*e.g.*, overflow) into consideration, which could be helpful for analyzing real-world projects. Schrammel *et al.* [48] presented a modular termination analysis for C programs using template-based inter-procedural summarization towards analyzing real-world software with bit-precise termination arguments that were synthesized over lexicographic linear ranking function templates. Maurica *et al.* [43] transposed the termination analysis of floating-point loops into termination analysis of rational loops through the use of an innovative rational approximation, which covers overflow issues.

The existing techniques are mainly evaluated on standard simplified benchmarks which fail to represent most of the real-world non-termination bugs. This paper aims to study the real-world non-termination bugs and evaluate the practical value of the state-of-the-art techniques while identifying potential future research directions.

2.3 Existing Studies on Real-World Bugs

Researchers have made great efforts in exploring the root causes of various bugs and corresponding fix strategies in real-world projects, including Android bugs [17–19, 27, 28, 41, 44, 50], OSS fuzz-bugs [26], buffer overflow bugs [59], deep learning bugs [36, 47], autonomous vehicle bugs [30], *etc.*. These studies help developers understand the practical relevance of different bugs and provide insights through examples for researchers to develop more advanced detection techniques. However, almost all of them focus on the violation of safety properties, while the violation of liveness properties (*i.e.*, non-termination bugs) are not touched. To the best of our knowledge, our work is the first large-scale empirical study on non-termination bugs in real-world projects. Additionally, we constructed a new benchmark by simplifying the non-termination bugs from real-world projects.

3 DATA PREPARATION

3.1 Data Collection

In this paper, we mainly study the non-termination bugs in C/C++ projects. We first randomly collected 1,600 C/C++ projects from GitHub by GitHub APIs [32]. These projects are collected from two considerations: 1) most of the code are C/C++ programs and 2) they have different numbers of stars representing different popularity. The collected projects cover open source projects designed for various purposes (*e.g.*, database, operating system, media tools, and game). Then we chose five keywords, *i.e.*, "infinite loop", "endless loop", "long loop", "infinite recursion", and "deep recursion", to identify potential non-termination bugs from the commit messages of these projects. Finally, we obtained 3,142 commit messages that cover 466 out of the 1,600 projects. Table 1 shows the detailed results from each keyword.

We spent five person-months investigating the collected commits. Specifically, we manually analyzed the code snippets corresponding to the 3,142 commit messages to understand the root causes of the non-termination. Note that the key challenge is that there is no oracle for non-termination. Due to the high complexity of real-world code, it is difficult to understand some code in terms of whether and why they are non-terminating. We filtered some commits that are difficult to analyze: 1) their corresponding code snippets do not contain a clear repeated procedure (*e.g.*, loop structure and recursion); 2) the repeated procedures are too complex to understand (*e.g.*, loops with hundreds of lines of code or commits with large changes), and 3) the non-termination can be affected by non-C/C++ code. Finally, we kept 445 commits (*i.e.*, non-termination bugs) from 199 real-world projects, which cover the different numbers of stars and involve various types of projects. These commits mainly belong to the following two categories: *infinite loop* (318) and *infinite recursion* (127). To confirm these bugs, each of them is analyzed, discussed, and confirmed by at least two authors. For cases that they could not decide, all authors participated in the

Table 1: Details of the collected commits.

Keywords	#Projects	#Commits
infinite loop	313	2,400
endless loop	127	445
long loop	9	27
deep recursion	14	17
infinite recursion	97	253
Total	466	3,142

discussion and confirmation. Note that for the commits that are filtered, we cannot conclude that they are terminating either.

Finding 1: *A large portion (29.1%) of the collected OSS projects have non-termination issues. In particular, the confirmed non-termination bugs mainly belong to infinite loops (71.5%) and infinite recursion (28.5%).*

3.2 Manual Labelling

Based on the 445 non-termination bugs, we performed a deep analysis on the root causes. We adopted an open card sort strategy [29] to construct a new hierarchical taxonomy of the root causes. Three of the authors mainly participated in the taxonomy construction. Each participant had more than two years of experience in the research of termination analysis.

Two authors first constructed the leaf categories for all the bugs. Specifically, in the first round, the two participants individually analyzed the commit messages, the original code, and the changed code to define the root causes. In the second round, they had a discussion to iteratively unify the two versions of the leaf categories. If there were disagreements between them, other authors joined the discussion until a consensus was reached. In the third round, we randomly selected 20% of the 445 bugs, which were labeled by the third participant based on the created leaf categories. For results from the third participant that contradicted the labels of the first two authors, all authors discussed them until the contradiction was resolved by updating the leaf categories or re-labeling the conflicting cases. Finally, all authors discussed the leaf categories and grouped leaf categories into high-level categories to construct the hierarchical taxonomy of the root causes. For example, the common characteristic of "*Signed Overflow Error*" and "*Unsigned Wraparound Error*" is overflow, therefore, we grouped these two leaf categories into a high-level category "*Overflow*".

Note that, as infinite loops and infinite recursion have very different characteristics, we adopted the same methodology described above to establish the taxonomies for them separately.

4 ROOT CAUSES OF INFINITE LOOPS

Figure 1 shows the hierarchical taxonomy of infinite loops including four levels of categories. Our taxonomy of infinite loops consists of 10 inner categories (marked in grey color) and 16 leaf categories (marked in white color). To measure the frequency of bugs appearing in each category, we counted *the number of bugs* in each category and *the number of projects* where the bugs are located, shown in the upper the right corner of each category in Figure 1.

In general, the root causes of infinite loops can be divided into two categories, namely, *logical errors* (Category 1) that are more related to the loop structure itself (*e.g.*, loop condition and loop iterator variable), and *general programming errors* (Category 2) that may affect the loop execution (*e.g.*, overflow). Specifically, most of the infinite loops (85.8%) are caused by logical errors that are rooted at the improper design of loops (*e.g.*, incorrect usage of loop iterator variable or incomplete loop condition checking), which causes the loop to be stuck in a state. To our surprise, many infinite loops (14.2%) are caused by general programming errors such as improper type conversions, even if the logic of the loops is correct. However, due to other programming errors such as integer overflow and implicit casting, these loops can be executed infinitely. Next, we provide detail for each category.

Finding 2: *The main reasons for infinite loops are logical errors (85.8% of the bugs covering 92.7% of the projects), indicating the difficulty of designing loops correctly. In addition, common programming mistakes can make logically-correct loops execute infinitely (14.2% of the bugs covering 20.4% of the projects), which may challenge the methods that attempt to prove (non)termination based only on program logic.*

4.1 Logical Error (Category 1)

From the program-logic perspective, the behavior of a loop iteration depends on the *loop condition*, the *loop iterator variables*, and the *control statements* (*e.g.*, break). Specifically, the loop continues to iterate while the condition is true, a loop iterator variable (or *loop iterator*) serves as an index of the loop and may affect the value of loop conditions, and control statements may change the flow of loop execution. Logical errors are divided into three categories related to these three loop elements (*i.e.*, Category 1.1, 1.2, and 1.3). In total, there are 10 specific leaf categories under Category 1.

Finding 3: *Most of the logical errors are caused by incorrect loop iterators (54.6%) and incorrect loop conditions (40.3%). A small number of errors are due to incorrect control statements (5.1%). Furthermore, 10 different root cases were identified, indicating the diversity of logical errors.*

4.1.1 Loop Iterator Error (Category 1.1). During loop execution, loop variables are updated iteratively. A *Loop iterator error* refers to an incorrect update to loop iterators, making the loop condition to be always true. Updates to loop iterators usually involve only a few lines of code but are error-prone when their cascading effects span multiple iterations. We found 149 loop iterator bugs rooted in different causes.

Category 1.1.1: Misusing Same Loop Iterator in Nested Loops. We observed that, in nested loops, developers may confuse the loop iterators of the inner loops with those of the outer loops. There are six infinite loops (4.0%) that are caused by reusing the *same* iterator in nested loops. For example, if the inner loop incorrectly uses the loop iterator of the outer loop (*e.g.*, setting it to zero), the outer loop never terminates.

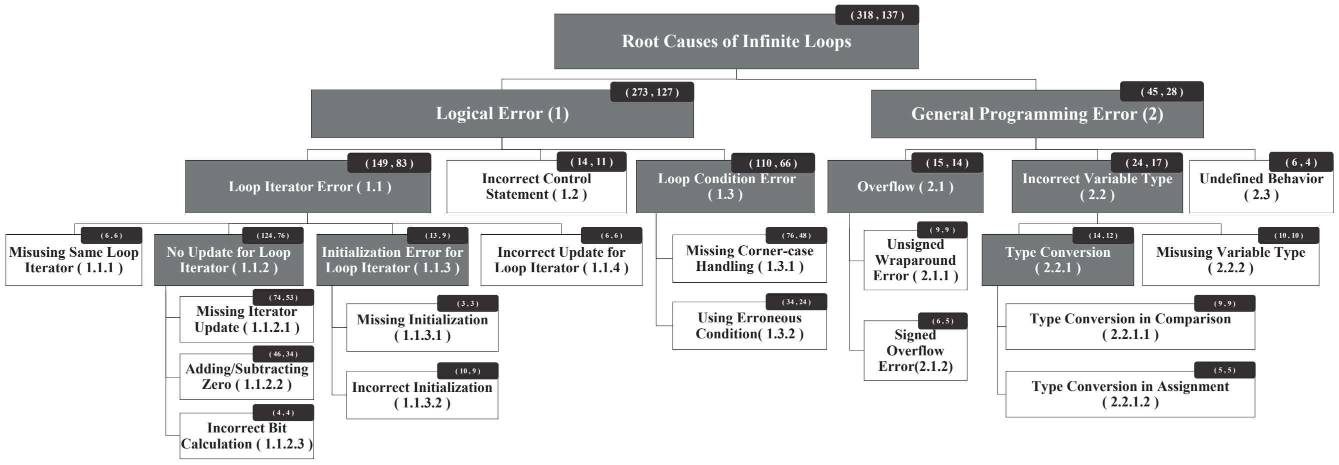
(318 , 137)
Root Causes of Infinite Loops
(273 , 127)
Logical Error (1)
(45 , 28)
General Programming Error (2)
(149 , 83)
Loop Iterator Error (1.1)
(14 , 11)
Incorrect Control Statement (1.2)
(110 , 66)
Loop Condition Error (1.3)
(15 , 14)
Overflow (2.1)
(24 , 17)
Incorrect Variable Type (2.2)
(6 , 4)
Undefined Behavior (2.3)
(6 , 6)
Misusing Same Loop Iterator (1.1.1)
(124 , 76)
No Update for Loop Iterator (1.1.2)
(13 , 9)
Initialization Error for Loop Iterator (1.1.3)
(6 , 6)
Incorrect Update for Loop Iterator (1.1.4)
(76 , 48)
Missing Corner-case Handling (1.3.1)
(34 , 24)
Using Erroneous Condition(1.3.2)
(9 , 9)
Unsigned Wraparound Error (2.1.1)
(6 , 5)
Signed Overflow Error(2.1.2)
(14 , 12)
Type Conversion (2.2.1)
(10 , 10)
Misusing Variable Type (2.2.2)
(74 , 53)
Missing Iterator Update (1.1.2.1)
(46 , 34)
Adding/Subtracting Zero (1.1.2.2)
(4 , 4)
Incorrect Bit Calculation (1.1.2.3)
(3 , 3)
Missing Initialization (1.1.3.1)
(10 , 9)
Incorrect Initialization (1.1.3.2)
(9 , 9)
Type Conversion in Comparison (2.2.1.1)
(5 , 5)
Type Conversion in Assignment (2.2.1.2)

Figure 1: Taxonomy of Infinite Loop.

Category 1.1.2: No Update to Loop Iterator. A large portion of errors are caused when the loop iterators remain constant under some conditions, making the loop stuck with no progress. Specifically, to our surprise, there are 74 cases (59.7% of Category 1.1.2) that are due to missing iterator updates (Category 1.1.2.1). Our analysis shows that the iterator updating statements are missed in 54 cases which are often due to careless mistakes of developers. The remaining 20 cases are because the iterator updates are placed incorrectly after the *continue* statements, not being executed as a result. Another group of errors (37.1%) is caused when the change made to the iterator value is effectively zero (Category 1.1.2.2). For example, in $i+=x$, the loop iterator i remains unchanged when x is zero. Bit manipulations (Category 1.1.2.3), such as '&' (AND), '|' (OR), and '≪' (shift left), can also lead to such errors. Figure 2 shows an example of *Bit Operation* from the project "brltty".[2] In this case, the loop terminates only if *wc* becomes zero. However, since *wc* is an extended *signed* character type (*i.e.*, `wchar_t`), the loop makes no progress if *wc* is negative before entering the loop. Because of the shift-right operations, *wc* will eventually remain -1, causing a non-termination.

```
    wchar_t wc;
+ + static const wchar_t mask = (1 << ((sizeof(wchar_t) * 8) - 6)) - 1;
    do {
      *--byte = (wc & 0X3F) | 0X80;
- - } while (wc >>= 6);
+ + } while ((wc = (wc >> 6) & mask));
```

Figure 2: An example of Incorrect Bit Operation.

Finding 4: *Most (83.2%) of the loop iterator errors are due to no update to loop iterators. They are due to careless mistakes* (i.e., *forgot to update iterator, 43.5%), incorrect locations of update statements (*i.e., *update after continue statement, 16.1%), incorrect update to iterators (*i.e., *update being zero, 37.1%) and incorrect bit manipulations (3.2%).*

Category 1.1.3: Initialization Error for Loop Iterator. Variable initialization is important but error-prone. We found 13 cases (8.7%) that are caused by the incorrect initialization of loop iterators, such as *Missing Initialization* (category 1.1.3.1) and *Incorrect Initialization* (category 1.1.3.2). Uninitialized variables may lead to undefined behaviors that can cause an infinite loop. Incorrect initialization can also affect the loop execution, which mainly includes incorrect positions of initialization statements (*e.g.*, the initialization for the outer loop is incorrectly put in the inner loop) and incorrect initialization values (*e.g.*, the binary search may not terminate if the variables *low* and *high* are not initialized properly).

Category 1.1.4: Incorrect Update for Loop Iterators. Incorrect updates to loop iterators lead to non-termination. We identified six infinite loops that are caused by incorrectly updated loop iterators. Note that we distinguish this category from Category 1.1.2 because we would like to emphasize the different effects of no update and incorrect update. No update for loop iterator (*i.e.*, Category 1.1.2) causes the loop to get stuck in one state, while incorrect updates may cause the loop to get stuck in a recurrent set of states. Figure 3 shows an infinite loop[3] from the *asterisk* project. In each loop iteration, l is decreased by 2, which results in an infinite execution if l is initialized to an odd number. The update should vary based on the parity of the initial value of l.

Finding 5: *Apart from the problem of no update to loop iterators, infinite loops can also be introduced by incorrect initializations (8.7%), incorrect updates (4.0%), and incorrect reusing of loop iterators (4.0%).*

[2]Commit: 5bec3fff0bac50f4b4d4d3b02e70161a2bf38d0f

[3]Commit: 3322180d4b452e11545b70abc9b2d5af3d241361

```
static void unpacksms16(unsigned char *i,
unsigned char l,..., unsigned short *ud, ...){
    unsigned short *o = ud;
    while (l--) {
        int v = *i++;
- -     if (l--)
+ +     if (l && l--)
            v = (v << 8) + *i++;
        *o++ = v;
    }
}
```

Figure 3: An infinite loop bug of Incorrect Update of Loop Iterator.

4.1.2 *Incorrect Control Statement (Category 1.2).*

Control statements are used to direct the control flow of the loop execution. In our analysis, 14 of the logical errors (5.1%) are attributed to the incorrect control statements (*i.e.*, *break*, *goto*, and *continue*). For example, the *continue* and *break* statements can be misused; *break* statements can be placed at incorrect locations, resulting in incorrect termination condition; and *goto* statements may jump to incorrect program locations.

> ***Finding 6:*** *Incorrect control statements (14 bugs from 11 projects) can also introduce infinite loops, where incorrect break statements caused the most (50.00%) non-termination bugs.*

4.1.3 *Loop Condition Error (Category 1.3).*

Loop conditions determine whether the loop can start or terminate. A proper loop condition is critical for the correctness and termination of the loop. We observed that a large number of infinite loops (40.3%) are caused by incorrect loop conditions.

Category 1.3.1: Missing Corner-case Handling. The loop condition restricts the scope of states (*i.e.*, different values of variables) that can be reached. If the scope is not well designed, it may cause infinite execution. We observe 76 (69.1%) bugs caused by loose conditions that miss handling some corner-cases. 11 of them are caused by the general incorrect comparison operators (*e.g.*, use $i \leq 0$ rather than $i < 0$). The incorrect comparison operators can miss some boundary checking. Figure 4 shows an infinite loop caused by the missing corner-case handling from "libssh".[4] The comparison operator is not correct, which makes the loop condition always satisfied. channel_read function returns 0 if no more data can be read. The remaining 65 cases are caused by loose conditions that are more related to the specific business logic of the programs.

```
- while ((rc = channel_read(channel, buffer, sizeof(buffer), 0)) >= 0){
+ while ((rc = channel_read(channel, buffer, sizeof(buffer), 0)) > 0){
    fwrite(buffer, 1, rc, stdout);
}
```

Figure 4: An example of Missing Corner-case Handling.

[4]Commit: 1b15896e8b29561447fff9a7bcaa028179eab51b

Category 1.3.2: Using Erroneous Condition. It is worth noting that there are 34 bugs (30.9%) caused by the totally incorrect loop conditions. These conditions may use incorrect iterator variables, incorrect termination logic, or *TRUE* condition (the value is always true), indicating the difficulty of setting correct termination conditions in some loops. Figure 5 shows an infinite loop from "binutils-gdb ".[5]. It is obvious that the loop condition can always be true when cached_frame->reg_count is not zero.

```
- - for (int i = 0; cached_frame->reg_count; i++)
+ + for (int i = 0; i < cached_frame->reg_count; i++)
        xfree (cached_frame->reg[i].data);
```

Figure 5: An example of Using Erroneous Condition.

> ***Finding 7:*** *A large part of logic errors (40.3%) are due to the incorrect loop conditions. Most of them (69.1%) are affected by improper corner-case handling, which reveals that developers should be very careful on the loop conditions (*e.g.*, the boundary). What is more worrisome is that 30.9% of them are caused by completely incorrect loop conditions that usually depend on the business logic.*

4.2 General Programming Error (Category 2)

In addition to the logical errors about the loop design, we observe that general programming errors can also lead to infinite loops, which account for 14.2% infinite loops. Specifically, integer overflow, variable type casting and undefined behavior can affect the update of loop iterators, resulting in non-termination. These errors are not directly related to the loop logic. Hence, it is hard to detect them by existing termination tools (see Section 6) that mainly focus on the logic errors.

4.2.1 *Overflow (Category 2.1).*

Without good consideration for overflow, developers can misestimate the update of loop iterators during the loop execution. We observe 15 infinite loops (4.7%) that are caused by overflow including *Unsigned Wraparound Error* (Category 2.1.1) and *Signed Overflow Error* (Category 2.1.2).

Category 2.1.1: Unsigned Wraparound Error. For unsigned integer types, wraparound operations will be executed when the value of the variable is out of scope. For example, the result of the expression "UINT_MAX+1" will be 0, which is well-defined. Due to the wraparound of unsigned numbers, the loop iterators can never break the loop condition, causing infinite loops. We find 9 infinite loops due to the wraparound of unsigned numbers, which account for 2.83% of all infinite loops. Figure 6 shows an example infinite loop from "mupdf".[6] *size_t* is an unsigned type. If n is less than 16, n will be wrapped around to be another large positive value.

Category 2.1.2: Signed Overflow Error. It is well-defined that wraparound operation will be executed when overflow or underflow of unsigned integer occurs. But when overflow occurs, the results of

[5]Commit: 8455d26243aef72f7b827ec0d8367b6b7816de07
[6]Commit: ce9d4462423ac74a1dbbc4ce52c2c81cfcdda766

```
void pdf_encrypt_data(..., size_t n){
    while( n > 0 ){
        size_t len = n;
- -     n -= 16;
+ +     n -= len;
    }
}
```

Figure 6: An example of Unsigned Wraparound Error.

signed integer types maybe various. In general, wraparound operation will be executed when overflow or underflow of signed integer occurs, *i.e.*, the result of expression INT_MAX+1 is INT_MIN [25]. Similarly, *Signed Overflow Error* leads to the abnormal change of loop iterators, leading to infinite execution. We totally find 6 such bugs, accounting for 1.89% of all loop bugs.

4.2.2 *Incorrect Variable Type (Category 2.2).* The storage of different types of variables is different (*i.e.*, different ranges). Incorrect use of variable types can also cause infinite loops by changing the value of loop iterators abnormally. There are 24 bugs in this category including *Type Conversion* (Category 2.2.1) and *Misuse Variable Type* (Category 2.2.2).

```
       uint16_t s,len;
- -    for (s = seqnum; s < seqnum + len; s++) {
+ +    for (i = 0, s = seqnum; i < len; i++, s++) { // int i,len
          ... }
```

Figure 7: An example of Type Conversion in Comparison.

Category 2.2.1: Type Conversion. When the types of left values and right values do not match, the type conversion can happen in assignment statements and comparison statements, which can affect the value of loop iterators and loop conditions. There are a total number of 14 bugs caused by type conversion including 9 *Type Conversion in Comparison* bugs (Category 2.2.1.1) and 5 *Type Conversion in Assignment* bugs (Category 2.2.1.2). Figure 7 shows an infinite loop from "owntone-server".[7] This loop will get stuck when $seqnum + len$ is greater than UINT16_MAX. In this case, bit expansion will occur, and the type of the right value in the loop condition will be a large 32-bit integer, *i.e.*, it is greater than UINT16_MAX. However, the maximum value of the left value (*i.e.*, *s*) is less than or equal to UINT16_MAX, indicating that the loop condition will be always TRUE. *Type Conversion in Assignment* (Category 2.2.1.2) involves the type conversion in assignment statements. For example, the value of the loop iterator can be truncated if it is assigned to a small type so that it never breaks the loop condition.

Category 2.2.2: Misusing Variable Type. The type of a variable determines the range of its values. The incorrect type may limit the range of the loop iterator, which can lead to an infinite loop. We totally find 10 bugs caused by misusing variables, which accounts for 3.14% of infinite loops. Figure 8 shows an infinite loop bug from

```
- - u32 div1, div2;
+ + int div1, div2;
    for (div1 = 1; div1 >= 0; div1--)
        for (div2 = 7; div2 >= 0; div2--)
            ....
```

Figure 8: An example of Misusing Variable Type.

"linux_media".[8] Because $div1$ and $div2$ are unsigned variable, they never become negative. Hence, the loop condition $div1 \geq 0$ and $div2 \geq 0$ will be always TRUE.

> ***Finding 8:*** *Not like logic errors that directly affect the loop execution, general programming errors can also affect the loop execution, which may be very different as expected by developers. Specifically, overflow (4.7%) and incorrect variable type (7.5%) can implicitly change the value of loop iterators and loop conditions, which leads to infinite loops. Furthermore, such errors are very complex and subtle, each taking an average of an hour to analyze and confirm.*

4.2.3 *Undefined Behavior (Category 2.3).* During our classification, we find some potential infinite bugs caused by undefined behavior and the termination of these programs may be different in different compilers and platforms. The best-known examples of undefined behaviors [37] in programming languages come from C and C++, which have hundreds of them, including simple local operations (overflowing signed integer arithmetic). *Undefined Behavior* could make unexpected consequences (*e.g.*, Silent Breakage, Time Bombs), which depends on different compilers and platforms [25].[9] The unexpected consequences can affect the termination of loops. In our study, undefined behavior is a root cause of infinite loop, accounting for 1.9% of all loop bugs and involving 2.92% projects.

```
    uint64_t val;
    int i, bytes = 1;
- - while (val >> bytes*8) bytes++;
+ + while (val >> bytes*8 && bytes < 8) bytes++;
```

Figure 9: An example of Undefined Behavior.

Figure 9 shows a potential non-termination loop bug from "FFmpeg".[10] When the value of *bytes* is 8, $uint64\_t >> 64$ is an undefined operation leading to the undefined behavior. Its value varies in different compilers and platforms, *i.e.*, the loop can be infinitely executed if its value is parsed as non-zero. The confirmation of this category of bugs is difficult. We confirmed these bugs from 1) the commit messages that clearly point out the non-termination and 2) we reproduce them by simplifying the program. For example, we compile the loop in Figure 9 with gcc (version 7.3.0) in Ubuntu

[7]Commit: f9bfec180f91671d8ba72a01cab1781c1f5e9999

[8]Commit: 090341b0a95d1f6d762915a75c13b393366f4ab3

[9]These types of bugs are mainly confirmed from the commit messages

[10]Commit: d597655f771979c70c08f8f8ed84c1319da121e8

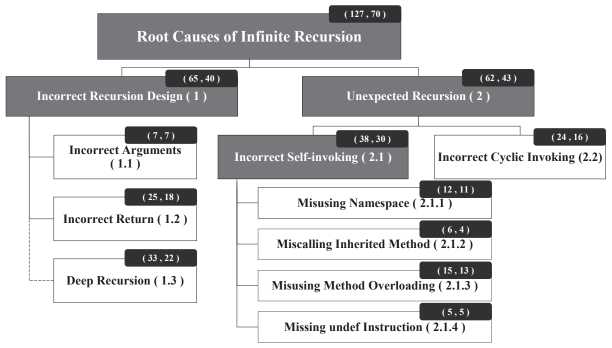
(127 , 70)
Root Causes of Infinite Recursion
(65 , 40)
Incorrect Recursion Design (1)
(62 , 43)
Unexpected Recursion (2)
(7 , 7)
Incorrect Arguments (1.1)
(38 , 30)
Incorrect Self-invoking (2.1)
(24 , 16)
Incorrect Cyclic Invoking (2.2)
(12 , 11)
Misusing Namespace (2.1.1)
(25 , 18)
Incorrect Return (1.2)
(6 , 4)
Miscalling Inherited Method (2.1.2)
(15 , 13)
Misusing Method Overloading (2.1.3)
(33 , 22)
Deep Recursion (1.3)
(5 , 5)
Missing undef Instruction (2.1.4)

Figure 10: Taxonomy of Infinite Recursion.

4.15.0. The value of *val* was initialized to -1, thus this loop falls into an infinite loop.

Finding 9: *Undefined Behavior* (1.9%) could lead to potential infinite loop bugs, which is more difficult to confirm as it depends on the compilers and platforms.

5 ROOT CAUSES OF INFINITE RECURSION

Recursion[11] is another repeated structure that one of the steps of the function reenters the function itself. Figure 10 illustrates the hierarchical taxonomy of infinite recursion bugs in real-world C and C++ projects. Generally speaking, our taxonomy of infinite recursion bugs consists of 4 inner categories (in grey) and 8 leaf categories (in white).

Finding 10: Infinite recursion accounts for a large portion (28.5%) of non-termination bugs. 51.2% of the bugs are caused by *Incorrect Recursion* that means the incorrect design of recursions. 48.8% of the bugs are caused by *Unexpected Recursion*, *i.e.*, developers do not intend to use recursions, but the recursions are unexpectedly generated due to programming errors.

5.1 Incorrect Recursion Design (Category 1)

The termination of recursion mainly depends on the *argument* that will be sent to the parameters of the recursive function and the *return* that can determine the exit of the current iteration. In general, the incorrect recursion is mainly caused by *Incorrect Arguments* (Category 1.1), *Incorrect Return* (Category 1.2) and *Deep Recursion* (Category 1.3).

Category 1.1: Incorrect Arguments. We observe 7 infinite recursions (10.8%) in *Incorrect Recursion* that are caused by using the unchanged arguments. Similar with loop, the arguments determine the number of iterations. If the arguments keep unchanged, the recursion gets stuck to a state (*i.e.*, the values of arguments do not change) leading to the infinite execution.

Category 1.2: Incorrect Return. Most (38.5%) of the incorrect recursions are caused by the incorrect return[12]. It is because of the incomplete condition for return. For example, the current iteration should exit, but couldn't because of an incorrect return condition.

Category 1.3: Deep Recursion. We also find a large part of recursion bugs (50.7%) in *Incorrect Recursion* that can cause stack overflow rather than non-termination bugs, so are linked with a dashed line in Figure 10. When recursion is executed, the variables and some information need to be saved into stack. As the recursion can be excessively deep, it causes call stack buffer overflow. The *Deep Recursion* bugs are related to complex data structures (*e.g.*, binary tree traversal, compilation process, and database operations).

Finding 11: 10.8% of the incorrect recursions are caused when incorrect arguments (*e.g.*, unchanged value) are used for the recursions. Determining the condition for returning value is error-prone, which accounts for 38.5% of incorrect recursions. In addition, a large portion of bugs (50.7%) in incorrect recursions are deep recursion which may lead to stack overflow although they can terminate in theory.

5.2 Unexpected Recursion (Category 2)

Programming errors can lead to unexpected recursion that does not terminate. Specifically, our study shows that *Unexpected Recursion* involves 62 infinite recursion bugs that cover 61.4% (43/70) projects. The major reasons include *Incorrect Self-invoking* (Category 2.1) and *Cyclic Invoking* (Category 2.2).

Category 2.1: Incorrect Self-invoking There are 38 infinite recursions (29.9%) caused by that the function invokes itself incorrectly and unintentionally. Specifically, 12 of them are due to *Misusing Namespace* (Category 2.1.1). The function $M::f$ intends to invoke another function $N::f$ where M and N are different namespaces. However, developers forget to use N causing it to call itself. 6 of them are due to the *Miscalling Inherited Method* (Category 2.1.2). In C++ inheritance, the method m in a child class A intends to invoke the method m in another child class B, However, developers forget to use the class name B (*i.e.*, $B::m$), thus $A::m$ calls itself. 15 of them are due to *Misusing Method Overloading* (Category 2.1.3). It happens when the method m intends to invoke another overloaded function m but it incorrectly invokes itself (*i.e.*, using the same arguments). 5 of them are caused by *Missing undef Instruction* (Category 2.1.4). For this category, developers first use #define to define an identifier A as a function B. However, in the function B, it invokes A (*i.e.*, itself) again before undef A.

Category 2.2: Incorrect Cyclic Invoking. Compared to Category 2.1 which involves self-invoking, there are 24 of infinite recursions (18.9%) that are caused by cyclic calling. Given two functions A and B, if A calls B and B also calls A, then the unexpected cyclic calling is formed, causing an infinite execution.

[11]Due to the space limit and the complex structure of recursion, we put more detailed examples and discussions on our website.

[12]Without loss of generality, a recursive function that does not return anything can be considered to return null.

Finding 12: Our study shows that the number of infinite recursions caused by unexpected recursion and incorrect recursion are very close (65 and 62), indicating that unexpected recursion is also important for non-termination checking. The main reasons include the unexpected self-invoking (61.3%) and unexpected cyclic-invoking (38.7%).

6 EMPIRICAL STUDY ON TERMINATION ANALYSIS TOOLS

In this section, we introduce the benchmark extraction based on the 445 non-termination bugs and conduct an assessment of the SOTA tools based on the extracted benchmark to answer RQ2. We then conduct an in-depth analysis for the detection failures of these tools to answer RQ3.

6.1 Benchmark Extraction

To evaluate the performance of the state-of-the-art tools in analyzing the termination of real-world programs, one important step is to setup a ground-truth benchmark with real-world non-termination bugs based on a series of validity criteria. This is because real-world source code contains noises that are not relevant to non-termination bugs, and existing tools cannot be directly applied to real-world projects. For ensuring the representativeness of our benchmark, we extracted and sliced our benchmark from 445 bugs based on the following principles: *(0) Bugs Filtration.* The non-termination bugs should be further filtered because most non-termination bugs have very complex dependencies (e.g., the bugs depend on multiple code files, complex variable types, and APIs). We keep small-size bugs and filter complex bugs because it is more difficult for us to guarantee the correctness of large-scale benchmarks and existing benchmarks often cannot support complex dependencies. After this step, we keep 56 bugs to construct our benchmark from 445 non-termination bugs and used them for assessment. *(1) Context Simplification.* For infinite loop bugs, we follow four steps: First, we identify all the loop iterations (defined in §4.1) and retain the complete loop condition. Second, we retain instructions (*e.g.*, if-else branch) and objects in the source code of the project that involve loop iterators, which can change the value of loop iterators. Then, we retain all control instructions and date structure (*e.g.*, circular linked list). Finally, we remove other instructions that are extraneous with the loop in the project. For infinite recursion bugs, we retain the instructions related to the value change of recursion parameters. *(2) Function Rewriting.* We rewrite the function and retain the effect and return values of the function (including API function and custom function). For recursive bugs, we focus more attention on retaining recursive calls between functions. *(3) Reserve Name and Type.* We keep the consistent name and type of variables and functions in our benchmark with projects. *(4) Make Benchmark Executable.* We adapt the selected loops and recursion functions by putting the loops and the first function call of recursion in a main function and adding non-deterministic initialization for the variables to make them executable. Note that we extracted the non-termination benchmark based on non-termination bugs and the termination benchmark from the corresponding fixed versions relying on the above principles. We highlight that the extraction of the recursion benchmark is more difficult than loop because it involves more data structures and function calls. Finally, we constructed our benchmark set, including a non-termination dataset with 56 real and reproducible non-termination bugs and a termination dataset with 58 fixed programs. Through our statistics, the average numbers of lines in SV-COMP 2021 benchmarks and our benchmarks are 20.82 and 24.52, respectively.

```
1  int main(){
2      unsigned char l = __VERIFIER_nondet_uchar();
3      while( l-- )
4          if( l-- ){ //loop }
5      return 0;
6  }
```

Figure 11: Benchmark program extracted from Figure 3.

Figure 3 shows an infinite loop and the corresponding extracted benchmark is shown in Figure 11. First, based on the definition of loop iterator in §4.1, the unsigned char variable l is a loop iterator in loop condition (Line 4). Therefore, we retain the source code in Line 2, 4 and 6 (7) to our benchmark (correspond to Line 2, 3 and 4 in Figure 11). Other variables (v, i, and o) can be ignored because they do not impact any loop iterators and loop termination property of this benchmark. Notice that we keep the consistent name of variables (l) and variable type (unsigned char) in our benchmark with real-world projects and we set l to be non-deterministic. In addition, we set a main function to make it executable.

Correctness of Benchmark. Since the extracted benchmarks may contain mistakes due to subjective biases (details in § 7.2), we put lots of effort to ensuring the correctness of our benchmarks (*e.g.*, whether it can terminate or not). First, we manually verify our benchmark with at least three co-authors. Only if all the authors confirm the benchmark, we accept the results. For non-termination benchmarks, except for the manual confirmation, we also directly run the program by setting the potential bug-triggering values. We ensure that the benchmarks cannot terminate in half an hour which could be a reasonable indicator because our program is simple and has no huge bound (*e.g.*, i<1000000). For terminating benchmarks, theoretically, it is difficult to ensure that the programs can terminate for all inputs. Except for our manual confirmation, we found that existing tools also rarely determine them as non-terminating which could be an indicator for the correctness of terminating cases. Finally, we construct a benchmark set, including a non-termination dataset with 56 real and reproducible bugs and a termination dataset with 58 fixed versions. Note that 2 non-terminating programs related to undefined behavior are removed from our benchmark because they cannot be successfully reproduced.

6.2 Tool Assessment

In this section, our goal is to explore whether the termination of programs in our benchmark can be determined correctly by state-of-the-art termination analysis tools. In order to make our assessment

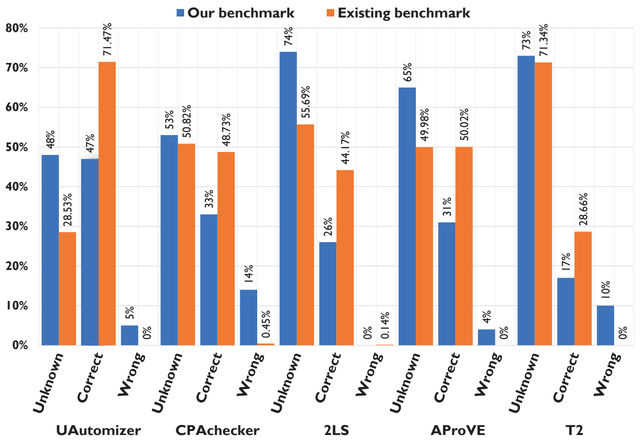

Figure 12: Comparison results of the termination analysis tools on the existing benchmark and ours.

Table 2: The result of these five tools on our benchmark involving special features. UA. refers to UAutomizer and Ap. presents AProVE. UN refers to UNKNOWN and W refers to WRONG.

Features (Total)	Cate.	UA.	CPA	2LS	Ap.	T2	Avg.	Perc.
Pointer Manipulation (10)	UN	8	10	10	9	10	9.4	94.00%
Recursion (10)	UN	5	7	7	7	10	7.2	72.00%
Array (16)	UN	9	16	13	14	16	13.6	85.00%
Data Structure (14)	UN	8	12	12	13	14	11.8	84.29%
Overflow (16)	UN	10	5	10	11	8	8.8	55.00%
	W	3	6	0	3	3	3.0	18.75%
Type (10)	UN	7	2	6	7	4	5.2	52.00%
	W	0	4	0	1	3	1.6	16.00%
Bit Calculation (15)	UN	5	2	11	14	6	7.6	50.67%
	W	3	6	0	0	4	2.6	17.33%
Total (67)	UN+W	43	51	48	59	56	51.4	76.72%

more convincing, we select five existing tools for conducting our experiment: UAutomizer [34], CPAchecker [9], 2LS [48], AProVE [31], and T2 [11]. Among them, UAutomizer won the first prize in the termination category from SV-COMP 2017 to SV-COMP 2021 [51]. CPAchecker and 2LS respectively won the silver and bronze for the termination category in SV-COMP 2020 and SV-COMP 2021. The performance of AProVE is demonstrated in the annual international competition of Termination Tools, and AProVE got the top three from SV-COMP 2017 to SV-COMP 2019. T2 is powerful and more successful than Julia [49] and TNT [33], which is demonstrated in [14]. Therefore, our selected five tools are representative tools for termination analysis. The versions of these tools we used in the experiments are: UAutomizer (0.2.2, provided version for SV-COMP 2022), CPAchecker (2.0), 2LS (0.9.5), AProVE (provided a version for SV-COMP 2022), and T2 (2016 version). Note that, we tested these five tools in Ubuntu 4.15.0 with a memory limit of 14.6 GiB of RAM, a runtime limit of 15 min of CPU time, and a limit to 8 processing units of a CPU, the same configurations used in SV-COMP.

We evaluate our benchmark on the state-of-the-art termination analysis tools, however, these tools cannot directly work on a part of special variable types and functions in real-world projects. Therefore, we first replace these special variable types with their supported types and rewrite the functions with the form that can be supported in the termination analysis tools, *e.g.*, file type variables are replaced by array. We find that the five tools all cannot work on the C++ benchmark but can work on the C benchmark. Therefore, 14 recursion benchmarks involving C++ characteristics (*e.g.*, class, inheritance) cannot be accepted by these five tools. In conclude, we test 100 benchmarks in C programs in total, and 90 of them are loops, the others are recursions. We provide replication packages (including log files, command lines, and our benchmark files) on Zenodo[13] and we make the packages available for others to allow other researchers and practitioners to generate interesting ideas and build upon our work.

Figure 12 shows the statistical results of these tools on our benchmark. Overall, the capabilities of these five tools to make correct verdicts drop compared with existing benchmarks. Among them, the capabilities of UAutomizer to make correct verdicts drop most, from 71.47% correctly in SV-COMP 2021 [8] to 47% correctly in our benchmark. Besides, as shown in Figure 12, all five tools prefer to answer "UNKNOWN", which indicates the complexity of our benchmark is more than that in existing benchmarks. The dropping in accuracy and the preference to answer "UNKNOWN" involves two main reasons: *(1)* During our benchmark extraction, we set the precondition of these programs to be non-deterministic, which causes an over-approximation to their real preconditions and makes termination analysis more complex. For example, in Figure 3 and Figure 11, the value of l depends on the value passed when calling the function, and we set l to be non-deterministic. *(2)* The benchmark extracted from real-world projects may contain complex computation (*e.g.*, bit calculation, pointer manipulation), various data structures (*e.g.*, linked list), and data type (*e.g.*, size_t). Furthermore, the error rate of these tools is increased in Figure 12. Among them, CPAchecker has the highest error rate (*i.e.*, 14%), more than 30 times the error rate in SV-COMP 2021 [8] (*i.e.*, 0.45%).

We also tried to analyze the relationship between the success rate and different categories [14]. In general, these tools perform better on the Logical Error category than General Programming Error category, confirmed that existing tools mainly prove (non)termination logically. For Logical Error benchmarks, since they have been manually simplified from real-world projects, these tools could handle them relatively easier. To our surprise, the non-terminating benchmarks about *Reusing Same Loop Iterator* cannot be handled by all tools although they look quite simple. For others, we could not find a clear relation between the success rate and different leaf categories. Therefore, we further manually analyzed the failed cases and summarized common features that revealed the weaknesses of existing tools (refer to Section 6.3).

[13]DOI:10.5281/zenodo.6548310

[14]Due to the space limit, more details including the results of each tool and the results on each category can be found on our website [3].

6.3 Analysis of the Unhandled Cases

Based on the results shown in Figure 12, compared with the results on existing benchmarks, the error rate of four tools except 2LS and the "UNKNOWN" rate of these five tools are raising on our benchmark. Therefore, we further analyze these unhandled bugs to explore the weaknesses of these existing tools.

Table 2 shows the results that the tools cannot handle benchmarks involving special features. The first column represents the special features that can significantly affect the results of the tools. We also list the total number of benchmarks involving corresponding features. The second column refers to the results that the tools analyze these programs. First, all of these special features in the first column can make the termination analysis tools prefer to answer "UNKNOWN", especially for the *pointer manipulation*. 94.00% benchmarks involving *pointer manipulation* are unhandled by these tools. However, these special features are commonly used in real world, which indicates that the state-of-the-art tools are inefficient and immature in analyzing the termination of real-world projects.

Except for making these tools prefer to answer "UNKNOWN", some features can affect the soundness of these tools (*i.e.*, wrong answers), which should be paid more attention. It is mainly because some programming errors caused by *Overflow, Type*, and *Bit Calculation*, which can change the ideal execution of programs, are not well considered. For example, on average, the programs involving *Overflow, Type*, and *Bit Calculation* can introduce 18.75%, 16.00% and 17.33% wrong results, respectively. This strongly indicates that the termination analysis techniques should take these features into account to make them sound and more practical. In addition, we observe that 2LS does not have any wrong answers because these potential programming errors have been considered [48].

Finding 13: We identify seven programming features that can pose challenges to the existing termination analysis tools. They tend to produce unknown and erroneous results (76.72%) on the programs involving these features.

7 DISCUSSIONS

7.1 Implications

For program developers. Firstly, the findings from our study can help avoid non-termination bugs for developers in developing programs. Specifically, when working on loops, developers should (1) be careful with the writes to loop iterators and ensure the loop variables never end up staying constant; (2) be careful with the reuse of loop iterators; (3) check for possible overflow and type conversion errors, and carefully choose appropriate variable types. For recursions, programmers should be careful about unexpected recursions and pay attention to the recursion arguments and returns.

Secondly, we provide useful advice for troubleshooting non-termination bugs. Specifically, fix strategies of infinite loops mainly include: (1) *Add missing constraints to loop conditions*. This fixes incorrectly chosen loop condition that is inconsistent with the programmers' expectations. (2) *Handle specific values*. Most infinite loops end up being stuck in one state. Developers should first identify the problematic program state and then handle specific values (*e.g.*, add an if statement and "break" the execution from the loop) for the state. (3) *Use loop counters*. Loop counters can limit the maximum executions of the loop. This fix strategy is simple but may lead to errors in subsequent procedures. It is a reasonable choice when the value of each variable at the end of the loop has no impact on the execution of the subsequent code. For infinite recursions, the developers should identify if recursion is used intentionally. If so, they mainly correct the *argument* as well as the *return*. Otherwise, one should break from the recursive or cyclic calls.

For researchers. Based on the experimental results in § 6.2, we noticed that these tools fail to work on real-world projects directly and perform worse on our real-world benchmark than on existing well-established benchmarks. Therefore, existing termination analysis tools should be improved in terms of their scalability and applicability on real-world projects. Furthermore, general programming errors, such as overflows, should be paid more attention in future termination analysis research.

7.2 Threats to Validity

The collection of real-world projects may introduce bias. To mitigate this threat, we downloaded 1,600 C/C++ projects to expand the evaluation scope with our best efforts. Furthermore, we further selected five commonly used keywords to identify potential non-termination-related commits. Due to the complexity of real-world projects, the classification of the non-termination bugs inevitably involves subjective biases. To address this, we filtered our dataset based on well-thought-out criteria, mentioned in § 3.1. While constructing the non-termination benchmark, the necessary simplification and abstractionmay change the original program behaviors. To mitigate such a threat, each benchmark extracted was inspected by two authors independently and any discrepancy was discussed until a consensus was reached.

8 CONCLUSION

In this paper, we conducted a study of 445 non-termination bugs collected from 199 real-world OSS projects. With substantial manual efforts, we presented a systematic taxonomy of non-termination bugs including 16 categories of infinite loops and 8 groups of recursions, and further extracted a novel benchmark including 114 programs simplified from these bugs. Moreover, we evaluated five state-of-the-art termination analysis tools using our newly constructed benchmark and identified challenges that these tools face. Finally, we highlighted the limitations of existing termination analysis techniques and discussed new research directions.

9 DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in [2] and the artifact is openly available in [1].

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (No. 61872262), the Ministry of Education, Singapore under its Academic Research Fund Tier 1 (21-SIS-SMU-033, T1-251RES1901), Tier 2 (MOE2019-T2-1-040, T2EP20120-0019), and Tier 3 (MOET32020-0004).

REFERENCES

[1] 2022. *Artifact for Large-Scale Analysis of Non-Termination Bugs in Real-World OSS Projects*. https://doi.org/10.34740/kaggle/ds/2440949

[2] 2022. *Data for Large-Scale Analysis of Non-Termination Bugs in Real-World OSS Projects*. https://zenodo.org/record/6548310#.Yn-DBOhByUk

[3] 2022. *Large-Scale Analysis of Non-Termination Bugs in Real-World OSS Projects*. https://sites.google.com/view/non-termbug/home

[4] Sheshansh Agrawal, Krishnendu Chatterjee, and Petr Novotný. 2017. Lexicographic Ranking Supermartingales: An Efficient Approach to Termination of Probabilistic Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 34 (dec 2017), 32 pages. https://doi.org/10.1145/3158122

[5] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 117–133.

[6] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. 2012. Detecting Fair Non-termination in Multithreaded Programs. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 210–226.

[7] Amir M. Ben-Amram and Samir Genaim. 2015. Complexity of Bradley-Manna-Sipma Lexicographic Ranking Functions. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 304–321.

[8] Dirk Beyer. 2021. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 401–422.

[9] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–190.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 491–504.

[11] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 387–393.

[12] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 609–633.

[13] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and undefinedorundefinede Žikelić. 2021. Proving Non-Termination by Program Reversal. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3453483.3454093

[14] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–171.

[15] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2015. Synthesising Interprocedural Bit-Precise Termination Proofs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, Lincoln, Nebraska, 53–64. https://doi.org/10.1109/ASE.2015.10

[16] Jianhui Chen and Fei He. 2020. Proving Termination by <i>k</i>-Induction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1239–1243. https://doi.org/10.1145/3324884.3418929

[17] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An empirical assessment of security risks of global Android banking apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1310–1322.

[18] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are mobile banking apps secure? what can be improved?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 797–802.

[19] Sen Chen, Yuxin Zhang, Lingling Fan, Jiaming Li, and Yang Liu. 2022. AUSERA: Automated Security Risk Assessment for Vulnerability Detection in Android Apps. In *Proceedings of the 37th ACM/IEEE International Conference on Automated Software Engineering*. 1–5.

[20] Yinghua Chen, Bican Xia, Lu Yang, Naijun Zhan, and Chaochen Zhou. 2007. Discovering Non-linear Ranking Functions by Solving Semi-algebraic Systems. In *Theoretical Aspects of Computing – ICTAC 2007*, Cliff B. Jones, Zhiming Liu, and Jim Woodcock (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 34–49.

[21] International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2021. *Collection of Verification Tasks.* International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Retrieved Oct 5, 2021 from https://github.com/sosy-lab/sv-benchmarks/

[22] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Disproving Termination with Overapproximation. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design* (Lausanne, Switzerland) *(FMCAD '14)*. FMCAD Inc, Austin, Texas, 67–74.

[23] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving Program Termination. *Commun. ACM* 54, 5 (may 2011), 88–98. https://doi.org/10.1145/1941487.1941509

[24] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Verification, Model Checking, and Abstract Interpretation*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–24.

[25] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding Integer Overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 2 (dec 2015), 29 pages. https://doi.org/10.1145/2743019

[26] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 131–142.

[27] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018. Efficiently manifesting asynchronous programming errors in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 486–497.

[28] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-Scale Analysis of Framework-Specific Exceptions in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 408–419. https://doi.org/10.1145/3180155.3180222

[29] Sally Fincher and Josh Tenenberg. 2010. Making sense of card sorting data. *Expert Systems* 22, 3 (2010), 89–93.

[30] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, . Chen, and Qi Alfred. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 385–396.

[31] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2014. Proving Termination of Programs Automatically with AProVE. In *Automated Reasoning*, Stéphane Demri, Deepak Kapur, and Christoph Weidenbach (Eds.). Springer International Publishing, Cham, 184–191.

[32] GitHub 2022. *GitHub Docs.* GitHub. Retrieved Mar 10, 2022 from https://docs.github.com/cn

[33] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving Non-Termination. *SIGPLAN Not.* 43, 1 (jan 2008), 147–158. https://doi.org/10.1145/1328897.1328459

[34] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 797–813.

[35] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.

[36] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.

[37] ISO/IEC JTC 1/SC 22 Programming languages, their environments and system software interfaces 1999. *ISO/IEC 9899:1999 Programming languages — C.* ISO/IEC JTC 1/SC 22 Programming languages, their environments and system software interfaces. https://www.iso.org/standard/29237.html

[38] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2014. Proving Non-termination Using Max-SMT. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 779–796.

[39] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: Dynamic Termination and Non-Termination Proofs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 189 (nov 2020), 30 pages. https://doi.org/10.1145/3428257

[40] Yingwen Lin, Yao Zhang, Sen Chen, Fu Song, Xiaofei Xie, Xiaohong Li, and Lintan Sun. 2021. Inferring Loop Invariants for Multi-Path Loops. In *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 63–70. https://doi.org/10.1109/TASE52547.2021.00030

[41] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1013–1024. https://doi.org/10.1145/2568225.2568229

[42] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. 329–339.

[43] Fonenantsoa Maurica, Frédéric Mesnard, and Étienne Payet. 2016. Termination Analysis of Floating-Point Programs Using Parameterizable Rational Approximations. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. Association for Computing Machinery, New York, NY, USA, 1674–1679. https://doi.org/10.1145/2851613.2851834

[44] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *2013 IEEE International Conference on Software Maintenance*. 70–79. https://doi.org/10.1109/ICSM.2013.18

[45] David Menendez and Santosh Nagarakatte. 2016. Termination-Checking for LLVM Peephole Optimizations. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/2884781.2884809

[46] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation*, Bernhard Steffen and Giorgio Levi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 239–251.

[47] Lili Quan, Qianyu Guo, Xiaofei Xie, Sen Chen, Xiaohong Li, and Yang. Liu. 2022. Towards Understanding the Faults of JavaScript-Based Deep Learning Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*.

[48] Peter Schrammel and Daniel Kroening. 2016. 2LS for Program Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 905–907.

[49] Fausto Spoto, Fred Mesnard, and Étienne Payet. 2010. A Termination Analyzer for Java Bytecode Based on Path-Length. *ACM Trans. Program. Lang. Syst.* 32, 3, Article 8 (mar 2010), 70 pages. https://doi.org/10.1145/1709093.1709095

[50] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. 2015. Securing Android: A Survey, Taxonomy, and Challenges. *ACM Comput. Surv.* 47, 4, Article 58 (may 2015), 45 pages. https://doi.org/10.1145/2733306

[51] International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2021. *Result of 10th Competition on Software Verification (SV-COMP 2021)*. International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Retrieved Oct 5, 2021 from https://sv-comp.sosy-lab.org/2021/results/results-verified/

[52] The Termination Problem Database 2021. The Termination Problem Database. Retrieved 1 Jul 2021 from https://github.com/TermCOMP/TPDB

[53] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 999–1010.

[54] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1176–1199.

[55] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2950290.2950340

[56] Xiaofei Xie, Bihuan Chen, Liang Zou, Shang-Wei Lin, Yang Liu, and Xiaohong Li. 2017. Loopster: Static Loop Termination Analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 84–94. https://doi.org/10.1145/3106237.3106260

[57] Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. 2019. Automatic Loop Summarization via Path Dependency Analysis. *IEEE Transactions on Software Engineering* 45, 6 (2019), 537–557. https://doi.org/10.1109/TSE.2017.2788018

[58] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-Looper: Automatic Summarization for Multipath String Loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 188–198. https://doi.org/10.1145/2771783.2771815

[59] Tao Ye, Lingming Zhang, Linzhang Wang, and Xuandong Li. 2016. An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 91–101.