Precise Divide-By-Zero Detection with Affirmative Evidence

Yiyuan Guo  
The Hong Kong University of Science and Technology  
Hong Kong, China  
yguoaz@cse.ust.hk

Jinguo Zhou  
Ant Group  
China  
jinguo.zjg@antgroup.com

Peisen Yao  
The Hong Kong University of Science and Technology  
Hong Kong, China  
pyao@cse.ust.hk

Qingkai Shi  
Ant Group  
China  
qingkai.sqk@antgroup.com

Charles Zhang  
The Hong Kong University of Science and Technology  
Hong Kong, China  
charlesz@cse.ust.hk

ABSTRACT
The static detection of divide-by-zero, a common programming error, is particularly prone to false positives because conventional static analysis reports a divide-by-zero bug whenever it cannot prove the safety property — the divisor variable is not zero in all executions. When reasoning the program semantics over a large number of under-constrained variables, conventional static analyses significantly loose the bounds of divisor variables, which easily fails the safety proof and leads to a massive number of false positives. We propose a static analysis to detect divide-by-zero bugs taking additional evidence for under-constrained variables into consideration. Based on an extensive empirical study of known divide-by-zero bugs, we no longer arbitrarily report a bug once the safety verification fails. Instead, we actively look for affirmative evidences, namely source evidence and bound evidence, that imply a high possibility of the bug to be triggerable at runtime. When applying our tool Wrr to the real-world software such as the Linux kernel, we have found 72 new divide-by-zero bugs with a low false positive rate of 22%.

CCS CONCEPTS
- Software and its engineering → Software verification and validation.

KEYWORDS
Static program analysis, bug detection, divide-by-zero.

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.

ACM Reference Format:

1 INTRODUCTION
Divide-by-zero is one of the most common programming errors that can lead to undefined runtime behaviors. Over the past five years, there are more than 90 security vulnerabilities related to divide-by-zero documented in the Common Vulnerabilities and Exposures (CVE) database. Static analysis is an important approach to combat this problem. However, its high false positive rate remains a major obstacle for adoption. In our experiments, we observe false positive rates of over 70% in representative static analyzers.

To understand the reason behind these high false positive rates, we first examine how static analysis reports a potential divide-by-zero bug. The central question is how to decide if the divisor variable may equal zero in some executions. More specifically, a divide-by-zero happens when

C1. There exists a feasible execution path P reaching the division instruction (Let \( pc \) denote the path condition for \( P \)).
C2. The divisor variable \( v \) may equal zero when \( P \) reaches the division instruction (Let \( Q \) be the error condition for triggering a divide-by-zero bug).

Conventionally, static analysis reports a divide-by-zero bug if \( pc \land Q \) is satisfiable, i.e., under the condition \( pc \), the divisor variable \( v \) cannot be proved to be non-zero. However, we observe that this scheme often leads to a high false positive rate in divide-by-zero detection since the under-constrainedness of static analysis makes it easy to satisfy the query \( pc \land Q \). Many variables are under-constrained in the static analysis [13]:

- The program under analysis can be an open program. For example, the analysis often sees a module by interface only, of which the variables are under-constrained [11, 13].
- The inherent limitations of static analysis introduce the under-constrained variables to represent imprecision in the analysis, e.g., the runtime state of the operating system and the presence of unmodeled code constructs, such as inline assembly and many others [13, 14].

To improve the precision, our key insight is that, instead of reporting errors whenever safety verification fails, we can attack the divide-by-zero detection problem from a different angle by actively finding affirmative evidence for triggering the bug. Such evidence serves as the extra information on the under-constrained variables, thus contributes to improving the precision.
We use the motivating example in Figure 1 to illustrate the idea. In Figure 1, the function move updates the control panel ctr by adding values to its $(x, y, z)$ coordinates computed in the function get_step through a division operation. Also, we assume that the caller function top_fun() in the function move is external to the module under analysis and, thus, not tracked by the analysis. Now consider how static analysis can detect potential divide-by-zero errors in this program.

Line 20 cannot trigger a divide-by-zero because the used divisor df is either $-1$ or no less than $1$. A path-sensitive static analysis correctly verifies its safety since the two queries $dx \geq dy \land 1 + dx - dy = 0$ and $dx < dy \land -1 = 0$ associated with the two paths across Lines 15-18 are both unsatisfiable.

However, the path-sensitive static analysis still encounters the precision problem. In our example, the divide-by-zero errors are reported for the three calls to get_step at Line 25, 26, and 27 because the queries $dx = 0$, $dy = 0$, and $dz = 0$ are all satisfiable. However, since top_fun() only calls move with non-zero values for $dx$ and $dy$ (Line 10), the reports at Lines 25-26 are false positives. This fact is unknown to the static analysis because it fails to trace back to the origin of the arguments of move in the function top_fun().

On the other hand, we notice that the divide-by-zero report at Line 27 has high confidence to be true since we can find evidence to explain it. Specifically, Line 21 of Figure 1 explicitly compares $dx$ and $dy$ with $d$, suggesting that the programmer has beliefs that they may be equal to $d$. If such beliefs actually hold, $dz$ must equal zero at Line 24, i.e., $dz = 2*dx - (dy + d) = 2*d - (d + d) = 0$, leading to a plausible divide-by-zero report for Line 27.

How to report the high confidence divide-by-zero bug at Line 27 instead of the false positives at Lines 25-26? Note that all of the three divisions cannot be proved safe by the static analysis, regardless of being path-sensitive or not. However, we have identified the evidence based on analyzing the programmer’s beliefs for the bug report at Line 27, which leads to its high confidence to occur. Specifically, in this work, we identify two categories of evidence:

(a) **Source evidence**: The fact that an explicit source of “bad” value is assigned to a variable $v$. The source can be either a direct assignment of zero (e.g., the assignment $v := 0$) or takes the value of some tainted input (e.g., $v := atoi(argv[1])$).

(b) **Bound evidence**: The equality fact $v_1 = v_2$ generated from a bound checking statement in the program that compares $v_1$ and $v_2$. Statements like Line 21 of Figure 1 that check numerical bounds of variables can convey the important information on the possible values of the checked variables: a comparison between $v_1$ and $v_2$ suggests that the programmer may believe that $v_1$ can equal $v_2$.

With the generated evidence, our method reports a divide-by-zero bug by adapting the requirements C1 and C2 as follows.

C1'. C1 and C2 hold: $pc \land Q$ is satisfiable, where $pc$ denotes the path condition for reaching the division instruction, and $Q \equiv (v = 0)$ is the error condition for divide-by-zero.

C2'. One of the following conditions holds:

- The divisor variable $v$ has the source evidence.
- There exists a set $E$ of bound evidence consistent with $pc$ such that $v$ must be zero under $E$.

In other words, we seek to detect a fraction of divide-by-zero bugs with high confidence by finding the affirmative evidence to explain its occurrence. For example, in Figure 1, $dz$ at Line 27 is the only variable that meets these requirements: $dz = 0$ must hold if the evidence $dx = d$ and $dy = d$ hold. Hence, our approach will only report one divide-by-zero bug at Line 27.

In this paper, we propose Wit, a framework for the precise detection of divide-by-zero with the affirmative evidence. First, to understand the applicability of our definition of evidence and the criteria C1’-C2’ for detecting bugs, we perform an empirical study on existing CVEs related to divide-by-zero bugs and investigate if they can be detected by finding a set of supporting evidence. The result shows that 74% of the studied divide-by-zero bugs have the supporting evidence that explains its occurrence, showing the generality of our intuition. To capture the evidence-based reasoning in achieving the precise divide-by-zero detection, we design a static analysis algorithm to perform the evidence-based inference, adhering to the criteria C1’-C2’ for reporting bugs. The algorithm generates evidence directly from certain code patterns and propagates the generated evidence path-sensitively. To improve its efficiency, we utilize a symbolic domain to compactly encode the possible numerical values for variables and apply the data dependence analysis [17] to prune irrelevant execution paths, scaling to million lines of code.

In summary, we make the following contributions in this paper:

- The insight for improving the precision of divide-by-zero detection by finding the affirmative evidence to trigger the bug.

```
struct control_panel {
  int x; int y; int z;
  int flag; int distance; ...
};

void top_fun() { // not tracked by the analysis when analyzing move
  if (dx != 0 & dy != 0)
    // calls move with non-zero values of dx and dy.
  move(dx, dy, d, ctr);
}

int move(int dx, int dy, int d, control_panel ctr) {
  int diff;
  if (dx == dy)
    diff = 1 - dx - dy;
  else
    diff = -1;
  ctr->x = get_step(ctr, dx);
  ctr->y = get_step(ctr, dy);
  ctr->z = get_step(ctr, dz);
}

int get_step(control_panel ctr, int step_size) {
  return ctr->distance / step_size;
}
```

**Figure 1**: A motivating example.
• An empirical study of CVEs related to divide-by-zero bugs. We investigate and classify these existing divide-by-zero bugs and show that many of them can be effectively detected based on evidence.
• A formalization of the insight in a semantic framework and an algorithm for finding high confidence divide-by-zero errors through the evidence propagation.
• An implementation and extensive evaluation of the divide-by-zero checker. We demonstrate that it is both precise and efficient, uncovering 72 divide-by-zero bugs (14 of which are confirmed by the developers) in large codebases such as the Linux kernel with a low false positive rate of 22%.

2 EMPIRICAL STUDY ON DIVIDE-BY-ZERO BUGS

To further understand the applicability of the evidence-based divide-by-zero detection method, we perform an empirical study on CVEs caused by divide-by-zero bugs. Through the empirical study, we aim to answer the following research questions:

• RQ1: How often can we find the evidence for the divide-by-zero bugs? In other words, how often do our bug detection criteria C1* - C2* apply to the existing divide-by-zero bugs?
• RQ2: What is the distribution for the two kinds of evidence defined in § 1?

2.1 Data Collection

We search for the keywords "divide-by-zero" and "divide by zero" in the CVE database and examine the CVEs starting from the year 2011. There are 123 CVEs in total that are caused by the divide-by-zero bugs. We exclude the bugs with no source code or stack traces (16), or unable to understand without a deep knowledge of the system (12), or sharing the same root with other bugs (8). Thus, we are left with 87 CVEs to study.

2.2 Classification Criteria

Since our goal is to study the problem in general without being tied to a specific analysis algorithm or target system, we have adopted the following criteria to mimic the reasoning process of a static analyzer:

(1) Starting from the crash site of the bug (i.e., the division instruction with zero divisor), we manually examine a backward slice \( S \) of some fixed size. In our experiment, we examine backwards at most 10 call frames from the involved division. This is reasonable as lengthy bug traces output by static analyzers take a non-trivial amount of time for users to triage [1, 4], thus should be avoided by practical tools.
(2) Based on our manual inspection of \( S \), we classify the divide-by-zero bugs based on the evidence found:
(a) Class Src: Source evidence is found. An explicit source of zero value for the divisor variable exists in \( S \). The source is either a direct assignment of zero value to the variable or of a tainted value from the input.
(b) Class Bd: Bound evidence is found. We can find a set of evidence from the bound checking statements in \( S \) that guarantee the divisor variable to be zero. Specifically, for any branching statement involving comparisons: \( x \ cmp y \) (**cmp** is \( < \), \( <= \), etc.), we consider the fact \( x=y \) as the bound evidence likely to be true.

(c) Class Un: No evidence is found (i.e., unknown). We can neither find source evidence nor bound evidence in \( S \). For example, CVE-2018-19628 is marked as unknown because it requires a deep context of 14 call frames to understand the root cause, exceeding the code range of \( S \).

Conventional static analysis methods detect all three classes of divide-by-zero bugs. As illustrated in § 1, they are likely to incur massive false positives. Our method detects bugs of Class Src and Class Bd since they are the classes that satisfy the requirement C1* - C2*, aiming for a fraction of high confidence divide-by-zero bugs with possible false negatives. The classification attempts to study how much can be covered by our method.

2.3 Study Result

**Bug Classification.** Table 1 shows the classification result of the 87 CVEs caused by divide-by-zero bugs. For Class Src bugs with explicit sources, we further divide them into two groups: those with constant zero value as the source (the column labeled with "const") and those with the tainted input as the source (the column labeled with "taint"). The bugs with both the source evidence and the bound evidence are counted in the column labeled "Class Src ∩ Class Bd".

<table>
<thead>
<tr>
<th>Class Src</th>
<th>Class Bd</th>
<th>Class Src ∩ Class Bd</th>
<th>Class Un</th>
</tr>
</thead>
<tbody>
<tr>
<td>const</td>
<td>19</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>taint</td>
<td>21</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

From the data we gather, we answer the two research questions empirically as follows:

**Answer to RQ1:** We can find evidence for a large proportion \( \frac{|\text{Class Src} \cap \text{Class Bd}|}{87} = 74% \) of divide-by-zero bugs. Thus, our bug detection criteria C1* - C2* are widely applicable.

**Answer to RQ2:** For the divide-by-zero bugs that we can find evidence, 73% of them have source evidence, 33% of them have bound evidence, and 6% of them have both evidence.

Therefore, we conclude that finding evidence to facilitate the precise divide-by-zero detection indeed works in real scenarios, covering a large proportion of interesting bugs. Also, both source evidence and bound evidence are useful for bug detection in practice.

**Case Study of Bugs with Evidence.** To give an intuitive understanding of the bugs detected based on evidence, we illustrate some example CVEs from Class Src and Bd.

Table 1: Classification of 87 CVEs caused by divide-by-zero bugs. Class Src is bugs with source evidence (further classified into constant source and taint source), Class Bd is bugs with bound evidence, and Class Un is bugs with no evidence.
void start_input_gifj_compress_ptr info, cjpeg_source_ptr sinfo) {
U_CHAR hdrbuf[10];
//ReadOK is a wrapper of read
if ((ReadOK(source->pub.input_file, hdrbuf, 6))
    EEXIT(info, JERR_GIF_NOT);
width = LM_to_uint(hdrbuf, 0);
height = LM_to_uint(hdrbuf, 2);
//use the second arg as divisor
DIVOP(info, width, NUMCOLORS);
}

(a) Example #1: CVE-2021-20205
int ApplyEvaluateOperator(RandomInfo *r, Quantum p, Operator type, int v) {
    switch(op) {
    case DivideEvaluateOperator: 
        result = p / (v == 0 ? 1 : v);
        break;
    case GaussianNoiseEvaluateOperator: 
        result = GenerateDifferentialNoise(r, GaussianNoise, v);
    }
    return result;
}
int GenerateDifferentialNoise(RandomInfo *r, Quantum p, NoiseType ty, int v) {
    noise = (QuantumRange * 1 / v);
}

(b) Example #2: CVE-2021-20176

Figure 2: Example of bugs with evidence.

If we apply the polyhedra abstract domain [9] that is an expressive domain capable of reasoning about the linear relations among variables, the branch statements from Line 15 to Line 18 in Figure 1 will produce the following abstractions along the two paths:

\[
\begin{align*}
A_{b1} : (dx - dy &\geq 0, dx - dy - diff \neq 1 = 0) \\
A_{b2} : (dx - dy &\leq 0, diff + 1 = 0)
\end{align*}
\]

Since the two abstractions are joined to produce a sound over-approximation when paths merge in abstract interpretation, a severe precision loss can occur. Specifically, since \( A_{b1} \lor A_{b2} = (diff \geq -1) \), the analysis concludes that \( diff \) may be equal to zero and generates a false positive divide-by-zero bug for the call at Line 20. Thus, lacking path sensitivity, the numerical abstract interpretation can be imprecise for divide-by-zero detection.

On the other hand, the under-constrained symbolic execution performs the path-by-path exhaustive exploration, starting from the tested function \( \text{move} \) (recall we assume that the analysis does not track the function \( \text{top}_\text{fun} \)). The analysis records the two paths in the function \( \text{move} \) reaching Line 20 as the following execution states:

\[
\begin{align*}
s_1 = (dx \geq dy) \land (diff + 1 = dx - dy) \\
s_2 = (dx < dy) \land (diff = -1)
\end{align*}
\]

With the path-sensitive state encoding, the under-constrained symbolic execution successfully proves that divide-by-zero cannot happen for Line 20. However, as illustrated in § 1, due to the presence of the under-constrained variables, we conclude that all three calls at Lines 25-27 of Figure 1 can trigger divide-by-zero with two false positives. This is because all the execution paths reaching these lines have satisfiable constraints for triggering divide-by-zero. Thus, without finding the affirmative evidence, even a path-sensitive technique for divide-by-zero detection can be imprecise when faced with under-constrained variables.

To summarize, conventional approaches to divide-by-zero detection suffer from imprecision problems because they may lack path sensitivity or do not attempt to find evidence for potential bugs.

Our Approach. As illustrated in § 1, our work uses the criteria \( C^1 \sim C^2 \) to precisely report potential bugs. We argue that both path sensitivity of the analysis and finding affirmative evidence are crucial in achieving good precision. Thus, we seek to track the evidence path-sensitively to achieve the precise divide-by-zero detection.

For this purpose, the analysis needs to reason about the numerical computations path-sensitively and fuse the evidence in the analysis process. We propose to apply a dedicated symbolic domain \( \Gamma \) to track the numerical computation in the analysis path-sensitively and utilize the evidence to refine the analysis result. Specifically, \( \Gamma \) represents the abstract state for a variable \( v \) as a guarded symbolic value set \( \Gamma(v) = (v_{a1}, c_1), \ldots, (v_{aq}, c_q) \), meaning that \( v = v_{a1} \) may hold under the condition \( c_1 \). Note that it is nontrivial to compute \( \Gamma \) efficiently, and we defer the details of the algorithm to § 4. For the example in Figure 1, our analysis deduces that

\[
(1 + dx - dy, dx \geq dy), (-1, dx < dy) \in \Gamma(diff) \\
(2 \times dx - (dy + d), true) \in \Gamma(dz)
\]
The analysis successfully infers that no divide-by-zero can occur for Line 20 based on the path-sensitive representation of $\Gamma(\text{diff})$. However, it still reports two spurious divide-by-zero bugs for Lines 25-26, similar to the under-constrained symbolic execution approach, as the variables $dx, dy,$ and $d$ are under-constrained in $\Gamma$.

Thus, to further improve the precision, our analysis attempts to deduce the evidence for those under-constrained variables. First, it attempts to find the source evidence for $dx, dy,$ and $d$ by asking “Are these variables tainted?” The tracking of the source evidence can be done by the taint analysis [31], following the value that originates from a bad source and checking where it flows to. In this example, no such source evidence exists. On the other hand, the analysis tries to find the bound evidence to constrain the under-constrained variables. As introduced in § 1, we generate the bound evidence $v_1 = v_2$ from the bound checking statements that compare $v_1$ with $v_2$, which represent the possible beliefs the programmers may have. For example, in Figure 1, Lines 15 compares $dx$ with $dy$, while Line 21 compares $dx$ with $d$ and $dy$ with $d$, causing the analysis to generate the following bound evidence:

$$dx = dy, dx = d, dy = d$$

How can we take advantage of the generated bound evidence to improve the precision of divide-by-zero detection? We treat the bound evidence as the additional constraints and propagate it to update the representation of $\Gamma$, such that the additional constraints are enforced. For the variable $dz$, from the bound evidence above and the guarded value $(2 \times dx - (dy + d), \text{true})$ for $dz$, we deduce that

$$(2 \times a - (b + c), \text{true}) \in \Gamma(dz), a, b, c \in \{dx, dy, d\}$$

Apparently, $0 \in \Gamma(dx)$ and thus our analysis successfully reports Line 27 of Figure 1 as a divide-by-zero bug. Meanwhile, since no evidence is inferred for $dx$ or $dy$ to be zero (i.e., $0 \notin \Gamma(dx), 0 \notin \Gamma(dy)$), it avoids generating spurious reports at Lines 25-26.

In summary, our analysis encodes the possible values for the variables in the program path-sensitively using a symbolic domain. During the numerical inference process, it attempts to (1) directly find the source evidence and (2) propagate the generated bound evidence to refine the analysis result. Since we apply standard methods in tracking the source evidence, we mainly focus on utilizing the bound evidence in our method. There are two major challenges:

- How to efficiently compute the guarded symbolic value set $\Gamma(v)$ for variable $v$? A naive approach that exhaustively enumerates all execution paths can be too expensive, wasting time exploring irrelevant paths.
- How to propagate bound evidence to refine the symbolic analysis result?

To address the first challenge, we utilize the data dependence analysis to slice away the irrelevant control flow paths for improving the efficiency. For the second challenge, we encode the bound evidence as the additional constraints for $\Gamma$ and enforce these constraints when computing the guarded symbolic value sets.

Program $P$ ::= fun+  
Function fun ::= fun : (\(v_1, \ldots, v_n\)) \rightarrow r  
{\{\}}  
Statement s ::= s1; s2 \mid v := e  
| if (v cmp v_2) s_2 else s_3, cmp \in \{\leq, =\}  
| v := g(d_1, \ldots, a_n) \mid v := \phi(v_1, \ldots, v_n)  
Expression e ::= v \mid c \mid tainted \mid e_1 op e_2, op \in \{+, -, \times, \div\}  

Figure 3: A simple programming language.

Figure 4: Augmented data dependence graph for Figure 1.

4 METHODOLOGY

4.1 Preliminary Definitions

We formulate our analysis using a simple imperative language defined in Figure 3. The language is assumed in the static single assignment form [10] in which each variable has a unique definition, and we denote the SSA phi function by $\phi$. We use $v@l$ to denote a variable $v$ defined at the program location $l$. The language has standard semantics, and we omit a formal definition for brevity.

As mentioned in § 3, we utilize data dependence analysis to compute $\Gamma$ efficiently. Specifically, the analysis operates over a sparse representation of the program called the augmented data dependence graph, defined as follows:

**Definition 4.1.** An augmented data dependence graph $G$ is a directed graph $G = (V, E, L_V, L_E)$ where:

1. $V = V_c \cup V_n \cup V_v \cup V_t$ is the set of nodes. $V_c$ are constant values, $V_n$ are variable definitions, and $V_v$ are nodes corresponding to an arithmetic operation $v_1 \ op \ v_2$, where $v_1, v_2 \in V \setminus V_t$. $V_t$ is the set of tainted input sources.
2. $E \in V \times V$ is the set of edges representing data dependence relations such that $e = (v_1, v_2) \in E$ when $v_1$ is used to define $v_2$. $L_E$ labels each $e \in E$ with a condition $\text{cond}$ under which the value flow can happen.
3. $L_V$ labels each node $n \in V_n$ with $(v@l, \text{cond})$. $\text{cond}$ is the condition for $v$’s definition (i.e., for some execution to reach Line $l$). $L_V$ also labels nodes in $V_v$ with its arithmetic expression $v_1 \ op \ v_2$ and nodes in $V_c$ with its constant $c$.

**Example 4.1.** Figure 4 shows the augmented data dependence graph for the program in Figure 1. An arrow $v_1 \rightarrow v_2$ indicates that $v_1$ is used to define $v_2$ (i.e., $(v_1, v_2) \in E$), and each arrow is labeled with a path condition under which the flow of value can happen (omitted in Figure 4 if it is true). Notice that arithmetic operations are also compactly encoded on the graph by introducing temporary variables $t_1, t_2,$ and $t_3$ to represent intermediate computation results.
The sparse graph representation is used to track relevant data and control dependencies of the concerned variable while skipping irrelevant statements [17]. It has been previously shown effective in detecting null pointer dereference [3, 29], use after free [29], and memory leak [6, 30]. We take inspiration from these works to first utilize data dependence analysis in finding divide-by-zero bugs.

As illustrated in § 3, to achieve precise divide-by-zero detection, our analysis needs to track numerical computations path-sensitively and utilize affirmative evidence to find high confidence bugs. For tracking numerical computations, we apply a symbolic domain \( \Gamma \), formally defined as follows:

**Definition 4.2.** The guarded symbolic value set domain is a mapping \( \Gamma \in V \mapsto \mathcal{P}(Val \times Cond) \), where:

\[
Val \overset{def}{=} \{c, \hat{c} | c \in \mathbb{Z}, v \in V \} \cup \{va_1 \ op \ va_2 | va_1, va_2 \in Val\}
\]

\[
Cond \overset{def}{=} \{\text{set of path conditions}\}
\]

We use \( \hat{c} \) to denote an unknown symbolic value for a node \( v \).

\( \Gamma \) encodes the symbolic values for variables path-sensitively. It maps a node \( v \in G \) to guarded value pairs of the form \( (va, c) \), where \( va \in Val \) is a symbolic expression and \( c \) is the condition under which \( va \) may have the value \( va \). The symbolic expression in \( Val \) is either a basic term (i.e., constant or unknown symbolic value) or a binary operation involving other symbolic expressions.

At a high level, our analysis first constructs the augmented data dependence graph \( G \) utilizing the existing method [29]. We then compute the guarded symbolic value set \( \Gamma(v) \) for a divisor variable \( v \) on-demand. Finally, we use the result \( \Gamma(v) \) in detecting divide-by-zero bugs. Specifically, we compute \( \Gamma(v) \) by building and resolving a system of constraints for \( \Gamma \). To improve the precision, our analysis is path-sensitive and evidence aware, encoding bound evidence as additional constraints. To remain efficient, we utilize data dependence relations in constructing and resolving constraints.

### 4.2 Evidence-based Symbolic Analysis

Given a program \( P \) and a variable \( v \), our evidence-based symbolic analysis shown in Algorithm 1 computes the guarded symbolic value set for \( v \). We build the augmented data dependence graph \( G \) for \( P \) using standard methods (Line 2), generate a system of constraints \( Cons(\Gamma) \) on-demand (Line 4), and solve the constraints to obtain the guarded symbolic value set for \( v \) (Lines 5-6).

§ 4.2.1 demonstrates the process of on-demand constraints generation for a node \( n \in G \), which is the node corresponding to a given variable \( v \). We also generate bound evidence to produce additional constraints, further refining the result and improving the precision. § 4.2.2 discusses the procedure \( solve \) for resolving constraints and obtaining the solution \( \Gamma \), which maps from variables to their guarded symbolic value sets.

#### 4.2.1 Symbolic Constraints Generation

Our analysis generates constraints for \( \Gamma \) from the program’s augmented data dependence graph \( G \). Before presenting the rules for constraint generation, we first define some operators for \( \Gamma \) (cf. Definition 4.2):

**Definition 4.3.** Operators and helper functions definitions for \( \Gamma \):

1. The binary operator \( \overline{op} \) on \( \mathcal{P}(Val \times Cond) \) is defined as:

\[
GV_1 \overline{op} GV_2 = \{\text{simplify}(va_1 \ op \ va_2, c_1 \land c_2) | (va_1, c_1) \in GV_1, (va_2, c_2) \in GV_2\}
\]

2. The logical and operation \( \land \) is extended to add an additional condition to a guarded value set \( S \in \mathcal{P}(Val \times Cond) \): \( cond \land S = \{(va, c \land cond) | (va, c) \in S\} \)

3. \( \cup \) unions two guarded value sets \( S_1, S_2 \in \mathcal{P}(Val \times Cond) \) and combines guarding conditions for the same value: \( S_1 \cup S_2 = \{(va, c_1 \land c_2) | (va, c_1) \in S_1, (va, c_2) \in S_2\} \cup (S_1 \setminus S_2) \cup (S_2 \setminus S_1) \)

Figure 5 lists the inference rules for generating constraints for \( \Gamma \) based on \( G \). We use \( S_1 \leftarrow S_2 \) to denote a weak update to the guarded value set \( S_1 \) (i.e., \( S_1 = S_1 \cup S_2 \)), and \( \overline{op} \) to denote data dependence edges (c.f. Definition 4.1). The first five rules in Figure 5 translate the data dependence relations in \( G \) to constraints for \( \Gamma \). Rule \( init-var \) and Rule \( init-op \) indicate that every node \( n \in V_G \cup V_O \) is associated with an unknown symbolic value \( \hat{n} \), meaning that the value for \( n \) is initially unknown. On the other hand, constant nodes have fixed values (Rule \( init-cst \)). For node encoding operations, the operator \( \overline{op} \) is applied to the guarded symbolic value sets for the incoming nodes (Rule \( \overline{op} \)). For a variable node \( n \in V_G \), we aggregate the guarded symbolic value sets for all the nodes that \( n \) is data dependent on, and track the appropriate path conditions (Rule

1For brevity, the rule only lists the case when both operand nodes are from \( V_O \). Other cases can be transformed to it by introducing temporary variables.
Example 4.2. Finally, the condition associated with relations gives imprecise results. constrained variables when solely relying on the data dependence key challenge is to infer additional information for these under- analysis can still lead to imprecision, as illustrated in § 1. The relations on \( \Gamma \) paths, which are irrelevant in this case. holds on all eight execution paths. The above constraint (4) effec- dition in Figure 1 can affect the value of \( \Gamma (d) \) as discussed in § 3. Therefore, we propose to generate additional constraints on \( \Gamma \) for bound evidence. Specifically, Rule bound-evi of Figure 5 will unify the guarded symbolic value sets for the compared variables \( v_1, v_2 \) to \( \Gamma (v_1) \cup \Gamma (v_2) \). These additional evidence constraints, together with the data dependence constraints generated before will be resolved later to propagate evidence according to the data dependence relations.

Example 4.3. In our motivating example of Figure 1, the two bound checking statements at Line 15 and Line 21 will cause the analysis to generate bound evidence. Applying Rule bound-evi, we obtain the additional constraints:

\[
\begin{align*}
\Gamma (dx), \Gamma (dy) & \leftrightarrow \Gamma (dx) \cup \Gamma (dy) \\
\Gamma (dx), \Gamma (d) & \leftrightarrow \Gamma (dx) \cup \Gamma (d) \\
\Gamma (dy), \Gamma (d) & \leftrightarrow \Gamma (dy) \cup \Gamma (d)
\end{align*}
\]

We know from the above constraints that the symbolic values for \( dx, dy \) are unified. As we will see later, this information refines the constraints generated in Example 4.2 and helps us to infer the evidence for \( dz \) to be zero.

On-demand Constraints Generation. In Algorithm 1, the procedure genConstraints generates a system of constraints \( Cons(\Gamma) \) on-demand from \( G \) and a given node \( n \). Specifically, we perform a backward depth-first traversal on \( G \) starting from \( n \) (Line 9). This effectively computes a slice of \( G \) affecting the value of \( n \). Following the reverse node order of this DFS traversal, we collect constraints using the rules in Figure 5 (Lines 10-12) and finally return these constraints at Line 13. The order ensures that the rules for \( v_1 \) come before \( v_2 \) if \( v_1 \) is used to define \( v_2 \)’s value, thus speeding up the convergence of the solving process introduced later, similar to data flow analysis [2].

4.2.2 Constraint Resolution. Given a set of constraints \( Cons(\Gamma) \), the procedure solve in Algorithm 1 computes its solution, which is a mapping \( \Gamma \) from nodes to guarded symbolic value sets. At a high level, \( \Gamma \) starts with an initial state mapping any node to the empty set (Line 15) and is iteratively updated according to the constraints \( Cons(\Gamma) \). Notice that each constraint in \( Cons(\Gamma) \) is built from the rules in Figure 5 and, thus, has the syntactic form \( \Gamma (o) \leftrightarrow S \). For updating \( \Gamma (o) \), we calculate the result of the set operation encoded by \( S \), denoted by \( eval(S, \Gamma) \) at Line 18. The calculation result is combined with the old value \( \Gamma (o) \) using the \( \cup \) operator (c.f. Definition 4.3) to produce the updated value set \( S’ \) for \( o \) (Lines 18-19), since \( \leftrightarrow \) represents a weak update. The algorithm finishes when no constraint causes an update to \( \Gamma \) anymore (Lines 16-21).

Example 4.4. For the constraints (4)-(5) shown in Example 4.2, Lines 18-19 of the solve procedure compute the guarded symbolic

As motivated in § 1, a bound checking statement if \( v_1 \ cmp \ v_2 \) reveals programmer’s beliefs about the values of the checked variables: a comparison between \( v_1 \) and \( v_2 \) suggests that the programmer may believe that \( v_1 = v_2 \), which we call a bound evidence. Bound evidence can aid in providing additional information for the under-constrained variables, e.g., it is used to infer \( 0 \in \Gamma (dz) \) as discussed in § 3. Therefore, we propose to generate additional constraints on \( \Gamma \) for bound evidence. Specifically, Rule bound-evi of Figure 5 will unify the guarded symbolic value sets for the compared variables \( v_1 \) and \( v_2 \) to \( \Gamma (v_1) \cup \Gamma (v_2) \). These additional evidence constraints, together with the data dependence constraints generated before will be resolved later to propagate evidence according to the data dependence relations.

Example 4.3. In our motivating example of Figure 1, the two bound checking statements at Line 15 and Line 21 will cause the analysis to generate bound evidence. Applying Rule bound-evi, we obtain the additional constraints:

\[
\begin{align*}
\Gamma (dx), \Gamma (dy) & \leftrightarrow \Gamma (dx) \cup \Gamma (dy) \\
\Gamma (dx), \Gamma (d) & \leftrightarrow \Gamma (dx) \cup \Gamma (d) \\
\Gamma (dy), \Gamma (d) & \leftrightarrow \Gamma (dy) \cup \Gamma (d)
\end{align*}
\]

We know from the above constraints that the symbolic values for \( dx, dy \) are unified. As we will see later, this information refines the constraints generated in Example 4.2 and helps us to infer the evidence for \( dz \) to be zero.

On-demand Constraints Generation. In Algorithm 1, the procedure genConstraints generates a system of constraints \( Cons(\Gamma) \) on-demand from \( G \) and a given node \( n \). Specifically, we perform a backward depth-first traversal on \( G \) starting from \( n \) (Line 9). This effectively computes a slice of \( G \) affecting the value of \( n \). Following the reverse node order of this DFS traversal, we collect constraints using the rules in Figure 5 (Lines 10-12) and finally return these constraints at Line 13. The order ensures that the rules for \( v_1 \) come before \( v_2 \) if \( v_1 \) is used to define \( v_2 \)’s value, thus speeding up the convergence of the solving process introduced later, similar to data flow analysis [2].

4.2.2 Constraint Resolution. Given a set of constraints \( Cons(\Gamma) \), the procedure solve in Algorithm 1 computes its solution, which is a mapping \( \Gamma \) from nodes to guarded symbolic value sets. At a high level, \( \Gamma \) starts with an initial state mapping any node to the empty set (Line 15) and is iteratively updated according to the constraints \( Cons(\Gamma) \). Notice that each constraint in \( Cons(\Gamma) \) is built from the rules in Figure 5 and, thus, has the syntactic form \( \Gamma (o) \leftrightarrow S \). For updating \( \Gamma (o) \), we calculate the result of the set operation encoded by \( S \), denoted by \( eval(S, \Gamma) \) at Line 18. The calculation result is combined with the old value \( \Gamma (o) \) using the \( \cup \) operator (c.f. Definition 4.3) to produce the updated value set \( S’ \) for \( o \) (Lines 18-19), since \( \leftrightarrow \) represents a weak update. The algorithm finishes when no constraint causes an update to \( \Gamma \) anymore (Lines 16-21).

Example 4.4. For the constraints (4)-(5) shown in Example 4.2, Lines 18-19 of the solve procedure compute the guarded symbolic

As motivated in § 1, a bound checking statement if \( v_1 \ cmp \ v_2 \) reveals programmer’s beliefs about the values of the checked variables: a comparison between \( v_1 \) and \( v_2 \) suggests that the programmer may believe that \( v_1 = v_2 \), which we call a bound evidence. Bound evidence can aid in providing additional information for the under-constrained variables, e.g., it is used to infer \( 0 \in \Gamma (dz) \) as discussed in § 3. Therefore, we propose to generate additional constraints on \( \Gamma \) for bound evidence. Specifically, Rule bound-evi of Figure 5 will unify the guarded symbolic value sets for the compared variables \( v_1 \) and \( v_2 \) to \( \Gamma (v_1) \cup \Gamma (v_2) \). These additional evidence constraints, together with the data dependence constraints generated before will be resolved later to propagate evidence according to the data dependence relations.

Example 4.3. In our motivating example of Figure 1, the two bound checking statements at Line 15 and Line 21 will cause the analysis to generate bound evidence. Applying Rule bound-evi, we obtain the additional constraints:

\[
\begin{align*}
\Gamma (dx), \Gamma (dy) & \leftrightarrow \Gamma (dx) \cup \Gamma (dy) \\
\Gamma (dx), \Gamma (d) & \leftrightarrow \Gamma (dx) \cup \Gamma (d) \\
\Gamma (dy), \Gamma (d) & \leftrightarrow \Gamma (dy) \cup \Gamma (d)
\end{align*}
\]

We know from the above constraints that the symbolic values for \( dx, dy \) are unified. As we will see later, this information refines the constraints generated in Example 4.2 and helps us to infer the evidence for \( dz \) to be zero.
value set for $dz$ as follows:

$$\Gamma(dz) = \{ (\hat{dz}, true) \} \cup \{ (2, true) \} \times \{ (\hat{dz}, true) \} \supseteq \{ (\hat{dy}, true) \} \supset \{ (\hat{d}, true) \} \}

$$

$$= \{ (2 \times \hat{dx} - (\hat{dy} + \hat{d}), true), (\hat{dz}, true) \}

$$

Constraints (6)-(8) from Example 4.3 further refines the result to:

$$\Gamma(dz) = \{ (2 \times \hat{a} - (\hat{b} + \hat{c}), true), (\hat{dz}, true) \} | a, b, c \in \{ dx, dy, d \}

$$

Since $(0, true) \in \Gamma(dz)$, we have found the evidence for the variable $dz$ to trigger divide-by-zero.

The above example shows the advantage of generating and resolving constraints from $G$ according to the data dependence relations. Our constraints are compact, i.e., resolving the constraints does not require reasoning about the irrelevant control flow paths. In contrast, an exhaustive approach such as symbolic execution needs to enumerate all eight paths in the function move of Figure 1, just to figure out the guarded symbolic value set for $dz$:

<table>
<thead>
<tr>
<th>Path</th>
<th>$dz$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$dx \geq dy \land dx \geq d \land dy \geq d \land 2 \times dx - (dy + d)$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$dx &lt; dy \land dx &lt; d \land dy &lt; d \land 2 \times dx - (dy + d)$</td>
</tr>
</tbody>
</table>

The constraints built by the exhaustive path enumeration would have been overly redundant:

$$\Gamma(dz) \leftarrow \text{trans}((\Gamma(2) \times \Gamma(dx) \supset (\Gamma(dy) \supset \Gamma(d), c_1)))

\ldots

\Gamma(dz) \leftarrow \text{trans}((\Gamma(2) \times \Gamma(dx) \supset (\Gamma(dy) \supset \Gamma(d), c_8)))

$$

Since $c_1 \lor \cdots \lor c_8 \equiv \text{true}$, the above constraints are equivalent to our generated constraint (4) in Example 4.2.

One noteworthy point about the sole procedure is that it can produce a value set $S'$ with a large size. To keep the analysis tractable, we join the additional values for variable $v$ to its sound abstraction $\delta$ when the size of $\Gamma(\delta)$ has reached a predefined threshold. In this work, we choose the threshold to be 20 by experience.

### 4.3 Divide-by-Zero Bug Detection

Our system, Wit, for precise divide-by-zero detection is shown in Algorithm 2. Specifically, given a program $P$, a divisor variable $v$, and the location $I$ of the division instruction, Algorithm 2 returns whether a divide-by-zero bug may happen at $l$. We first call symbolicAnalysis of Algorithm 1 using $P$ and $v$ as arguments to obtain the guarded symbolic value set $S$ for $v$ (Line 2). Divide-by-zero detection is achieved by enumerating each guarded value $(va, cond)$ in $S$ to check for the bug condition.

To report a divide-by-zero bug, we adhere to the criteria C1* - C2* mentioned in § 1. First, according to C1*, the conjunction of the path condition and the error condition should be satisfiable. In Algorithm 2, the path condition is denoted by $pc$, consisting of the condition $C$ for reaching the division instruction and the guarding condition $cond$ for the checked value $va$ (Line 5). The error condition is simply $va = 0$. Therefore, Line 6 checks the satisfiability of $pc \land va = 0$ for determining C1*. Second, according to C2*, the concerned symbolic value $va$ should have the affirmative evidence to be zero:

$$\text{if } SAT(pc \land va = 0) \text{ then }
\text{ if tainted}(va) \text{ then return true }
\text{ else if } \text{ UNSAT}(pc \land va \neq 0) \text{ then return true }
\text{ end}
\text{ return false}
$$

Figure 6: Criteria for propagating the tainted flag.

- **Source evidence**: If $va$ comes from the tainted input, we can report a divide-by-zero bug (Lines 7-8). For taint detection, we can apply any existing taint analysis to compute the tainted nodes on $G$. Specifically, we use Rule $taint-evi$ in Figure 6: a node is tainted if it is data dependent on the tainted input and the condition of the value flow is satisfiable.

- **Bound evidence**: Otherwise, we report a divide-by-zero bug if $va$ must be zero under the condition $pc$ (Lines 9-10). This can happen for two reasons: either the symbolic value $va$ is literally constant zero, or it is forced to be zero by the condition $pc$ (e.g., when $pc = -1 < va < 1$). Recall that the bound evidence is encoded as additional constraints in Cons($\Gamma$) during the analysis and can implicitly enforce $va$ to be zero, it is thus more likely to satisfy the must query.

For the program in Figure 1, Algorithm 1 deduces that $(0, true) \in \Gamma(dz)$ as illustrated by Example 4.4. Thus, Lines 9-10 of Algorithm 2 report a divide-by-zero bug for the variable $dz$.

### 5 EVALUATION

We implement Wit based on LLVM [20]. Similar to previous works on static bug detection [3, 29, 30, 32], we unroll each loop once in control flow graphs and call graphs. Our experimental evaluation is designed to answer the following research questions:

- **RQ3**: The effectiveness of Wit. Particularly,
  - **RQ3.1**: How does our intuition of evidence-based reasoning affect the analysis precision?
  - **RQ3.2**: Compared to the conventional evidence agnostic method, how many bugs are missed by Wit? What are the reasons for missing them?
  - **RQ3.3**: Can Wit detect real-world divide-by-zero bugs?
Table 2: Selected projects for evaluation.

<table>
<thead>
<tr>
<th>Project</th>
<th>Loc</th>
<th>#Div/KLoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>masscan</td>
<td>34k</td>
<td>5.4</td>
</tr>
<tr>
<td>goaccess</td>
<td>53k</td>
<td>1.1</td>
</tr>
<tr>
<td>libuv</td>
<td>59k</td>
<td>0.8</td>
</tr>
<tr>
<td>redis</td>
<td>131k</td>
<td>5.0</td>
</tr>
<tr>
<td>git</td>
<td>226k</td>
<td>4.5</td>
</tr>
<tr>
<td>vim</td>
<td>354k</td>
<td>1.8</td>
</tr>
<tr>
<td>ImageMagick</td>
<td>382k</td>
<td>6.6</td>
</tr>
<tr>
<td>openssl</td>
<td>465k</td>
<td>4.1</td>
</tr>
<tr>
<td>systemd</td>
<td>600k</td>
<td>5.0</td>
</tr>
<tr>
<td>php</td>
<td>1,012k</td>
<td>1.3</td>
</tr>
<tr>
<td>gdb</td>
<td>1,932k</td>
<td>1.6</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>15,164k</td>
<td>2.1</td>
</tr>
</tbody>
</table>

- **RQ4**: How does Wit perform compared with existing divide-by-zero detectors?

5.1 Experimental Setup

**Subjects.** We have selected 12 open source C/C++ projects to perform the evaluation, shown in Table 2. Our selection criteria are as follows:

- **Popularity**: The selected projects are popular (e.g., have at least 10K stars on GitHub) and actively maintained.
- **Generality**: The projects cover different sizes (ranging from tens of thousands to tens of millions of lines of code) and functionalities (including operating system, image processing, database system, network library, etc.)
- **Intensive use of division instruction**: The projects perform division operation intensively, e.g., they contain 3.3 division instructions every 1k lines of code on average.

**Environment.** The experiments were performed on a computer with two 20 core processors Intel(R) Xeon(R) CPU E5-2698 v4@2.20GHz and 256GB physical memory running Ubuntu-16.04.

**Open Data.** The results of our empirical study and confirmed bugs detected by Wit are available at the link: https://github.com/yiyuaner/ICSE-2022-Wit-data.

5.2 Effectiveness of Divide-by-Zero Detection

To study the effectiveness of the evidence-based symbolic analysis algorithm, we compare Wit with its variant Wit$^{-}$. Wit$^{-}$ is the evidence agnostic path-sensitive method. In Wit$^{-}$, we remove Rule $bound$-evi from Figure 5 and report an error whenever the condition is satisfiable, i.e., whenever the check at Line 6 of Algorithm 2 passes. All other aspects of Wit are the same as Wit.

Table 3 shows the experiment result. We compare the total number of bug reports, the false positive rate, and the analysis time for Wit and Wit$^{-}$. For each variant on a specific project, we examine its list of output reports and stop if the number of false positives has exceeded 200 (when this happens, we use “NA” to denote its false positive rate). Wit effectively reports 95 divide-by-zero bugs (72 of them are true positives) among 12 real-world applications, with a low average false positive rate of 22%. In contrast, Wit$^{-}$ has an average false positive rate of 86%.

Answer to RQ3.1: The precision of Wit greatly outperforms its no-evidence counterpart, proving the significance of affirmative evidence for precise divide-by-zero detection.

Table 4 shows the distribution of true positives reported by Wit (the column “Total”) into Class Src and Class Bd. The column “Missed” shows the number of true positives reported by Wit$^{-}$ but missed by Wit.

Table 5 shows the distribution of true positives reported by Wit$^{-}$ into Class Src and Class Bd. The column “Missed” shows the number of true positives detected by Wit but missed by Wit$^{-}$.

Table 5: Divide-by-zero bugs confirmed by developers.

<table>
<thead>
<tr>
<th>Project</th>
<th>Total</th>
<th>Src</th>
<th>Bd</th>
<th>Missed</th>
</tr>
</thead>
<tbody>
<tr>
<td>masscan</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>goaccess</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libuv</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>redis</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>git</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>vim</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ImageMagick</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>openssl</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>systemd</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>php</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>gdb</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>43</td>
<td>36</td>
<td>10</td>
<td>NA</td>
</tr>
</tbody>
</table>

**git** Linux gdb ImageMagick goaccess libuv openssl vim systemd

with 1 1 1 1 1 1 1.
which is relatively low considering its drastic precision improvement. Most of the missed bugs are due to the incomplete modeling of the taint sources, i.e., the analysis fails to identify certain APIs as the taint sources, thus missing the source evidence and fails to report certain bugs. Wit currently models some common taint sources from the standard C library but each target program may contain its specific taint sources that we do not know beforehand.

We report the detected true positives and 14 bugs have been confirmed by the developers, as shown in Table 5. Figure 7 shows a real divide-by-zero bug Wit discovered from the Linux kernel. The function queue_index uses nr_queues as a divisor, which comes from the unknown input argument qmap->nr_queues. Since the analysis generates a bound evidence q\*nr_queues from Line 7 of Figure 7 and q is assigned zero value in Line 5, it deduces that nr_queues may be zero and reports a divide-by-zero bug.

Answer to RQ3.2: Wit only misses a small proportion of bugs compared with its no-evidence counterpart Wit−, mainly due to its incomplete modeling of the taint sources.

Answer to RQ3.3: Wit is capable of detecting real divide-by-zero bugs: it has found 72 divide-by-zero bugs, 14 of which have been confirmed by the developers.

5.3 Comparison with Existing Detectors

We compare Wit with three competing tools, namely Crab [18], Infer [5], and Clang Static Analyzer (CSA). Crab adopts numerical abstract interpretation, Infer takes the bi-abduction theorem proving technique, and CSA performs local symbolic execution.

Comparing with Crab. We instrument the program by asserting the divisor variable to be non-zero and use Crab to verify the inserted assertions with the interval domain [8]. When Crab fails to verify the safety of a division instruction, it emits a warning as a potential divide-by-zero bug. We cannot directly evaluate the precision of Crab because it currently does not provide an interface to map its verification result to source code location for reports examination. Instead, we record the number of warnings it generates and the ratio of its generated warnings over the total number of checked assertions. Intuitively, a tool reporting warnings too often (e.g., consider a tool that reports a bug for half of the division instructions in the program) maybe unusable in the industrial setting [4, 7]. However, it should be noted that this only loosely relates to the precision and our comparison has the caveat of possibly underestimating the effectiveness of Crab.

The analysis result for Crab is shown in Table 6. On average, Crab produces 212 bug reports per project, failing to verify 44% of the checked assertions on average. Such a large number of bug reports will impede the user from adopting the tool in divide-by-zero detection. In contrast, Wit reports 95 divide-by-zero bugs for the 12 projects in total, with an average false positive rate of 22%.

Comparing with Infer and CSA. Table 7 shows the results of Infer and CSA. From the data, we conclude that:

1. Infer generates zero reports on six projects and fails on one project, while CSA generates zero reports on five projects. Thus, they both have missed many true positives detected

<table>
<thead>
<tr>
<th>Project</th>
<th># of reports</th>
<th>FP rate</th>
<th>Analysis time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>Infer</td>
<td>CSA</td>
<td>Infer</td>
</tr>
<tr>
<td>massscan</td>
<td>74</td>
<td>66%</td>
<td>NA</td>
</tr>
<tr>
<td>goaccess</td>
<td>9</td>
<td>53%</td>
<td>NA</td>
</tr>
<tr>
<td>libuv</td>
<td>6</td>
<td>27%</td>
<td>NA</td>
</tr>
<tr>
<td>redis</td>
<td>542</td>
<td>36%</td>
<td>200m30s</td>
</tr>
<tr>
<td>git</td>
<td>128</td>
<td>58%</td>
<td>175m49s</td>
</tr>
<tr>
<td>vim</td>
<td>174</td>
<td>42%</td>
<td>111m29s</td>
</tr>
<tr>
<td>ImageMagick</td>
<td>102</td>
<td>27%</td>
<td>389m49s</td>
</tr>
<tr>
<td>openssl</td>
<td>111</td>
<td>43%</td>
<td>68m41s</td>
</tr>
<tr>
<td>systemd</td>
<td>925</td>
<td>43%</td>
<td>141m30s</td>
</tr>
<tr>
<td>php</td>
<td>101</td>
<td>13%</td>
<td>68m41s</td>
</tr>
<tr>
<td>gdb</td>
<td>165</td>
<td>57%</td>
<td>20m10s</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>Crash</td>
<td>51%</td>
<td>Crash</td>
</tr>
</tbody>
</table>

Table 7: Divide-by-zero detection results for Infer and CSA. NA denotes the false positive rate when no bug is reported.
by WIT. The relatively low recall is due to practical considerations for the tools, such as the limited capability for detecting cross-file bugs and inherent approximations in their algorithms.

(2) Over the projects where the number of bug reports is non-zero, Infer has an average false positive rate of 64%, and CSA has an average false positive rate of 55%.

(3) WIT runs slower than Infer and CSA in projects with large sizes. This is mainly because we utilize SMT solving to be fully path-sensitive. However, WIT finishes within 7.5 hours in all projects, which is an acceptable performance given its precision improvement.

Answer to RQ4: compared with existing divide-by-zero detectors, WIT is significantly more precise and sometimes even detects more divide-by-zero bugs.

6 RELATED WORK

Abstract Interpretation. In abstract interpretation [8], many abstract domains have been designed to verify numerical properties, such as the octagon [25] and polyhedra domain [9]. Recent works have tried to balance between the precision and cost of abstract interpretation. Oh et al. [27] utilize an impact pre-analysis to apply context-sensitivity selectively, while LAIT [19] identifies and removes redundant constraints in numerical analysis to improve speed without hurting precision too much. Mansur et al. [24] propose to automatically tailor the configurations of abstract interpreters according to the code under analysis and resource constraints. Our work improves the precision from a different angle by finding the affirmative evidence for triggering the bug.

Symbolic Analysis for Numerical Bug Detection. To the best of our knowledge, we are the first to design a symbolic analysis algorithm targeting the problem of divide-by-zero detection. Previous works mainly focus on applying symbolic analysis in static buffer overflow detection and integer overflow detection. Li et al. [22] adopt a simple symbolic value representation and filter out irrelevant dependencies during the symbolic value computation. Marple [21] performs on-demand backward symbolic execution, starting from a buffer access statement and categorizing program paths to prioritize bug reports. SIFT [23] utilizes precondition inference in computing sound input filters to nullify integer overflow errors associated with critical program sites such as memory allocation or block copy sites. These works adopt path-sensitive symbolic domains similar to ours but do not discuss the imprecision problem brought by the under-constrained variables or how to address it, which is the key contribution of this work.

Reasoning About Programmer’s Beliefs. Engler et al. [16] first propose to infer from code about the programmers’ beliefs and detect potential bugs by spotting inconsistency of such beliefs. Dillig et al. [12] formalize the intuition of [16] in a framework for detecting semantic inconsistency, where they design a null pointer dereference checker. The belief propagation is carried out using type inference: e.g., a pointer inferred with a “possibly NULL” type should not be used in a context that requires a “not NULL” type. However, divide-by-zero detection requires an infinite analysis domain and existing approaches for type-state properties are not applicable.

Inspired by the success of utilizing beliefs in previous works on static analysis, we firstly introduce the use of beliefs to address the imprecision problem brought by under-constrained variables in divide-by-zero detection. We generalize the idea of “programmer’s belief” in analyzing numerical computation by generating bound evidence from the bound checking statements in the program. Although we do not directly infer inconsistency, the evidence we generate serves as additional constraints, which greatly improves the overall precision of the analysis. The imprecision problem caused by under-constrained variables in static analysis is observed in [13], where they tackle the problem with constraints rewriting. Since they do not consider numerical property in the constraints and assume a finite analysis domain, their approach is not applicable in our scenario.

7 CONCLUSION

We have proposed WIT, a static analysis method to find divide-by-zero bugs with affirmative evidence. The analysis looks for affirmative evidence, namely source evidence and bound evidence, to find divide-by-zero bugs with high confidence. It has been shown effective in detecting divide-by-zero bugs precisely in large-scale real-world software.

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. The authors are supported by the RGC16206517, IT/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei. Peisen Yao is the corresponding author.

REFERENCES

Lattice Model for Static Analysis of Programs by Construction or Approxi-
mation of Fixpoints. In Proceedings of the 4th ACM SIGPLAN-SIGACT Sym-
posium on Principles of Programming Languages (Los Angeles, California) (POPL
’77). Association for Computing Machinery, New York, NY, USA, 238–252. https:
//doi.org/10.1145/512950.512973
Restraints among Variables of a Program. In Proceedings of the 5th ACM SIGPLAN-
SIGACT Symposium on Principles of Programming Languages (Tucson, Arizona)
(POPL ’78). Association for Computing Machinery, New York, NY, USA, 84–96. https:
//doi.org/10.1145/512760.512779
//doi.org/10.1145/115372.115380
Partial Java Programs. In Proceedings of the 23rd ACM SIGPLAN Conference on
Object-Oriented Programming Systems Languages and Applications (Nashville, TN, USA)
(OOPSLA ’08). Association for Computing Machinery, New York, NY, USA, 313–328. https:
//doi.org/10.1145/1449764.1449790
[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2007. Static Error Detection Using
Semantic Inconsistency Inference. In Proceedings of the 26th ACM SIGPLAN
Conference on Programming Language Design and Implementation (San Diego, California, USA)
(PLDI ’07). Association for Computing Machinery, New York, NY, USA, 435–445. https:
//doi.org/10.1145/1250734.1250784
[13] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Reasoning about the Unknown
1145/1787234.1787259
[14] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated Error Diagnosis
Using Abductive Inference. SIGPLAN Not. 47, 6 (June 2012), 181–192. https:
//doi.org/10.1145/2254087
Scaling Static Analyses at Facebook. Commun. ACM 62, 8 (July 2019), 62–70. https:
//doi.org/10.1145/3338112
Bugs as Deviant Behavior: A General Approach to Inferring Errors in
Systems Principles (SOSP ’01). Association for Computing Machinery, New York,
NY, USA, 57–72. https://doi.org/10.1145/502034.502041
9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24044
The SealHorn Verification Framework. In Computer Aided Verification - 27th
ing Fast and Precise Numerical Analysis. In Proceedings of the 41st ACM SIGPLAN
Conference on Programming Language Design and Implementation (London, UK)
program analysis and transformation. In International Symposium on Code
Generation and Optimization, 2004. CGO 2004. 75–86. https://doi.org/10.1109/
CGO.2004.1281665
Buffer Overflow Detector. In Proceedings of the 16th ACM SIGSOFT International
Symposium on Foundations of Software Engineering (Atlanta, Georgia) (SIGSOFT
’08/FSE-16). Association for Computing Machinery, New York, NY, USA, 372–282. https:
//doi.org/10.1145/1453101.1453137
[22] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2010. Practical and Effective
Symbolic Analysis for Buffer Overflow Detection. In Proceedings of the Eighteenth
ACM SIGSOFT International Symposium on Foundations of Software Engineering
(Santa Fe, New Mexico, USA) (FSE ’10). Association for Computing Machinery, New
Sound Input Filter Generation for Integer Overflow Errors. In Proceedings of the
41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages
(San Diego, California, USA) (POPL ’14). Association for Computing Machinery,
New York, NY, USA, 439–452. https://doi.org/10.1145/2535838.2535888
[24] Muhammad Numair Mansur, Benjamin Mariano, Maria Christakis, Jorge A. Navas,
and Valentín Wüthrich. 2021. Automatically Tailoring Abstract Interpretation to
Custom Usage Scenarios. In Computer Aided Verification - 33rd International
Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture
(Eds.). Springer, 777–800. https://doi.org/10.1007/978-3-030-81688-9_36
[26] Antoine Miné. 2007. A New Numerical Abstract Domain Based on Difference-
0703073
Selective Context-Sensitivity Guided by Impact Pre-Analysis. In Proceedings of the
35th ACM SIGPLAN Conference on Programming Language Design and Implemen-
tation (Edinburgh, United Kingdom) (PLDI ’14). Association for Computing Ma-
chinery, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318
[28] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Ex-
icution: Correctness Checking for Real Code. In 24th USENIX Security Sym-
posium (USENIX Security 15) USENIX Association, Washington, D.C., 49–
64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/
presentation/ramos
[29] Qingkai Shi, Xiao Xiao, Rongxian Wu, Jingguo Zhou, Gang Fan, and Charles
Lines of Code. In Proceedings of the 39th ACM SIGPLAN Conference on Program-
Association for Computing Machinery, New York, NY, USA, 693–706. https:
//doi.org/10.1145/2594291.2594318
[30] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static Memory Leak Detection Us-
Symposium on Software Testing and Analysis (Minneapolis, MN, USA) (ISSTA
2012). Association for Computing Machinery, New York, NY, USA, 254–264. https:
//doi.org/10.1145/2338965.2336784
[31] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman.
ACM SIGPLAN Conference on Programming Language Design and Implementation
(Dublin, Ireland) (PLDI ’09). Association for Computing Machinery, New York,
NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486
Satisfiability. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on
Principles of Programming Languages (Long Beach, California, USA) (POPL ’05).
Association for Computing Machinery, New York, NY, USA, 351–363. https:
//doi.org/10.1145/1040385.1040334