Do I Use the Wrong Definition?

DefUse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs *

Yao Shi
Tsinghua University
shiyao00@mails.tsinghua.edu.cn

Soyeon Park
University of California, San Diego
soyeon@cs.ucsd.edu

Zuoning Yin
University of Illinois at Urbana-Champaign
zyin2@uiuc.edu

Shan Lu
University of Wisconsin-Madison
shanlu@cs.wisc.edu

Yuanyuan Zhou
University of California, San Diego
yyzhou@cs.ucsd.edu

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

Weimin Zheng
Tsinghua University
zwm-dcs@tsinghua.edu.cn

Abstract

Software bugs, such as concurrency, memory and semantic bugs, can significantly affect system reliability. Although much effort has been made to address this problem, there are still many bugs that cannot be detected, especially concurrency bugs due to the complexity of concurrent programs. Effective approaches for detecting these common bugs are therefore highly desired.

This paper presents an invariant-based bug detection tool, DefUse, which can detect not only concurrency bugs (including the previously under-studied order violation bugs), but also memory and semantic bugs. Based on the observation that many bugs appear as violations to programmers' data flow intentions, we introduce three different types of definition-use invariants that commonly exist in both sequential and concurrent programs. We also design an algorithm to automatically extract such invariants from programs, which are then used to detect bugs. Moreover, DefUse uses various techniques to prune false positives and rank error reports.

We evaluated DefUse using **sixteen** real-world applications with twenty real-world concurrency and sequential bugs. Our results show that DefUse can effectively detect 19 of these bugs, including 2 new bugs that were never reported before, with only a few false positives. Our training sensitivity results show that, with the benefit of the pruning and ranking algorithms, DefUse is accurate even with insufficient training.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability; D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics

General Terms Reliability

Keywords Concurrency Bug, Sequential Bug, Atomicity Violation, Order Violation

1. Introduction

1.1 Motivation

As software has grown in size and complexity, the difficulty of finding and fixing bugs has increased. Inevitably, many bugs leak into deployed software, contributing to up to 27% of system failures [22].

Among the different types of bugs, concurrency bugs are the most notorious, due to their non-deterministic nature. Unlike sequential bugs, concurrency bugs depend not only on inputs and execution environments, but also on thread-interleaving and other timing-related events that are to be

* This work is supported by IBM CAS Canada research fellowship "May-Happen-In-Parallel Analysis for POSIX Thread" and Intel-MOE joint research award "Program Analysis for Parallel Applications".

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

manifested [3, 18, 27, 30, 34, 40]. This makes them hard to be exposed and detected during in-house testing. Although concurrency bugs may appear less frequently than sequential bugs, they can cause more severe consequences, such as data corruption, hanging systems, or even catastrophic disasters (e.g., the Northeastern Electricity Blackout Incident [35]). With the pervasiveness of multi-core machines and concurrent programs, this problem is becoming more and more severe.

Recent research works have made significant advances in detecting concurrency bugs, especially for data races [10, 23, 34, 39] and atomicity violations [1] [9, 11, 19, 33, 42, 44], but order violation bugs have been neglected. An order violation occurs if a programming assumption on the order of certain events is not guaranteed during the implementation [18] [2]. According to a recent real-world concurrency bug characteristic study [18], order violations account for *one third* of all non-deadlock concurrency bugs.

Figure 1(a) shows an order violation bug in HTTrack. In this example, the programmer incorrectly assumes that $S2$ in Thread 1 is always executed after $S3$ in Thread 2 (probably due to some misunderstanding in the thread creation process and scheduling). Unfortunately, this order assumption is not guaranteed in the implementation. In some cases, $S2$ may be executed before $S3$ and results in a null-pointer parameter, which may lead to a null-pointer dereference, crashing the program later.

Besides concurrency bugs, semantic bugs are another form of bugs that is hard to detect. To date, only a few tools have been proposed to detect semantic bugs due to their versatility and lack of general patterns. Figure 2(a) shows such an example from `gzip`. This bug is caused by the missing reassignment of variable `ifd` before $S2$. With certain inputs, `ifd` can incorrectly reuse the value set by $S3$, causing the program to misbehave. This bug is program-specific and cannot be detected easily.

Finally, memory corruption bugs such as buffer overflow and dangling pointer are also important since they can be exploited by malicious users. Many tools have been built for detecting those bugs but they each target only a special subtype of bugs. Figure 2(c) and 2(d) show two real-world memory corruption bugs.

1.2 Commonality among Bug Types

Interestingly, regardless of the difference between these bugs' root causes, many of them share a common characteristic: when triggered, they usually are followed by an incorrect data flow, i.e., a read instruction uses the value from an unexpected definition [3]. We refer to such a definition as an *incorrect* definition. Figure 1 and 2 use eight real-world bugs to demonstrate such commonality. Although these bugs have different root causes, when they are manifested, they invariably result in a read (highlighted in each figure) that uses a value from an incorrect definition.

For example, in the HTTrack example (Figure 1(a)), during correct execution, $S2$ always uses the definition from $S3$. However, in a buggy run, $S2$ uses the definition from $S1$, resulting in a null-pointer dereference later. Similarly, in the atomicity violation bug example from Apache (Figure 1(b)), during correct execution, $S2$ always uses the definition from $S1$. However, in a buggy run, $S3$ is interleaved between $S1$ and $S2$. Thus $S2$ uses the definition from $S3$ which causes both threads to call cleanup_cash_obj(). This will lead to a null-pointer dereference inside this function. The other two bugs in Figure 1(c)(d) are also similar.

Some sequential bugs also share a similar characteristic. For example, in the semantic bug example shown in Figure 2(a), in correct runs, $S2$ always uses the definition from $S1$. However, when the bug manifests, $S2$ can read the definition from $S3$ from the last loop iteration. The other three bugs shown in Figure 2, including one semantic bug and two memory bugs, are similar.

The above commonality indicates that, *if we can detect such incorrect definition-use data flow, it is possible to catch these bugs, regardless of their different root causes.*

In order to detect incorrect definition-use flow, we first need to determine what definitions a correct instruction should use. Unfortunately, it is not easy to get such information. Firstly, it is tedious to ask programmers to provide such information (if they were aware of such information consciously, they would not have introduced such bugs in the first place). Secondly, it is also hard to extract such information by a static analysis. Though an advanced static analysis could reveal all possible definitions that an instruction could use, it cannot differentiate the correct definitions from incorrect ones. For example, in Figure 2(a), static analysis tools would report the definitions from both $S1$ and $S3$ can reach $S2$, but it cannot identify $S3$ as an incorrect definition for $S1$.

In this paper, we use a different but complementary technique to address this problem for both concurrent and sequential programs written in unsafe programming languages.

1.3 Our Contributions

The technique proposed in this paper is a new class of program invariants called **definition-use invariants**, which can capture the inherent relationships between definitions and

[1] An atomicity violation bug is a software error occurring when a set of operations/accesses that is supposed to be atomic is not protected with appropriate synchronizations.

[2] Even though order violations can be fixed in ways similar to atomicity violation bugs (i.e., by adding locks), their root causes are different. Order is not essential in atomicity violation, as long as there is no intervening remote access breaking the assumed atomicity.

[3] When a variable, `v`, is on the left-hand side of an assignment statement, this statement is a **definition** of `v`. If variable `v` is on the right-hand-side of a statement, `v` has a **use** at this statement. A *reaching definition* for a given instruction is another instruction, the target variable of which reaches the given instruction without an intervening assignment [1]. For simplicity, we refer to a use's reaching definition as its definition.

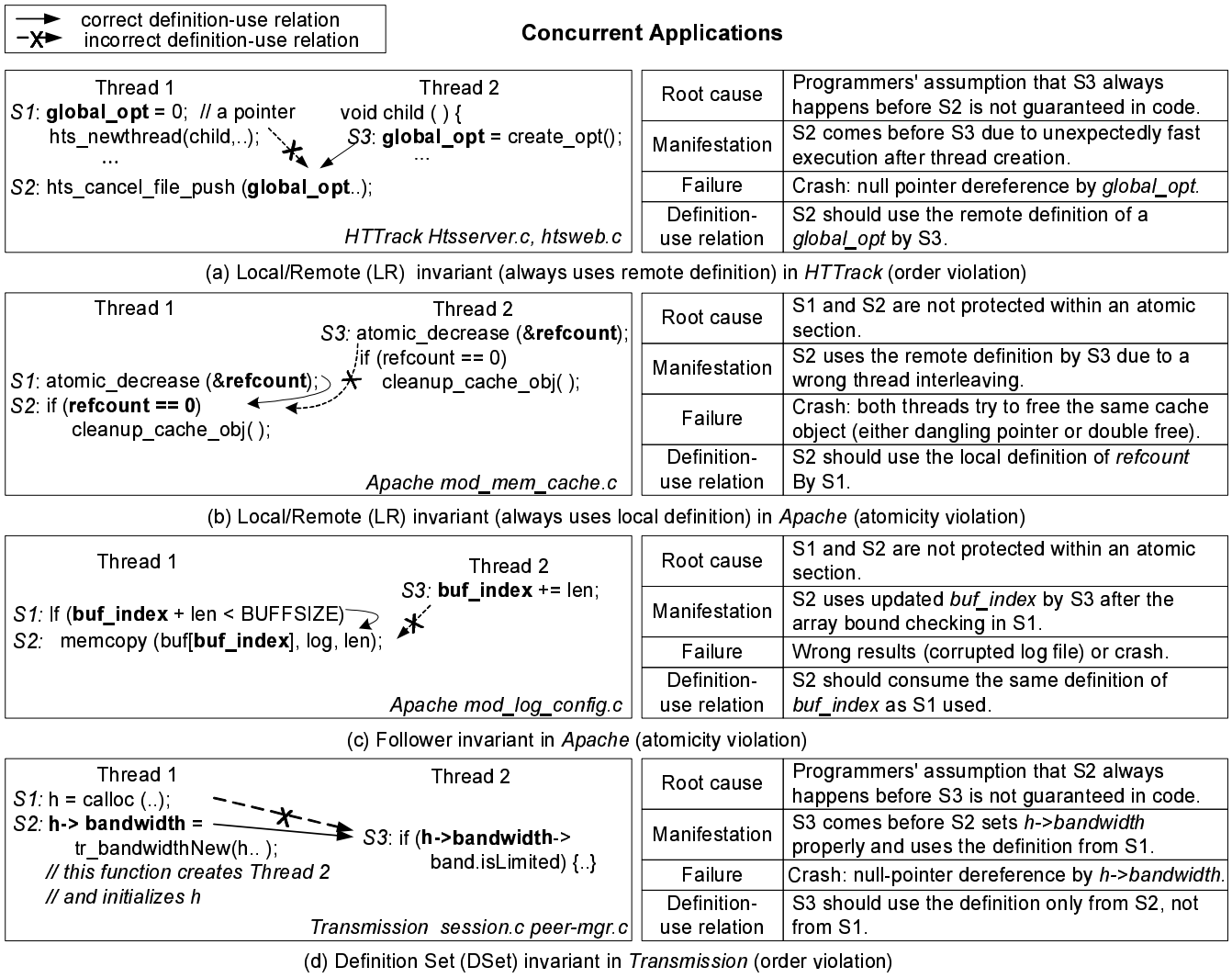


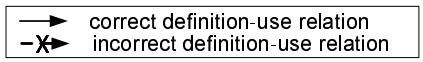
Figure 1. Real-world concurrency bug examples, definition-use (DefUse) invariants and violations.

uses for concurrent and sequential programs. A definition-use invariant reflects the programmers' assumptions about which definition(s) a read should use. Specifically, we have observed the following three types of definition-use invariants. We combine all the three types into **DefUse Invariants**.

- **Local/Remote(LR) invariants.** A read only uses definitions from either the local thread (e.g., Figure 1(b)); or a remote thread (e.g., Figure 1(a)). LR invariants are useful in detecting order violations and read-after-write atomicity violations in multi-threaded programs.
- **Follower invariants.** When there are two consecutive reads upon the same variable from the same thread, the second always uses the same definition as the first. For example, in Figure 1(c), $S2$ should use the same definition of `buf_index` as $S1$. Follower invariants are useful for detecting read-after-read atomicity violations and certain memory bugs.
- **Definition Set(DSet) invariants.** A read should always use definition(s) from a certain set of writes. For example, in Figure 2(a), $S1$ is the only correct definition for $S2$ to use. It can be useful for detecting both concurrency and sequential bugs but this would introduce more false positives.

We have designed a tool, called DefUse that automatically extracts definition-use invariants from programs written in C/C++ and uses them to detect software bugs of various types. *To the best of our knowledge, this is one of the first techniques that can be used to detect a large variety of software bugs, including concurrency bugs (both order violations and atomicity violations) as well as sequential bugs.*

Similar as many previous invariant-based bug detection studies [8, 13, 19, 37, 44, 45], DefUse leverages in-house testing as training runs to extract the definition-use invariants, which are then used to detect concurrency and sequen-



Sequential Applications

```
S1: int ifd = 0; /* file descriptor, 0 for stdin */
while (...) {
   iname = argv[optind++];
   if (strcmp(iname, "-") == 0) // "-?for stdin
S2:   get_method(ifd); // process stdin
   else { // process a normal file
S3:    ifd = open_input_file (iname, ...);
       get_method(ifd);
   }
}                                              gzip gzip.c
```

Root cause	Missing a reassignment to *ifd* before S2.
Manifestation	The program processes a normal file before stdin. S3 sets *ifd* first and S2 in the next iteration will incorrectly use this definiton.
Failure	Use the wrong file descriptor for stdin processing. The program fails to read from stdin.
Definition-use relation	Variable *ifd* in S2 should use only the definition from S1.

(a) Definition Set (DSet) invariant in *gzip* (semantic bug)

```
   S1: memcpy(fmt, ..., length);
   S2: fmt[length] = '\0';
   /* check the character after "%" in fmt. Any one in "efgaEFGA" is
      valid. But the programmer did not consider that '\0' is invalid. */
   S3: if (! strchr ("efgaEFGA", fmt[i]))
       return NULL;
   ... // convert fmt to ldfmt here
   S4: asprintf(..., ldfmt, ...)
                                                seq  seq.c
```

Root cause	Programmer did not consider the string ending character '\0' as valid.
Manifestation	When i is equal to length, *fmt[i]* at S3 is '\0'. But *strchr()* still returns non-NULL value because '\0' is the terminator of "efgaEFGA". Then *asprintf()* gets the wrong format string.
Failure	Wrong format for *asprintf()*.
Definition-use relation	*fmt[i]* at S3 should use the definition from S1. S2 is the wrong definition for *fmt [i]* at S3.

(b) Definition Set (DSet) invariant in GNU coreutil *seq* (semantic bug)

```
int char_to_clump (char c) {
   char *s = clump_buff;
   if (c == '\t') {          // width should be less than size of clump_buff
       width = chars_per_c - input_position % chars_per_c;
S1:    for (i = 0; i<width; i++)    *s++ = ' ';
   }
   else if (c == '\b') width = -1;
   input_position += width;        // input_position should be positive
}                                                pr  pr.c
```

Root cause	The value of width should be less than the length of *clump_buff*.
Manifestation	'\b' keys are entered until *input_position* becomes negative. It makes width much larger than the size of *clump_buff*.
Failure	Crash: corrupt a function pointer adjacent to *clump_buff*.
Definition-use relation	S1 should define data only for the uses of *clump_buff* (Definition Set (DS) invariant).

(c) Definition Set (DSet) invariant in GNU coreutil *pr* (buffer overflow)

```
/* If field_1_buffer is NULL, getndelim2 will allocate memory for the
pointer. Otherwise, it accesses the memory area leaded by
field_1_buffer directly. */
while (1) {
   len = getndelim2(&field_1_buffer, ...); // return -1 if failure
   If (len < 0)
      free(field_1_buffer); // field_1_buffer is dangling pointer
}                                                cut  cut.c
```

Root cause	After free, *field_1_buffer* is not cleared and is used again in *getndelim()*.
Manifestation	After first failure of *getndelim2*, *field_1_buffer* is freed. In next iteration, *getndelim2* will access invalid memory and fail again. Finally, *field_1_buffer* is double freed.
Failure	Invalid memory access and double free.
Definition-use relation	Memory accesses in *getndelim2* and the second time of *free* are out of the lifetime.

(d) Definition Set (DSet) invariant in GNU coreutil *cut* (dangling pointer)

Figure 2. Real-world sequential bug examples, definition-use (DefUse) invariants and violations.

tial bugs. To tolerate insufficient training, DefUse automatically prunes out unlikely invariants and violations, and ranks remaining violations based on confidence. Our invariant extraction algorithm also takes possible training noises (which may be incorrectly labeled training runs) into consideration.

We envision our tool being used in two scenarios: (1) *General bug detection during testing*: Once the invariants are extracted, DefUse can be used to monitor testing runs for possible violations. (2) *Post-mortem diagnosis*: When a programmer tries to diagnose a bug, he can use DefUse to see if any DefUse invariant is violated, and which "incorrect" definition-use data flow is detected. Such information provides useful clues for narrowing down the root cause. For such usage, we assume a deterministic replay support like the Flight Data Recorder [41] or others [27] to reproduce the failure occurrence.

We have evaluated DefUse with **twenty** representative *real-world* concurrency and sequential bugs [4], including order violation, atomicity violation, memory corruption, and semantic bugs, from **sixteen** server and desktop applications such as Apache, MySQL, Mozilla, HTTrack, GNU Linux coreutils, and so on. DefUse successfully detects 19 of them

[4] Like other dynamic tools such as AVIO [19] or Purify [15], our bug detection tool requires the bug to be manifested during execution (either a monitored run for post-mortem diagnosis or a production run). Hence, evaluation for a dynamic tool is usually done with mostly known bugs—whether it can detect these bugs or not. This does not mean that it can only detect known bugs. When such a tool (e.g., the commonly-used Purify [15] or our tool) is used in field, it can also help programmers to detect unknown bugs. Furthermore, during our evaluation process, we discovered two *new* bugs that were never reported before.

including 2 new bugs that were never reported before. Furthermore, DefUse reports only a few (0-3) false positives for our evaluated applications, benefiting from DefUse's pruning algorithm that can reduce the number of false positives by up to 87%. Our training sensitivity results show that DefUse is reasonably tolerant of insufficient training and achieves acceptable false positive rates after only a few (around 20) training runs.

2. Definition-Use Invariants

2.1 Overview

Classification of Program Invariants. Invariants are program properties that are preserved during correct executions. They reflect programmer intentions. Invariants are traditionally used for compiler optimization (e.g., loop invariants). Recent works automatically infer *likely* invariants of certain types from training runs and use them to detect software bugs. Previously proposed invariants include value-range invariants [8, 13], access-set invariants [45], access interleaving invariants [19] and others [6, 17].

Since few previous studies have classified program invariants, we summarize them into three categories as below:

- *Characteristic-based Invariants*: Sometimes, a program element's state may change, but all states share a common characteristic. Examples include "variable `x`'s value should be less than 5", "function `f`'s return value should be positive", "basic block `b` should be atomic", and so on.
- *Relation-based Invariants*: Some invariants are about the relationship among multiple program elements. An example is the multi-variable invariant discussed in MUVI [17]. It captures the relations between multiple variables that are always accessed together.
- *Value-based Invariants*: In some cases, certain program elements always have a fixed set of values. In this case, we use value in a broad sense. The value can be a variable value, or an access set, and so on. We call such invariants as value-based. The access set invariants discovered in AccMon [45] is such an example. It captures the set of instructions (program counters) that access a target memory location.

Definition-Use Invariants. As discussed in Section 1.2, *definition-use invariants* widely exist in both sequential and concurrent programs, but have not been studied before. They reflect programmers' intention on data flows and are closely related to many software bugs, such as those shown in Figure 1 and Figure 2.

From many real-world programs, we have observed that common data flow intention can be captured by the following three types of definition-use invariants, with each belonging to a category discussed above.

- *Local/Remote (LR)* is a type of characteristic-based invariants for concurrent programs. An LR invariant reflects programmers' intention on an important property of a read instruction: *should it use only definitions from a local (respectively remote) thread?* It is useful for detecting order violation bugs, and read-after-write atomicity violation bugs in concurrent programs.
- *Follower* is a type of relation-based invariant that is especially useful for concurrent programs. It reflects the relation between two consecutive reads to the same memory location from the same thread: *should the second read use the same definition as the first one?* It captures the atomicity assumption between two consecutive reads, and can be used to detect read-after-read atomicity violation bugs.
- *Definition Set (DSet)* is a value-based invariant and captures the set of definitions (i.e., write instructions) that a read instruction should use. Unlike LR and Follower, DSet is applicable to both concurrent and sequential programs. It is useful for detecting concurrency bugs (atomicity violations, especially read-after-write atomicity violations, and order violations), as well as memory bugs (dangling pointers, buffer overflow, and so on) and certain semantic bugs which are introduced by incorrect data flow.

The above types of definition-use invariants capture different data flow intentions and complement each other well. When used for detecting bugs, each has different trade-offs between detection power and the false positive rate. Therefore, we combine these three types into a single definition-use framework, called *DefUse*.

In the rest of this section, we first discuss each one with real-world examples, followed by the rationale for combining all three of them in DefUse.

2.2 Local/Remote (LR) Invariants

In concurrent programs, certain reads may use only local definitions (maybe due to atomicity) or only remote definitions (maybe for the purpose of communication or synchronization). LR invariants can capture these properties.

Formally speaking, an LR invariant, $LR(r)$, at a read, r, equals to "`LOCAL`" (respectively "`REMOTE`") if r uses only definitions from the local (respectively remote) thread. We denote them as `LR-Local` and `LR-Remote`, respectively. If r can read from either a local or a remote definition, r has no LR invariant. Figure 3(a) illustrates the basic idea of LR invariants.

Figure 1(a) and (b) demonstrate two different real-world examples of LR invariants from HTTrack and Apache and their corresponding violations. As we explained before in the introduction, in the example shown in Figure 1(a), the `global_opt` pointer at $S2$ in thread 1 should always use a definition from a remote thread, i.e., a child thread of thread 1, not from the local thread. Otherwise, it can lead to a null-pointer dereference. In other words, it has an `LR-Remote`

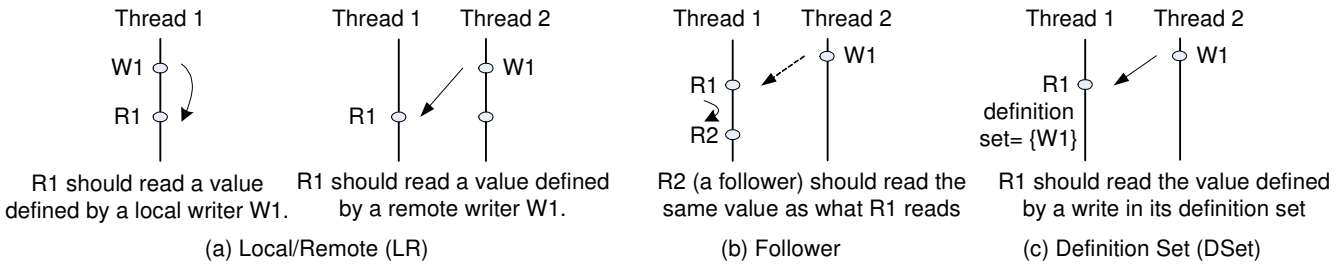


Figure 3. Examples of real-world definition-use invariants and their violations.

invariant but unfortunately it is not guaranteed by the implementation (probably due to the programmer's wrong assumption about the execution order after thread creation).

Conversely, in the Apache example shown on Figure 1(b), $S2$ always needs to read the `refcount` value decremented by the local thread, i.e., $S2$ holds a `LR-Local` invariant. Furthermore, due to the incorrect implementation, the above assumption is not guaranteed. Therefore, $S2$ and $S1$ can be interleaved by $S3$, which causes $S2$ to read the definition from a remote write $S3$. This violates $S2$'s `LR-Local` invariant.

2.3 Follower Invariants

In concurrent programs, programmers frequently make assumptions about the atomicity of certain code regions. The LR invariants already captures the case of read-after-write data flow relation in an assumed atomic region, but not the read-after-read case, which can be captured by using a Follower invariant.

Specifically, for two consecutive reads, r_1 and r_2, to the same location from the same thread, if r_2 always uses the same definition as r_1, we say r_2 has a Follower invariant. Follower is different from LR because as long as r_2 uses the same definition as r_1, the definition can come from either local or remote. Figure 3(b) demonstrates Follower invariants.

Figure 1(c) shows a real-world example of Follower invariants from Apache. As long as $S2$ reads the same definition as $S1$, wherever `buf_index` is updated, the execution is correct, i.e., $S2$ holds a Follower invariant. However, when such an assumption is not guaranteed, the sequence of $S1$ and $S2$ may be interrupted by the interleaved remote access $S3$, making $S2$ read a different value from $S1$. Therefore, $S2$'s Follower invariant is violated. This is an atomicity violation bug.

2.4 Definition Set (DSet) Invariants

While concurrent programs have special inter-thread data flows, definition-use is not specific to only concurrent programs. Our third type of invariants, *Definition Set (DSet)*, is suitable for both sequential and concurrent programs. A DSet invariant at a read is defined as the set of all writes whose definitions this read may use. Figure 3(c) shows a DSet invariant at R1. Every read instruction has such a DSet. When it consumes a value defined by an instruction outside its DSet, a DSet invariant is violated, indicating a likely bug.

Figure 1(d) shows a real-world example of a DSet invariant in a concurrent program, Transmission. $S3$ is supposed to use `h->bandwidth`'s value defined only by $S2$. If the thread creation and initialization in $S2$ takes a longer time than the programmer's assumption, $S3$ may use the definition from $S1$, violating the DSet invariant of $S1$. It is of note that there is no Follower invariant here. Even though there is an LR-Remote invariant, the bug's manifestation condition does not violate LR-Remote at $S3$ because the wrong definition $S1$ also comes from a remote thread (Thread 1). Therefore, LR or Follower invariants are not effective in detecting this bug, while DSet is.

Apart from concurrent programs, DSet is also applicable to sequential programs, as shown in Figure 2. The DSet invariant at $S2$ in Figure 2 (a) captures the set of valid definitions for S2 to use: $\{S1\}$. When the semantic bug is triggered, $S2$ would use a definition from $S3$, violating $S2$'s DSet invariant. Hence, this bug can be detected. The other three sequential bug examples are similar.

2.5 DefUse: Combining All Three Invariants

The above three types of invariants capture different aspects of definition-use properties. Although DSet invariants appear more general than LR and Follower, they do not necessarily subsume LR and Follower since they do not capture inter-thread information. In a definition set, we do not express whether this definition is local or remote. As a result, in some concurrency bug cases, DSet may not be violated, but LR or Follower invariants are, as shown in the examples below.

Figure 1(b) shows such an example from Apache Httpd. The variable `refcount` definition used at $S2$ is always from `atomic_decrease()` at $S1$. However, when the bug is triggered, $S2$ can use a definition from a remote thread also executing `atomic_decrease()`. Therefore, the DSet invariant at $S2$ is not violated during such abnormal execution. However, the LR invariant is violated.

Similarly, DSet invariants do not completely cover Follower invariants either. For example, in Figure 1(c), there are two definition-use chains, $S3$-$S1$ and $S3$-$S2$, which can be recorded in two DSets. However, the dependence relation

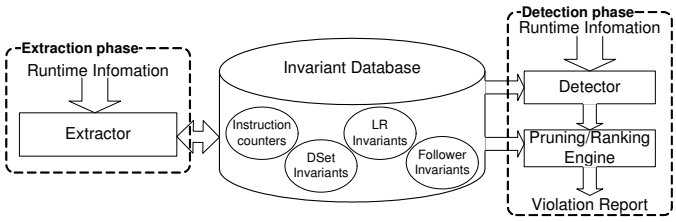


Figure 4. Overview of DefUse system.

between the two reads' definitions are not captured by DSet. In an incorrect run, if another instance of $S3$ is interleaved between $S1$ and $S2$, no DSet invariant is violated, although the Follower invariant at $S2$ is.

In addition, since LR and Follower invariants capture general properties instead of concrete values, they provide fewer false positives in bug detection. It follows that their generality may also lead to failure to detect the actual bugs. In particular, LR and Follower invariants are not suitable for capturing sequential data flow, whereas DSet is. Therefore, DSet can be used for detecting memory and semantic bugs.

Due to the above reasons, our work combines all three invariants into a single framework, called *DefUse*.

3. Invariant Extraction

DefUse has two phases: (1) an extraction phase for inferring definition-use invariants; and (2) a detection phase for detecting invariant violations and reporting potential bugs after pruning and ranking. DefUse uses four components for these two phases: Extractor, Detector, Invariant Database, and Pruning/Ranking Engine, which are shown in Figure 4.

This section discusses how to automatically infer programs' definition-use invariants. It first describes the general idea and then provides the extraction algorithms for LR, Follower and DSet invariants. The next section will discuss violation detection, pruning and ranking.

3.1 The Main Idea

Definition-use invariant are hard to extract using static code analysis, especially from concurrent programs. It is also too tedious to require programmers to write down all invariants. In addition, programmers may not be consciously aware of many such invariants. Otherwise, they probably would have implemented the code correctly without introducing such bugs.

Similar to the previous works on extracting program invariants, such as Daikon [8], DIDUCE [13], AVIO [19], AccMon [45], and so on, we also rely on correct runs from in-house testing to statistically infer definition-use invariants. Although it is possible to combine this with static analysis, it is difficult to use such information especially for concurrent programs written in C/C++, for which it is hard to statically determine which code can be executed in parallel. Similar to the previous works, we also assume that the target program is reasonably mature and thereby feasible for extracting invariants statistically.

Symbol	Description
I	a static instruction
I_U	a read instruction (i.e., a use)
I_D	a write instruction (i.e., a definition)
i	the latest dynamic instance of I
i_u	the latest dynamic instance of I_U
i_d	the latest dynamic instance of I_D
$T(i_d)$ or $T(i_u)$	i_d's or i_u's thread identifier

Table 1. Symbols used in this paper. All instruction symbols are for the memory address m by default.

As pointed out by the previous works [19, 44], concurrent programs provide a unique challenge for generating different training runs. Different runs, even with the same input, can have different interleavings. A systematic concurrency testing framework such as CHESS [25, 26] or CTrigger [30] can be used to systematically explore different interleavings for each input.

Insufficient training problem and training sensitivity. It is conceivable that training runs may not cover all execution paths. Therefore, the extracted invariants may suffer from the problem of insufficient training. To address this, both our invariant extraction and bug detection algorithms explore various techniques to filter unlikely invariants or violations, and devalue unconfident invariants and bug reports (see Section 4.2). Moreover, the extracted invariants can be refined after each detection run based on the programmer's feedback on bug reports. Finally, in our experiments, we studied the training sensitivity by testing with the inputs that are very different from that used in the training (see Section 5.5).

3.2 Extraction Algorithms

In this section, we describe in detail the algorithms to extract definition-use invariants. Due to page limitations, we present only the algorithms for concurrent programs. The extraction for DSet invariants works for sequential programs as well. For simplicity, we use the symbols explained in Table 1.

DSet invariant extraction. Our DefUse extraction algorithm obtains DSet by collecting all definitions that are used by I_U during training runs. In our implementation, DefUse uses the Pin [21] binary instrumentation framework to instrument accesses to every monitored memory location. For each memory location, DefUse stores its most recent write instruction in a global hash-table, called *Definition-Table*. At a read i_u, DefUse gets its definition i_d from the *Definition-Table* and uses i_d's corresponding static instruction I_D to update I_U's DSet:

$$DSet(I_U) \leftarrow DSet(I_U) \cup \{I_D\} \quad (1)$$

After the training phase, information for every instructions' DSet is stored into the invariant database along with

some statistical information such as the number of times I_U and I_D are executed, and so on, for the purpose of pruning and ranking.

LR invariant extraction. In order to infer LR invariants, DefUse firstly needs to know which thread provides the definition for a read. This can be easily obtained through the *Definition-Table*. Specifically, when i_u is executed, DefUse finds its definition i_d from the *Definition-Table*. DefUse then compares $T(i_u)$ and $T(i_d)$ to determine whether they are from the same thread or not. Finally, it compares the answer with the $LR(I_U)$ associated with I_U. If they are different, $LR(I_U)$ is set to `NO_INV` to indicate that there is no LR invariant at I_U and this read is no longer monitored for LR invariant extraction. $LR(I_U)$ is initialized based on the definition source (either `REMOTE` or `LOCAL`) on the first time I_U is executed. This process can be formalized as follows:

$$LR(I_U) \leftarrow \begin{cases} LOCAL & \text{if } LR(I_U) = LOCAL \\ & \wedge\, T(i_d) = T(i_u) \\ REMOTE & \text{if } LR(I_U) = REMOTE \\ & \wedge\, T(i_d) <> T(i_u) \\ NO\_INV & \text{Otherwise} \end{cases} \tag{2}$$

Follower invariant extraction. In order to infer Follower invariants, DefUse needs to store its recent access history to determine whether an instruction and its predecessor use the same definition. To achieve this, DefUse maintains a bit-vector for every memory location m, called `has_read(m)`. A bit in the vector `has_read(m,t)` indicates whether the current definition to memory location m has already been used by thread t. By checking this bit-vector before every read instruction i_u, DefUse can easily determine whether i_u and its predecessor use the same definition.

Specifically, bit-vector `has_read(m)` is initialized as zero. Every write to m sets all bits of `has_read(m)` to zero. After any read from m in thread t, `has_read(m, t)` is set to one. In order to know whether an instruction i_u uses the same definition as its predecessor, before executing i_u, DefUse checks if the corresponding bit (i.e., `has_read(m, t)`) is one. If it is, it means that there is no new definition since thread $T(i_u)$'s last use to m. In other words, i_u shares the same definition with its predecessor.

To maintain and update the Follower invariant information for I_U, DefUse associates $Follower(I_U)$ with it. This flag is set to `TRUE` if I_U's dynamic instances always share definition with their predecessors. Whenever a dynamic instance of I_U uses a different definition from its predecessor, the flag is set to `FALSE` and I_U is no longer monitored for Follower invariant extraction.

$$Follower(I_U) \leftarrow Follower(I_U) \wedge has\_read(m, T(i_u)) \tag{3}$$

3.3 Design and Implementation Issues

Our implementation is based on Pin [21], a dynamic binary instrumentation framework. In the following paragraphs, we discuss several detailed implementation issues.

Monitored memory locations. The invariant extraction algorithms can be applied to any memory unit (byte, word, and so on) and any memory region (stack, heap, global data region, and so on). We use byte granularity in our implementation for the greatest accuracy. Currently, our DefUse prototype monitors heap and global data region. Stack is ignored because variables stored on stack seldom involve in concurrency bugs [18]. Nevertheless, some sequential bugs may not be detected because of this choice. We will extend DefUse in the future to cover these stack related sequential bugs.

External definitions. Programs may use external libraries and system calls. We term the definitions from those external codes as *external definitions*. The absence of these external definitions could result in some definition-use invariants not being detected.

We use two approaches to solve this problem. Firstly, we annotate some system calls and widely used library functions, such as `memset` and `memcpy` which can create definitions. We regard these functions as black boxes and their call-sites as definitions.

Of course, it is impractical to annotate all external functions and system calls. Therefore, our second approach is to introduce $\xi = (I_\xi, T_\xi)$ to represent unannotated external definitions. When DefUse meets a use and does not find any preceding definition to the same memory location, the symbol I_ξ is introduced to represent the virtual external definition.

It is of note that since we instrument the code at binary level, we can always detect every user-level definition and use, regardless of whether or not it is from libraries. However, annotation may improve context-sensitivity compared to directly processing the definitions within them.

Virtual address recycle. We also need to consider virtual address recycling caused by memory allocation and de-allocation. In other words, some instructions may access the same memory location simply due to memory recycling rather than for some inherent data flow relation. To address this problem, we intercept every de-allocation function, and delete all the previous information for accesses to the future-deallocated memory region.

Context-sensitivity. Small un-inlined functions may have many call-sites. As a result, a use in such a function may have many definitions from different call-sites, which can lead to inaccurate invariants. This issue can be addressed in two ways. The first is to inline this small function, and treat each call-site's use as different reads. In other words, we can make it context-sensitive. A simpler approach is to just prune

such use from the invariant database since it has too many definitions.

Training noise. Although existing in-house testing oracles are relatively accurate in labeling incorrect runs from correct ones in most cases, we also suffer from the problem of training noise (i.e., a buggy run is incorrectly labeled as correct), as experienced with all previous works in invariant-based bug detection [8, 13, 19, 45]. To handle this problem, our extraction algorithm can be extended by relaxing the 100% support constraint. For example, if an invariant is supported by 95% of the training runs, this invariant is not pruned and is still kept to detect potential violations during monitored runs. Intuitively, violations to such invariants should be ranked lower than those for invariants with 100% support from training runs.

4. Bug detection

4.1 Detection Algorithms

DefUse detects bugs by checking violations against the extracted DSet, LR, and Follower invariants. At every read instruction, if the monitored properties do not match the associated invariants, DefUse reports such violation to its Rank/Pruning Engine for further analysis.

The detection algorithms are similar to the extraction algorithms discussed in Section 3. In the below section, we briefly describe the basic detection algorithms against each type of definition-use invariant violations. We will discuss how to prune and rank bug reports in Section 4.2.

To detect DSet invariant violation, DefUse maintains a *Definition-Table* at runtime so that it knows which instruction provides a definition for each use. If the definition for a use is not in this use's DSet, a violation is issued. Formally speaking, the violation condition is:

$$I_D \notin DSet(I_U) \tag{4}$$

Detecting violations against LR invariants is also straightforward. At a monitored read, DefUse first checks whether this read has an LR invariant or not. If it does, DefUse examines whether the monitored read and its definition come from the same thread and matches the monitored condition with the type of LR invariant (`LOCAL` and `REMOTE`) extracted at this read. If there is a violation, it will be reported to the Pruning/Ranking Engine:

$$\begin{array}{l}\{LR(I_U) = LOCAL \ \wedge \ T(i_d) <> T(i_u)\} \vee \\ \{LR(I_U) = REMOTE \ \wedge \ T(i_d) = T(i_u)\}\end{array} \tag{5}$$

The basic idea of detecting violations to Follower invariants is to check whether an instruction with a Follower invariant shares the same definition with its predecessor (by leveraging the `has_read(m)` vector similar to that used in extraction). If not, a violation will be reported for pruning and ranking:

$$Follower(I_U) \ \wedge \ \overline{has\_read(m, T(i_u))} \tag{6}$$

4.2 Pruning and Ranking

For training-based approaches, it is usually hard to know whether the training is sufficient. Without ranking and pruning, bug detection using such approaches may generate a significant number of false positives due to insufficient training.

Pruning. DefUse automatically prunes the following cases in bug reports:

- *Barely exercised uses*: For reads that are never covered during training, we do not report any violations since we do not extract any invariants associated with them. It is also useful to extend this strategy to those uses that we have only seen once or twice during training runs and therefore have little confidence of in the extracted invariants.
- *Barely exercised definitions*: Similarly, for definitions that are never exercised during training runs, if we encounter them during detection, we are not confident whether they are violations. In this case, we also prune them from the bug reports.
- *Popular uses*: Some uses, such as those in a small function called from multiple call-sites, are very popular and have a large definition set. In this case, we also prune it as it might be perfectly acceptable to have yet another definition for this use during detection runs.

Ranking. After pruning the above cases, DefUse ranks every unpruned violations based on its confidence. In this way, users can start with the top-ranked bug reports and gradually move down the list. In addition, DefUse also provides an optional pruning phase, called *confidence-based pruning*, which prunes the violations whose confidence is lower than a certain threshold. By default, we do not use such an option for our experiments unless explicitly mentioned.

Intuitively, the following conditions increase the confidence of a true bug:

- many dynamic instances of the definition ($\#I_D$) and the use ($\#I_U$) during training;
- no significant difference between the number of instances of the definition and instances of the use ($|\#I_D - \#I_U|$) during training;
- small definition set ($|DSet(I_U)|$);
- few instances of this violation pair ($\#violation_{DSet}(I_D, I_U)$) during detection.

For a DSet invariant violation, if the invariant is supported by many instances of definitions and uses during training, any violation to it has a high probability of being a bug. Meanwhile, if the size of $DSet(I_U)$ is large, or the same violation is detected many times, the violation is unlikely to be a bug. As such, the confidence is computed as follows.

$$conf_{DSet} = \frac{\#I_D \times \#I_U}{(|\#I_D - \#I_U| + 1) \times |DSet(I_U)| \times \#violation_{DSet}(I_D, I_U)} \quad (7)$$

For LR and Follower violations, the confidence is computed based only on the number of dynamic instances of a use during training and the number of violations occurring during detection, as follows:

$$conf_{LR} = \#I_U / \#violation_{LR}(I_U) \quad (8)$$

$$conf_F = \#I_U / \#violation_F(I_U) \quad (9)$$

Violations to multiple invariants. In some cases, a bug may violate multiple invariants (e.g., both the DSet and LR invariants). In these case, DefUse uses a geometric mean of the confidence of all violations as DefUse's confidence. As an alternative approach, we can also weigh each invariant differently. For example, violations to LR and Follower invariants may have higher confidence values, since they are coarse-grained invariants and hence can gain more statistical support from training runs.

5. Experimental Evaluation

5.1 Test Platform, Applications and Bugs

We implemented DefUse using Intel's dynamic instrumentation tool, Pin [21], for invariant extraction and bug detection. All experiments are conducted on an 8-core Intel Xeon machine (2.33GHz, 4GB of memory) running Linux 2.6.9.

We used a total of **sixteen representative real-world applications** (shown in Table 2), including 8 multi-threaded applications and 8 sequential applications. Concurrent applications include 2 widely used servers (Apache Httpd and MySQL), 6 desktop/client applications (Mozilla JavaScript engine, HTTrack, Transmission, PBZip2, and so on). We used the original applications from their official websites and did not modify any of them for our evaluation.

We evaluated DefUse with **20 real-world software bugs** (including 11 concurrency bugs and 9 sequential bugs) of various root causes, including atomicity violations, order violations, memory bugs and semantic bugs.

5.2 Training Process

In our experiments, we tried our best to emulate how programmers would use DefUse in practice, especially for invariant extraction from training runs. The following is our training setup. Like all previous invariant-based works, we leverage the common available testing oracle from in-house regression testing to label correct testing runs for training.

Invariants are extracted systematically. Specifically, we used multiple different inputs for training, which are crucial for sequential applications. All inputs used for training are from standard test cases released together with the corresponding software. In other words, training inputs are selected in a way that is oblivious to the tested bug. In some experiments (for studying training sensitivity), we even manually examined the training inputs to ensure that they are different from the input used for detection (as explained in the next subsection).

For concurrent applications, in each input, training also needs to cover various interleavings [25, 26, 36]. Therefore, we executed the tested program with each input for 10 times [5] on our 8-core machine. It is conceivable to use more advanced interleaving testing tools [25, 26, 30, 36] to improve DefUse training process to cover more interleavings.

Type		Applications	LOC	Descriptions
Concurrent App.	Server App.	Apache	345K	Web server
		MySQL	1.9M	Database server
	Desktop App.	Mozilla	3.4M	Web browser suite
		HTTrack	54.8K	Website copier
		Transmission	86.4K	BitTorrent client
		PBZip2	2.0K	Parallel file compressor
		x264	29.7K	H.264 codec
		ZSNES	37.3K	Nintendo game emulator
Sequential App.	Utility	gzip	14.6K	Compression tool
		tar	41.2K	File archiving tool
	GNU Linux coreutils	seq	1.7K	Print number sequence
		paste	1.4K	Merge lines of files
		sort	4.3K	Sort lines in files
		ptx	5.7K	Index of file contents
		cut	3.3K	Remove sections from files
		pr	6.3K	Convert files for printing

Table 2. Evaluated applications.

Bug Type	ID	Bug description
Atomicity Violation Bugs	Apache#1	Random crash in cache management
	Apache#2	Non-deterministic Log-file corruption
	Apache#3	Random crash in worker-queue accesses
	Mozilla	Wrong results of JavaScript execution
	MySQL#1	Non-deterministic DB log disorder
Order Violation Bugs	MySQL#2	Read after free object
	HTTrack	Read before object creation
	Transmi -ssion	A shared variable is read before it is properly assigned by a child thread
	PBZip2	Random crash in file decompression
	x264	Read after file closed
	ZSNES	Inquiry lock before initialization
Semantics	gzip	Use wrong file descriptor
	seq	Read wrong string terminator
Unbounded memory read	tar#1	Read out of buffer
	paste	Read out of buffer
	sort	Read unopened file
Dangling Pointer	cut	Dangling pointer
Memory corruption	ptx	Buffer overflow
	pr	Buffer overflow
	tar#2	Overwrite wrong buffer fields

Table 3. Evaluated real-world bugs.

[5] In the case of Apache server, we define one run as the duration to service for 100 different client requests, and for HTTrack, web crawling time for one run is set to five minutes.

5.3 Two Sets of Experiments

DefUse is designed for two different usage scenarios (post-mortem diagnosis and general bug detection during testing) as mentioned in the introduction. During post-mortem diagnosis, developers usually know which input triggers the bug so that they can easily design a good training input set for DefUse. For general bug detection, programmers can rely only on in-house testing suites, and so it is possible that some monitored runs may have a very different input. In such cases, it would be desirable for a bug detector to not report many false positives.

Considering the above two different usage scenarios, we design two sets of experiments to thoroughly evaluate DefUse.

1. *DefUse bug detection with sufficient training inputs.* In this set of experiments, DefUse's training input set includes all suitable inputs provided by the software developers or those generated by representative tools(e.g., for MySQL, we used the *mysql_test* suite; for Apache, we used Surge [2] and httperf [24]). For some applications (e.g., tar, sort), we generated many different inputs. The results with this setting are shown in Table 4 and Table 5.
2. *DefUse bug detection with insufficient training inputs.* We use two experiments to show how DefUse would perform with insufficient trainings: how the number of false positives changes with inputs different from those used in training (Figure 5); and the amount of training runs/inputs needed to achieve a good detection accuracy (Figure 6).

In all our experiments, confidence is only used for ranking, but not for pruning, unless specifically mentioned (Figure 5(b) and Figure 6(b)).

5.4 Overall Results

Table 4 shows DefUse's detection capability and the number of false positives, as well as the bug rankings in DefUse's violation report.

Bug detection. As shown on Table 4, DefUse can detect 19 of the 20 tested bugs in the 16 applications. *It is of note that these bugs are of various types including atomicity violations, order violations, semantic bugs, buffer overflows, dangling pointers, unbound reads, and so on.* Section 5.6 explains why DefUse misses the bug of tar#2.

The results also show that the three types of definition-use invariants (LR, Follower and DSet) are very complementary. DSet can detect 7 of the 11 concurrency bugs and all the tested sequential bugs, but misses 4 concurrency bugs which can be detected by LR or Follower. For example, Apache#1 and Mozilla are detected only by using LR invariants, and Apache#2 is detected only by using Follower. This justifies the benefit of combining all three invariants in DefUse.

DefUse can also detect bugs that would be missed by previous tools. One is an order violation bug in HTTrack and the other two are semantic bugs in gzip and seq. Specifically, the recently proposed invariant-based bug detector, PSet [44], cannot detect the HTTrack bug, since their detection is not strictly based on definition-use relations. Figure 8 in Section 6 shows the simplified codes of HTTrack, and compares DefUse with PSet's behavior. It can be noted that DefUse can detect the bug with both DSet and LR invariants. A more detailed comparison with PSet with a bug example is in Section 6.

The evaluated two semantic bugs are caused only by a semantically wrong definition-use pair and not by accessing an invalid memory location. Hence, they would not be detected by previous memory checkers such as Valgrind [28].

False positives. In Table 4, for all the applications, the number of false positives is fewer than four. Moreover, almost all bugs ranked top in the violation report, which helps programmers easily identify bugs.

The low false positive numbers are a result of our pruning scheme described in Section 4.2 as well as sufficient training (the results of insufficient training are shown later). Table 5 demonstrates the effectiveness of our basic pruning algorithm (please note that no confidence-based pruning is used here). Overall, the number of false positives is reduced by up to 87% by pruning.

Comparing LR and Follower with DSet, the first two have much fewer false positives because they are coarse-grained (i.e., they look at only high-level definition-use properties instead of concrete definitions) and have more statistical supports from training runs.

5.5 Training Sensitivity

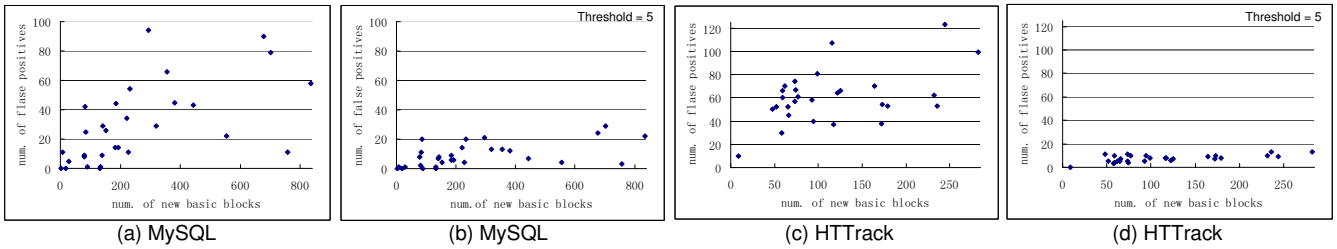
As discussed in Section 5.3, in some scenarios and especially if used for general bug detection instead of post-mortem analysis, it is possible that training is not sufficient. That is, the training does not have a good coverage of execution paths or inputs. Our second set of experiments aim to answer the following related questions:

What if detection runs are dramatically different from training runs? To show DefUse's training sensitivity, we first analyze the effect of training inputs as shown in Figure 5. For detection, we use various inputs that are different from training inputs. To quantitatively measure their differences, we use the number of basic blocks newly explored in a detection run (but not in any training run).

Without any confidence-based pruning, as shown on Figure 5(a)(c), when a detection run is very different from training runs (e.g., for MySQL, with 835 number of untrained basic blocks), DefUse can introduce a few (e.g., 58 in MySQL) false positives. Fortunately, this problem can be addressed by applying confidence-based pruning, i.e., pruning out those with low confidence. In Figure 5(b)(d), after confidence-based pruning, even with 700 untrained basic blocks in a detection run, the number of false positives is always below 30. However, confidence-based pruning may also increase

Bug Type	Applications	LR		Follower		DSet		DefUse	
		Bug Detected?	False Positives	Bug Detected?	False Positives	Bug Detected	False Positives	Bug Detected?	False Positives
Concurrent Applications									
Atomicity Violation Bugs	Apache#1	Yes(1)	0	no	0	no	1	Yes (2)	1
	Apache#2	no	2	Yes(1)	0	no	3	Yes(1)	3
	Apache#3	Yes(1)	0	Yes(1)	0	no	0	Yes(1)	0
	Mozilla	Yes (1)	0	no	0	no	3	Yes (1)	3
	MySQL#1	no	0	no	1	Yes(1)	0	Yes(2)	1
Order Violation Bugs	MySQL#2	no	0	no	1	Yes(1)	2	Yes(1)	3
	HTTrack *	Yes(1)	0	no	0	Yes(1)	0	Yes(1)	0
	Transmission *	no	0	no	0	Yes(1)	0	Yes(1)	0
	PBZip2	no	0	Yes(1)	0	Yes(1)	0	Yes(1)	0
	x264	no	0	no	0	Yes(1)	0	Yes(1)	0
	ZSNES	Yes(1)	0	no	0	Yes(1)	0	Yes(1)	0
Sequential Applications									
Semantics	gzip	-	-	-	-	Yes(1)	0	Yes(1)	0
	seq	-	-	-	-	Yes(1)	0	Yes(1)	0
Unbounded memory read	tar#1	-	-	-	-	Yes(1)	0	Yes(1)	0
	paste	-	-	-	-	Yes(1)	0	Yes(1)	0
	sort	-	-	-	-	Yes(2)	1	Yes(2)	1
Dangling Pointer	cut	-	-	-	-	Yes(1)	1	Yes(1)	1
Memory corruption	ptx	-	-	-	-	Yes(1)	0	Yes(1)	0
	pr	-	-	-	-	Yes(1)	0	Yes(1)	0
	tar#2	-	-	-	-	no	0	no	0

Table 4. Overall results from DefUse with invariants extracted from sufficient training runs described in Section 5.3. The number inside each parenthesis is the bug's ranking in the violation report of DefUse. The bugs marked * are new bugs that were never reported before.



(a) MySQL before confidence-based pruning | (b) MySQL after confidence-based pruning | (c) HTTrack before confidence-based pruning | (d) HTTrack after confidence-based pruning

Figure 5. Training sensitivity in terms of input difference. Each dot is one detection run. The detection runs use various inputs that differ from those used in training runs by various degrees. For each detection, its difference from the training runs is reflected by the number of basic blocks that are not covered by the training runs. The x-axis is the number of such basic blocks. The y-axis is the number of false positives occurred in each detection run.

the chance of pruning out some true bugs, even though this was not demonstrated in our experiments.

How many training runs/inputs are sufficient? Figure 6 shows how the number of training inputs/runs affects the number of false positives. Without confidence-based pruning (shown on Figure 6(a)), after DefUse training with 13 inputs, the number of false positives is gradually reduced, and finally reaches to 1. Confidence-based pruning (shown on Figure 6(b)) makes the false positive curves drop faster.

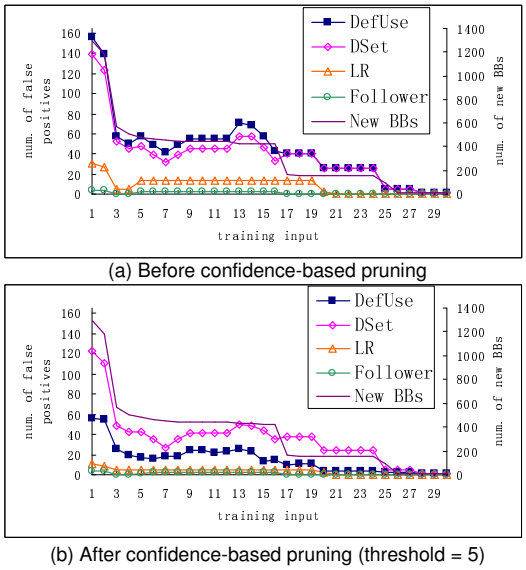
In comparison, LR and Follower's false positives are always fewer than DSet. DefUse has fewer false positives than DSet with confidence pruning, because its confidence is calculated using a geometric mean of the confidences from all three types of invariants.

5.6 Bug Missed by DefUse

Even though DefUse can detect various types of concurrency and sequential bugs, it still misses one tested bug (tar#2), which is described in Figure 7. If a file name (i.e., `read_s_name`) is longer than `posix_header.name`, $S1$ corrupts the following fields, e.g., `header.mode`, and dumps the wrong data to `fd`. Note that the wrong write $S1$ and its use $S3$ do not violate the DSet invariant held by $S3$, since this definition-use relation has already been captured from `header.name` during training. Interestingly, other existing memory checkers (i.e., Valgrind [28]) cannot detect it as well, since $S1$ always writes to valid location, even when this bug is manifested.

Application	LR		Follower		DSet		DefUse	
	BP	AP	BP	AP	BP	AP	BP	AP
Apache#1	0	0	0	0	2	1	2	1
Apache#2	2	2	0	0	3	3	3	3
Apache#3	0	0	0	0	0	0	0	0
Mozilla	0	0	1	0	4	3	5	3
MySQL#1	0	0	1	1	4	0	5	1
MySQL#2	0	0	1	1	22	2	23	3
HTTrack	0	0	0	0	1	0	1	0
Transmission	0	0	0	0	0	0	0	0
PBZip2	0	0	0	0	0	0	0	0
x264	0	0	0	0	1	0	1	0
ZSNES	0	0	0	0	0	0	0	0

Table 5. Comparison on the number of false positives before pruning (BP) and after pruning (AP). AP prunes barely exercised uses and definitions, and popular uses, as described in Section 4.2. The results show the worst case from five detection runs. The evaluation setup is the same as Table 4.



(a) Before confidence-based pruning

(b) After confidence-based pruning (threshold = 5)

Figure 6. False positives with increasing number of training runs. The plain-line curve shows the numbers of untrained basic blocks in the detection runs.

To address this problem, DefUse needs to be extended to consider other types of definition-use invariants such as *whether a definition is used at least once before it is re-defined*. In Figure 7, when the bug is manifested, `header.typeflag` is re-defined by $S2$ without any uses after $S1$, which can be detected by the extension.

Like previous invariant-based approaches, DefUse would miss bugs where there is no statistical support to extract any invariants, and bugs that do not violate any invariants.

5.7 Overhead of DefUse

In Table 6, the slowdown of DefUse is less than 5X in most cases and up to 20X for memory intensive concurrent applications such as application "x264". Compared to some other

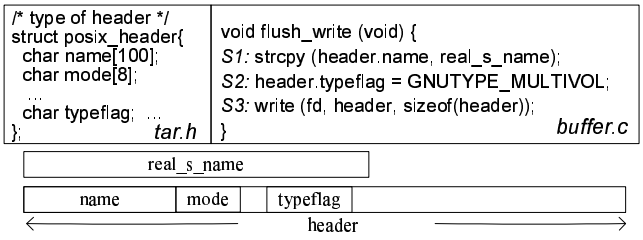


Figure 7. A memory corruption bug in *GNU tar* that cannot be detected by DefUse.

Mozilla	MySQL	Transmission	PBZip2	x264	gzip
3.26X	5.53X	1.03X	3.39X	20.26X	4.12X
seq	tar	paste	sort	cut	ptx
1.87X	2.18X	2.71X	2.45X	2.39X	2.41X

Table 6. Overhead of DefUse.

software only concurrency bug detection tools such as Valgrind (which includes a lockset data race detector and incurs 10X-100X overhead) [28], and the AVIO software implementation (which has 15X-40X slowdowns) [19], DefUse is much faster. Even though DefUse overhead may be high for some applications, it is not critically important because we expect Defuse to be used during post-mortem diagnosis and in-house testing/debugging, similar to Valgrind.

6. Related work

Program invariants. Our work was inspired by many previous works on program invariants including Daikon [8], DIDUCE [13], AccMon [45], AVIO [19], DITTO [37], MUVI [17], and so on. Similar to these studies, we also extract invariants from training runs, and dynamically detect violations to report potential bugs. Unlike these works, our invariants focus on data flow. More specifically, the focus is on definition-use invariants, which can be used to detect both concurrency and sequential bugs. Few previous methods can detect both types. Furthermore, our tool is also one of the first that can detect order violations in concurrent programs.

Definition-use and data flow. A recent study [4] used data flow to detect security vulnerabilities, i.e., buffer overflow attacks. It was based on the insight that if an intended data flow is subverted with a unexpected use, there may lead to an attack. Our work differs from this as we determine whether a use reads from a wrong definition by extracting definition-use invariants from both concurrent and sequential programs, and report potential concurrency (including atomicity and order violations) and sequential bugs.

Definition-use relations have also been used for software testing. Many works [14, 16, 43] have focused on testing definition-use relations to increase test coverage.

Data flow has many other usages. For example, EIO [12] uses data flow analysis to see whether low-level error codes (e.g., "I/O error") are correctly propagated to file systems so

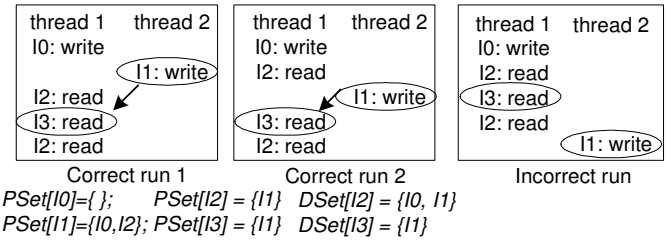


Figure 8. Simplified code of HTTrack showing the difference between PSet and DSet. In the incorrect run (the execution order of $I1$ and $I3$ does not match with the assumption), $I3$ obtains the definition from $I0$, violating $DSet(I3)$.

that recovery mechanisms can handle them. In [5], data flow analysis is used to detect memory leak and double free.

Concurrency bug detection and avoidance. There have been a lot of works on detecting data races [10, 23, 39]. The happens-before [31], lockset algorithms [7, 34, 38], and many of the extensions [29, 32] have been proposed for data races detection. However, there are two problems for data race detectors: (1) many data races in real-world applications are benign such that race detectors may have too many false alarms; and (2) some wrong interleavings and orders are not related to data races. Therefore, concurrency bugs caused by atomicity violations and order violations are introduced [18]. Atomicity violations have been well studied [9, 11, 19, 33, 42, 44], but order violations are, so far, rarely studied.

Much research has been conducted on detecting or avoiding concurrency bugs. Recently, [44] proposed an innovative method to avoid concurrency bugs by using data dependency information collected during correct runs. Their approach encodes the set of tested correct interleavings in a program's binary executable, and enforces only those interleavings at runtime. They use Predecessor Set(PSet) constraints to capture the tested interleavings between two dependent memory instructions. Specifically, for each shared memory instruction, PSet specifies the set of all valid *remote* memory instructions that it can be immediately dependent upon [6].

Their work is quite different from DefUse in the following aspects. (1) Their work focuses on runtime fault tolerance and recovery, while DefUse is tailored to bug detection and post-mortem failure diagnosis. (2) PSet does not directly target definition-use relations even though it reflects some aspect of them. Specifically PSet does not capture such definition-use relations as it cares about only *remote*, *preceding* instructions. Figure 8 shows the simplified code from HTTrack, where PSet does not capture the definition-use relation on $I3$ and $I1$, and misses the order violation bug. DefUse (with DSet or LR invariants) can detect it. (3) Our work also includes LR and Follower invariants and thereby can check other properties of concurrent execution. (4) DefUse can also detect sequential bugs and work for sequential programs.

Bugaboo [20] uses context-aware communication graphs for concurrency bug detection. However, it is limited by the context size due to the exponential cost. If the granularity of the bug manifestation is larger than the context size, 5 by default, the detection may be inaccurate. For example, in Figure 1(a), if the interval between $S1$ and $S3$ in execution is larger than the context size, it may miss the bug or may report the false positive. In addition, Bugaboo needs the special hardware extension because of the expensive software implementation even for the default context size.

Our tool, DefUse, can detect not only atomicity violations, but also order violations that have been seldom studied in previous research. Using the same framework, we can also detect memory bugs and semantic bugs.

7. Conclusion

This paper proposed definition-use invariants which can be used to detect a wide spectrum of software bugs, including both concurrency bugs (atomicity and order violations) and sequential bugs (memory corruptions and certain semantic bugs). Our experimental results with 16 real-world applications and 20 real-world bugs of different types have shown that DefUse was effective in detecting 19 of them, including 2 new bugs that were never reported before, while introducing only 0-3 false positives. Our training sensitivity experiments showed that DefUse can reasonably tolerate insufficient training, especially with confidence-based pruning.

There are several possible directions for the future work. Firstly, as shown in our experiments, DefUse still fails to detect one tested bug (Section 5.6). To handle such bugs, we need to extend DefUse invariants to include constraints. Secondly, it is interesting to consider other types of definition-use invariants. Thirdly, DefUse is currently implemented entirely in software, but it is conceivable that some simple hardware extensions can significantly reduce DefUse's runtime overhead and make it possible to use DefUse to monitor production runs.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS*, June 1998.

[3] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice & Experience*, 16(12):1161–1172, 2004.

[4] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *OSDI*, 2006.

[6] The PSet of a static memory instruction M includes another instruction P only if (i) P and M are in two different threads; (ii) M is immediately dependent on P; (iii) neither of the two threads executes a read or a write (to the same memory location) that interleaved between P and M [44].

[5] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI*, 2007.

[6] T. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. In *ASPLOS*, 2006.

[7] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[9] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004.

[10] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[11] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349, 2003.

[12] H. S. Gunawi, C. Rubio-Gonzaiz, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *FAST*, 2008.

[13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[14] M. J. Harrold and B. A. Malloy. Data flow testing of parallelized code. In *ICSM*, 1992.

[15] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*, 1992.

[16] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *FSE*, 2007.

[17] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP*, 2007.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008.

[19] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[20] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[22] E. Marcus and H. Stern. Blueprints for high availability (2nd edition). John Wiley and Sons, 2003.

[23] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *PLDI*, 2009.

[24] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *Performance Evaluation Review*, 26(3): 31–37, 1998.

[25] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[26] M. Musuvathi, S. Qadeer, T. Ball, and G. Basler. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[27] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS*, 2006.

[28] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[29] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP*, 2003.

[30] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[31] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, 1996.

[32] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP*, 2003.

[33] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, pages 83–94, 2005.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997.

[35] SecurityFocus. Software bug contributed to blackout. http://www.securityfocus.com/news/8016.

[36] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[37] A. Shankar and R. Bodik. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI*, 2007.

[38] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, 2001.

[39] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object oriented programs. In *PLDI*, 2003.

[40] M. Xu, R. Bodik, and M. Hill. A regulated transitive reduction for longer memory race recording. In *ASPLOS*, 2006.

[41] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[42] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, pages 1–14, 2005.

[43] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, 1998.

[44] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

[45] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *MICRO*, 2004.