WATCHER: In-Situ Failure Diagnosis

HONGYU LIU*, Purdue University, USA
SAM SILVESTRO, University of Texas at San Antonio, USA
XIANGYU ZHANG, Purdue University, USA
JIAN HUANG, University of Illinois at Urbana-Champaign, USA
TONGPING LIU*, University of Massachusetts Amherst, USA

Diagnosing software failures is important but notoriously challenging. Existing work either requires extensive manual effort, imposing a serious privacy concern (for in-production systems), or cannot report sufficient information for bug fixes. This paper presents a novel diagnosis system, named WATCHER, that can pinpoint root causes of program failures within the failing process ("in-situ"), eliminating the privacy concern. It combines identical record-and-replay, binary analysis, dynamic analysis, and hardware support together to perform the diagnosis without human involvement. It further proposes two optimizations to reduce the diagnosis time and diagnose failures with control flow hijacks. WATCHER can be easily deployed, without requiring custom hardware or operating system, program modification, or recompilation. We evaluate WATCHER with 24 program failures in real-world deployed software, including large-scale applications, such as Memcached, SQLite, and OpenJPEG. Experimental results show that WATCHER can accurately identify the root causes in only a few seconds.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Dynamic analysis**.

Additional Key Words and Phrases: In-Situ Diagnosis, Failure Diagnosis, Root Cause Analysis

ACM Reference Format:
Hongyu Liu, Sam Silvestro, Xiangyu Zhang, Jian Huang, and Tongping Liu. 2020. WATCHER: In-Situ Failure Diagnosis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 143 (November 2020), 27 pages. https://doi.org/10.1145/3428211

1 INTRODUCTION

Software contains latent bugs [Qin et al. 2005; Tucek et al. 2007]. Although software testing helps identify these bugs, the schedule pressure often causes vendors to release software without comprehensive testing. Moreover, it is practically impossible to expunge all bugs of large software via testing, especially for concurrency bugs [Lu et al. 2007]. Consequently, bugs inevitably escape the in-house testing phase and lurk into the production phase [Sahoo et al. 2010], which may cause system crashes, program hangs, or security breaches [Szekeres et al. 2013]. Based on the existing study, software failures have led to 1.7 trillion financial losses in 2017 alone [Freyja 2017].

*This work was initiated and partially conducted while Hongyu Liu and Tongping Liu were at the University of Texas at San Antonio.

Authors' addresses: Hongyu Liu, liu2978@purdue.edu, Purdue University, USA; Sam Silvestro, sam.silvestro@utsa.edu, University of Texas at San Antonio, USA; Xiangyu Zhang, xyzhang@cs.purdue.edu, Purdue University, USA; Jian Huang, jianh@illinois.edu, University of Illinois at Urbana-Champaign, USA; Tongping Liu, tongping@umass.edu, University of Massachusetts Amherst, USA.

2475-1421/2020/11-ART143
https://doi.org/10.1145/3428211

Therefore, it is extremely important to diagnose in-production software failures [Kasikci et al. 2015; Tucek et al. 2007]. However, in-production software users are typically not experts, who do not have the expertise or willingness to debug the in-production failures. On the other hand, software developers are usually not able to access the production environment, which significantly limits their capability of diagnosis. As shown by a Quora poll, "a coder's worst nightmare" is "the bug only occurs in production and cannot be replicated locally" [Quora 2015].

To diagnose software failures, one approach is to collect various execution events, then search for the symptoms of bugs, such as deadlocks [Joshi et al. 2009], race conditions [Choi et al. 2002; Savage et al. 1997; Zhang et al. 2017], or different concurrency bugs [Jin et al. 2010]. They can even detect latent bugs that do not cause explicit behavior. However, they are tied to the collected events, which cannot diagnose general failures. Another approach is program slicing [Korel and Laski 1988; Sahoo et al. 2013; Wang et al. 2014; Zhang et al. 2003], especially thin-slicing [Musuvathi et al. 2008], which identifies all relevant statements to a seed statement [Musuvathi et al. 2008]. However, even thin-slicing may include many unnecessary statements caused by imprecise pointer analysis and the lack of runtime information.

Recently researchers proposed to perform offline or postmortem analysis after crashes or failures, based on memory core dumps and execution records [Cui et al. 2018, 2016; Glerum et al. 2009; Kasikci et al. 2017, 2015; Machado et al. 2015a; Xu et al. 2016, 2017]. They could even reconstruct the history of a failing execution [Cui et al. 2018], enabling offline debugging for in-production failures. However, offline approaches share multiple unsolvable shortcomings. *First*, they may leak privacy information [Cui et al. 2018, 2016; Glerum et al. 2009; Kasikci et al. 2015; Xu et al. 2016, 2017], especially for approaches relying on memory core dumps [Cui et al. 2018, 2016; Glerum et al. 2009; Kasikci et al. 2015; Xu et al. 2016, 2017]. For example, if a browser crashed after a user just logged into a bank account, then the core dump may leak the account name and password. *Second*, offline diagnosis cannot provide the timely guidance for online failure prevention, but fixing a bug may take multiple weeks or even months [Godefroid and Nagappan 2008; Godefroid, Patrice and Nagappan, Nachi 2008; Yin et al. 2011]. *Third*, offline analysis cannot diagnose failures related to unavailable third-party libraries, where the diagnosis may be forced to stop. *Last*, offline approaches may not diagnose some failures related to the environment (e.g., system states). A real example is shown in Figure 1. The program crashes when the `read` system call (*L*4) overwrote the stack with the `len` larger than 1024. Therefore, it is critical to know the size of `len` to understand the overflow. However, *it is impossible to know offline without recording system call results*, since the whole stack (including the value of `len`) was corrupted by the stack overflow and registers were also destroyed by the stack pop operation.

This paper proposes a novel method to diagnose software failures within the failing process, also called "**in-situ diagnosis**", and designs such a system—WATCHER. WATCHER is built on top of an existing record-and-replay system–IR [Liu et al. 2018], which divides the whole execution into multiple epochs (e.g. irrevocable system calls) and supports endless re-executions of the last epoch. WATCHER focuses on explicit program crashes that will generate explicit failure signals, such as `SIGFPE`, `SIGSEGV`, and `SIGABRT`, which triggers the on-demand diagnosis within re-executions upon crashes. WATCHER is based on **a key observation** of program crashes: *many program crashes are typically caused by writing an incorrect/invalid value to a memory unit*. Based on this observation, WATCHER aims to identify the origin of the failing value and reports data-dependent slices of the failing value, sharing the same target as thin-slicing [Musuvathi et al. 2008]. WATCHER takes advantage of the "in-situ" environment to solve the imprecise pointer analysis and the lack of runtime information of static analysis utilized by existing work, such as thin-slicing [Musuvathi et al. 2008].

In order to identify the value propagation chain, *Watcher employs the watchpoint mechanism to track all memory writes (or the data flow) of a memory unit exactly within identical re-executions*.

```
int tcp_test(char* ip_str, const short port)
{
L1:   unsigned char packet[1024];
      int len;
      struct net_hdr nh;
      ......
      while(1) {
        ......
L2:     caplen = read(sock, &nh, sizeof(nh));
            ......
L3:     len = ntohl(nh.nh_len);
        ......
L4:     caplen = read(sock, packet, len);
        ......
      }
}
```

Fig. 1. Code snippet for a real-world bug (Aireplay-ng).

Therefore, only the instructions that directly contribute to the seed memory unit will be captured, omitting all irrelevant instructions. Since a memory unit (e.g., a stack variable or heap object) can be re-utilized for different purposes, WATCHER further proposes **last-win** and **value-confirmation** mechanisms to ensure its correctness and simplify its diagnosis. The last-win mechanism indicates that only the last-write operation of each thread will be considered (or win), since all previous writes will be overwritten by the latest write. WATCHER also utilizes the value of a write operation to prune irrelevant write instructions. For example, WATCHER only focuses on instructions writing the zero value to a specific pointer, if the crash is caused by a NULL pointer dereference failure. More details of these mechanisms are described in Section 2.

Watcher *proposes a hybrid analysis to determine a specific memory unit that contributes to a crash*. Basically, it utilizes the binary analysis to identify all possibilities, and then confirms the actual one with dynamic analysis, with the assistance of another debugging method—breakpoints. Debugging breakpoints are employed to collect the control flow information. WATCHER places breakpoints on all possible instructions, and then employs the "last-win" rule to prune irrelevant instructions/branches (see Section 2). Since modern hardware only has a limited number of breakpoints that may not cover all possibilities at a time, which will lead to unnecessary re-executions and therefore longer diagnosis time unnecessarily, WATCHER further proposes to employ software breakpoints to reduce the number of re-executions, and utilizes dynamic emulation to address possible race conditions. When the tracing hardware (e.g., Intel Processor Trace) is available, WATCHER could further employ it to further reduce the number of re-executions and eliminate false positives caused by control-flow hijacks that the executions will be redirected to a target location that cannot be reached in a normal execution [Xu et al. 2017]. Please refer to Section 2.2 for more discussion.

As a drop-in library, WATCHER can be deployed easily with the preloading mechanism. There is no need to modify the underlying operating system, change or re-compile the source code of applications, or use non-existent hardware. WATCHER invokes its failure diagnosis automatically upon failures, without human involvement. It relieves the pain of reproducing failures due to the execution environment. Further, in-production software may not contain source code or even symbol information in the binary, preventing the use of some static analysis tools directly. WATCHER employs dynamic analysis at the binary-level to infer the value propagation chain. Compared to offline analysis (e.g., REPT [Cui et al. 2018]), Watcher may seem to be more intrusive. However, it preserves the privacy by only reporting root causes (instead of the whole memory image), and improves its effectiveness due to its in-situ diagnosis.

WATCHER can be utilized in development phases, staging deployment phases, and production phases, as discussed in Section 6. WATCHER is able to diagnose a range of failures, such as segmentation faults, assertion failures, aborts, divide-by-zeros, and floating-point failures. We evaluated WATCHER with 24 bugs of real applications, including Memcached, SQLite, and OpenJPEG. Our experiments demonstrate WATCHER's effectiveness and performance. Overall, this paper makes the following contributions:

- It proposes the first in-situ diagnosis that can identify software failures in the failing process, overcoming multiple issues of offline analysis or static analysis.
- WATCHER proposes to employ the in-situ environment to improve its control and data dependence analysis, with last-win and value-confirmation mechanisms in particular.
- WATCHER proposes a systematic method that combines binary analysis and debugging methods together to perform dynamic slices efficiently within re-executions, where high-overhead diagnosis is only triggered upon failures.
- WATCHER further proposes to improve the efficiency and correctness of in-situ failure diagnosis with software breakpoints and hardware-assisted trace.
- Extensive experiments are performed to confirm that WATCHER is able to diagnose a range of failures in a short time.

Outline. The rest of this paper is organized as follows. Section 2 presents some formal definitions and the overview of WATCHER. Section 3 describes the detailed design and implementation of WATCHER. We discuss the optimization techniques in Section 4. Section 5 shows the evaluation of WATCHER, and Section 6 discusses its limitations. We discuss the related work in Section 7, and conclude the paper in Section 8.

2 WATCHER OVERVIEW

In this section, we start with definitions of several key concepts and discuss multiple observations on program failures. Then we discuss the basic idea of WATCHER and possible technical challenges/solutions based on previous discussion.

2.1 Definitions and Observations

WATCHER aims to identify root causes of program failures that will generate explicit failure signals, such as `SIGFPE`, `SIGSEGV`, `SIGPIPE`, and `SIGABRT`. But a program failure may have multiple root cause candidates [Zhang et al. 2019]. As shown in Figure 2, an assertion failure is triggered if the configuration flag (`CONFIG`) leads to the execution of a buggy instruction (line $L2$). That is, both the `CONFIG` flag and line $L2$ are root cause candidates for this failure. Therefore, it is important to define which type of root cause WATCHER can identify.

This paper adopts the definition from Wilson et al. [Wilson 1993]: "**root cause is the most basic reason for an undesirable condition or problem which, if eliminated or corrected, would have prevented it from existing or occurring**", similar to existing work [Bond et al. 2007; Musuvathi et al. 2008; Zhang et al. 2019]. We further redefine it to the context of program crashes that WATCHER focuses on.

DEFINITION 1. *The **root cause** is a write instruction (that writes an illegal value) in the crashing execution (or "origin" as defined in [Bond et al. 2007]), where correcting the value of this write operation will prevent the failure from occurring.*

This definition also confirms the original definition that the root cause is the most basic reason for a crash. For the failure in Figure 2, WATCHER will report the buggy instruction (at line $L2$) as the

```
L1: if(CONFIG)
L2:   y = 0;
    ......
L3: assert( x != 0 && y != 0 );
    ......
```

Fig. 2. An assertion failure.

```
64e: mov    0x2009c8(%rip),%eax        # <CONFIG>
654: test   %eax,%eax
656: je     662 <main+0x18>
658: movl   $0x0,0x2009b2(%rip)        # <y>
65f: 00 00 00
662: mov    0x2009a8(%rip),%eax        # <x>
668: test   %eax,%eax
66a: je     676 <main+0x2c>
66c: mov    0x2009a2(%rip),%eax        # <y>
672: test   %eax,%eax
674: jne    695 <main+0x4b>
676: lea    0xcb(%rip),%rcx
67d: mov    $0xf,%edx
682: lea    0x9f(%rip),%rsi
689: lea    0xa1(%rip),%rdi
690: callq  520 <__assert_fail@plt>
```

Fig. 3. Assembly code of assertion failure of Fig. 2.

root cause since it is the origin of the failing value, instead of the instruction setting `CONFIG` flag to be `true`. That is, writing 0 to y is the more basic reason that actually triggers the assertion failure, while the `CONFIG` flag is only the indirect reason that y is set to 0.

This definition also defines which type of failures WATCHER is able to diagnose. That is, Watcher *can identify failures that are caused by writing a wrong value. But it is not able to diagnose failures caused by not executing a specific instruction*. For the clarity of the description, we further define *failing instruction* and *failing condition* as follows.

DEFINITION 2. *The **failing instruction** is the instruction that a crash actually happens at, while the **failing condition** is the value of a memory unit or a register that makes a program crash.*

For Figure 2, the failing instruction is the assertion failure instruction located at statement $L3$ (at location 690 in Figure 3), and the failing condition is y's value (set to be 0). We have multiple observations based on these definitions and two previous examples.

First, sometimes the relationship between failing instruction and failing condition is not straightforward, which highly depends on the type of failure and the type of failing instruction. For the assertion failure as Figure 2, Figure 3 further shows the assembly code for this failure. For this failure, the failing instruction is located at location 690, where the assertion is triggered. But the failure condition is that y's value was set to be 0, which is tested at location $66c - -672$ and is set at location 658. Therefore, a diagnosis tool is required to connect failing instruction with failing condition, and then identify the root cause. For some crashes, such as segmentation faults, the failing condition is within the failing instruction.

Second, the failing condition is different from the root cause. For Figure 2, the failing condition is that y was set to be 0, while the root cause is the instruction that sets y to be 0 (at line $L2$). In order to explain other observations, we further define the *level* of root cause in the following.

DEFINITION 3. *An instruction performing data transfer towards the value of the failing condition is treated as a level of root cause.*

Third, the root cause of some failures are deeply hidden, which may require multi-level diagnosis. For the failure in Figure 1, line $L3$ is the first level root cause that crashes the return address. However, it cannot help the fix by knowing the first level of root cause. For this failure, the final root cause is located at line $L2$, where the len value is set to be larger than the $packet$'s length (1024) due to a `read` from a socket. Knowing the root cause is critical to the bug fix, then users should check

the validity of *len*, since it is read from a socket. WATCHER keeps track of the origins of a root cause, since the value may be transferred from another source, which is the reason for multi-level diagnosis. In the end, WATCHER reports all instructions that the failing instruction is transitively data-dependent, i.e., the backward dynamic data dependence slices.

Fourth, some failures may have multiple root causes, which are called multi-variable issues. A simple example is an assertion failure like `assert(x < y)`, where both x and y could be the reason of the failure. For such failures, WATCHER reports the value propagation chain for both variables, letting programmers to confirm the root cause based on the semantics of the program.

2.2 Basic Steps and Technical Challenges

Based on the definitions and observations of Section 2.1, WATCHER's diagnosis include the following steps. We will employ the assertion failure of Figure 2 as the example.

2.2.1 Identifying Failing Condition. *First,* Watcher *identifies the failing condition starting from the failing instruction*. For the example, the failing instruction is where the assertion is triggered (at location 690). However, this instruction does not contain the failing condition. Instead, the failing condition can be either $x = 0$ or $y = 0$. In order to understand this, WATCHER must identify whether x or y triggers the assertion failure, which are located on two different branches (at location $66a$ and location 674). That is, there exists a technical challenge in this step.

TECHNICAL CHALLENGE 1. *How can we identify the failing condition, starting from the failing instruction?*

In order to identify the failing condition, we could employ binary analysis, based on register values. But sometimes it is not always easy to do that. For the example of Figure 1, the register value related with *len* is already corrupted by the stack overflow. Even for the example in Figure 2, the failure signal is actually triggered inside the function __assert_fail, where all registers have been changed due to the function invocation. Therefore, WATCHER proposes a general solution: *WATCHER utilizes the binary analysis to identify all possibilities (e.g., all branch instructions), then utilizes the dynamic re-execution to confirm the real one via using the breakpoints*. More specifically, WATCHER installs breakpoints on all possible branches, and proposes the **last-win** rule to determine the failing branch as follows, if the execution can be identically reproduced.

RULE 1. *When there are many branches in a function, the last-taken branch just before the failure possibly includes the failing condition.*

This rule holds for all bugs studied in Section 5. To determine the last-taken branch, WATCHER analyzes all possible branches in the failing function, and installs the breakpoints to collect the order of branch-taken (control flow). By handling the exceptions caused by executing these breakpoints, WATCHER is able to determine the last-taken branch. However, there exists another challenge.

TECHNICAL CHALLENGE 2. *Modern hardware only has four debugging registers that can be utilized as either breakpoints, which may not be able to cover all branches of one function.*

WATCHER further proposes to use software breakpoints to overcome this limitation, since software breakpoints can be installed without the limit. Section 4.1 further discusses more details on this, especially avoiding race conditions of installing/removing breakpoints.

For the above method, WATCHER further assumes that a program cannot jump into the middle of a function directly from other functions. Otherwise, identifying all branches in the current function is not sufficient to cover all possibilities. However, this assumption is broken under control flow hijacking [Xu et al. 2017], since the execution can be jumped from a random placement, not in the

current function. That is, there is no way to identify all possible placements that the execution can be jumped from, which is another technical challenge.

TECHNICAL CHALLENGE 3. *How can we deal with the control flow hijacking where identifying all possibilities inside the failing function is not sufficient?*

For this challenge, WATCHER utilizes hardware-assisted trace to assist the diagnosis, when special hardware such as Intel's Process Trace (PT) exists. Hardware trace will allow WATCHER to infer the branching instruction statically. Different from existing work, WATCHER utilizes the PT differently: it only enables the trace during the re-execution phases after a program crashes; it avoids decoding some traces to reduce the diagnosis time. More details are further discussed in Section 4.2.

2.2.2 Identifying Important Memory Unit. After identifying the failing condition that is typically related to a register, *the second step of* Watcher *is to identify the origin of the register value*. That is, it should identify the corresponding memory unit so that it is able to trace the root cause. But it has a similar issue as Technical Challenge 1.

TECHNICAL CHALLENGE 4. *How can we identify the origin of a register, when there are multiple assignment instructions due to register reuse?*

One intuitive approach is also to infer this via binary analysis. However, it has the same issue as the first step, due to incomplete states about registers. WATCHER proposes a general solution for this step. *WATCHER utilizes the binary analysis to find all assignment instructions to this register, then confirms the real one within re-executions with the assistance of breakpoints*. Similarly, WATCHER further proposes the "**last-win**" rule and "**value-confirmation**" to prune unnecessary register assignments.

RULE 2. *When there are multiple assignment instructions assigning values to a register, only the last assignment assigning the failing value to this register is the one related to the failing condition.*

2.2.3 Identifying Root Cause. The third step is to identify the instruction that actually writes the value to the failing memory unit. If the failing register value comes from an immediate number, WATCHER will report the assignment instruction as the root cause. For other situations, *WATCHER proposes to utilize the hardware watchpoint to collect all memory accesses on the specific memory unit*. Similarly, it is possible that there are many accesses on the same memory unit, especially for multithreaded applications. *WATCHER further prunes unnecessary writes using the "last-win" and "value-confirmation" rules*.

RULE 3. *When there are multiple writes on the same memory unit, the previous writes will be overwritten by the latest one in the same thread. Therefore, only the last write instruction that writes the failing value to the specified memory address is considered to be the root cause.*

When there are multiple accesses from different threads, WATCHER infers the happens-before relationship between these accesses, and then confirms the root cause as described in Section 3.5.

2.2.4 Multi-level Root Cause Diagnosis. As described above, sometimes multi-level root diagnosis is required to identify the root cause of a failure. For this step, a research question is when should WATCHER stop the diagnosis?

WATCHER continues its multi-level root diagnosis until meeting with the following condition: the value is from an immediate value, an input parameter of the program, or the result of a system call, or WATCHER reaches the beginning of the current epoch (e.g. an irrevocable system call defined in iReplayer [Liu et al. 2018]). Overall, WATCHER utilizes multiple re-executions to gradually identify the root cause, starting from the failing instruction.

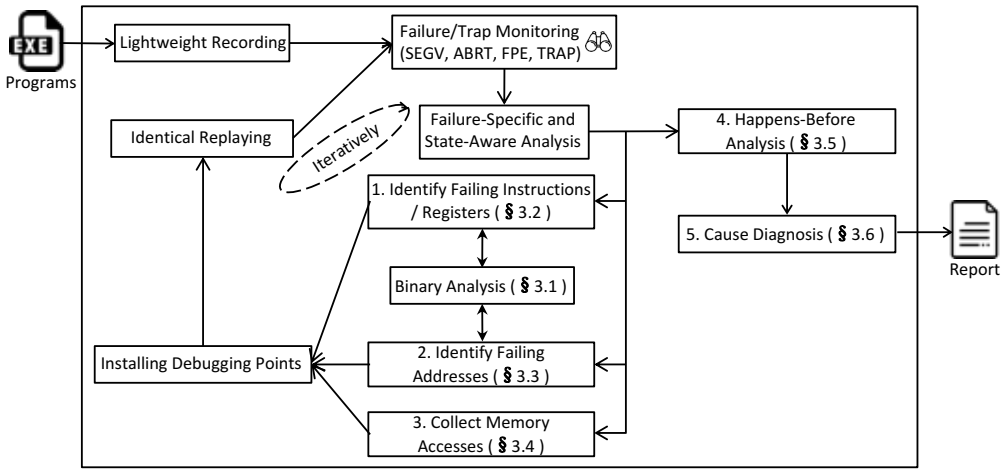

Fig. 4. System overview of WATCHER.

2.3 Basic Idea

Overall, WATCHER includes multiple components as shown in Figure 4. WATCHER performs a lightweight recording during normal executions. The failure diagnosis is invoked on-demand, when the "Failure/Trap Monitoring" unit detects a failure signal. Inside the signal handler, WATCHER performs the "Failure-Specific and State-Aware Analysis" to decide the next step. Typically, it first identifies the failing instruction, and then identifies the failing condition, with the assistance of binary analysis and debugging points. With the failing condition information, it further determines the relevant memory unit. After identifying the failing memory address, it further installs the watchpoint on the failing memory address in order to collect all memory references. As described above, WATCHER uses "last-win" and "value-confirmation" to prune unnecessary writes. If this still does not exclude all irrelevant instructions, WATCHER further utilizes the happens-before relationship between multiple threads to rule out unnecessary ones. In the end, WATCHER reports the value propagation chain to the users to assist bug fixes, which includes the root cause.

Basically, WATCHER employs multiple executions to identify the root cause gradually, with the combination of binary analysis and hardware debugging points. An alternative method is to collect the control and data flow via dynamic instrumentation or single-step executions within one re-execution, then perform static analysis to identify the root cause. However, this method unfortunately suffers from both correctness and efficiency issues. First, efficient record-and-replay systems, such as Castor [Mashtizadeh et al. 2017] and iReplayer [Liu et al. 2018], only guarantee the weak determinism that strong interference, like dynamic instrumentation or single-step execution, may make it impossible to produce the original execution. WATCHER employs the breakpoints to collect a small portion of control flow, and the watchpoints to collect the data flow on interesting addresses, avoiding the strong interference to the re-execution. Second, performing static analysis on a large number of instructions could be extremely slow [Huang et al. 2013; Kasikci et al. 2017], which is not suitable for in-situ diagnosis, since users are waiting on its completion. In contrast, WATCHER employs dynamic re-executions to avoid the complexity and slowness of static analysis, exploiting the advantage of being in the in-situ environment.

3 DESIGN AND IMPLEMENTATION

WATCHER is implemented on top of iReplayer [Liu et al. 2018] and the XED2 library [Intel 2017], which includes an additional 8,800 lines of code. The details are further discussed in the following.

3.1 Binary Analysis

WATCHER chooses Intel's XED2 as the basis for its binary analysis because of its widespread usage and the abundant information it provides [Intel 2017]. For instance, XED2 produces a data structure for each instruction that describes the opcode, operands, flags, and instruction type.

One challenge comes from the variable-length instructions of X86 machines, making it impossible to disassemble correctly when starting from a position that is not aligned with the real instruction. One naive method is to disassemble all instructions at one time, and then save information of all these instructions. However, the abundant information of each instruction may impose unnecessary memory overhead (192 bytes per instruction), especially when only a small portion of instructions will actually be analyzed. Instead, WATCHER disassembles all instructions at once, but saves only the starting address of every instruction instead. Therefore, this method significantly reduces its memory consumption, but still allows it to obtain the starting address correctly and perform the analysis on demand.

3.2 Identifying Failing Instructions and Registers

WATCHER obtains the failing instruction directly from the failing context, but identifying the failing register depends on the type of failures. For some failures, such as `SIGSEGV` and `SIGFPE` failures, it could utilize the binary analysis to identify failing registers inside. For instance, if a program encounters a segfault when accessing an invalid address, then the target register of the failing instruction will be the failing register. For divide-by-zero failures, WATCHER only needs to collect the division register inside the divide instruction.

However, for `SIGABRT` failures, the failing register is not inside the failing instruction. Instead, WATCHER utilizes a hybrid analysis to identify the failing register. Binary analysis is employed to identify all branch instructions in the current function, and then breakpoints are installed in order to track the control flow. Upon each trap, WATCHER records the calling context of the current trapping instruction. Then, the program is instructed to proceed forward. If the abort occurs subsequently, then the failing condition must be located in the comparison instruction before the branch instruction. WATCHER identifies the corresponding failing register(s) with binary analysis.

In some failures, multiple registers may be considered as the failing registers. For example, if a program aborts when $x < y$, then both registers related to x and y need to be tracked. WATCHER reports root causes of these two registers so that programmers can determine the root cause for the failure, based on the program's semantics.

3.3 Identifying Failing Addresses

After determining failing register(s), WATCHER should identify the origin of the failing register. Based on our observation, a register's value may come from one or more registers, a memory address, an immediate value, or an input parameter of a function. WATCHER takes different actions correspondingly: (1) If a register value is assigned from an immediate value (e.g., a constant value), the diagnosis will be stopped immediately, and WATCHER reports this assignment instruction as the root cause. (2) If a register value is from another register, WATCHER turns to track the new register instead. (3) If a register value comes from multiple registers, then all of these registers should be tracked afterward using the same procedure. (4) If the value is from a memory address, WATCHER starts to track the instruction that changes the memory unit to the current value, as discussed in

Section 3.4. (5) If the register value is coming from one input parameter of the current function, WATCHER enters the caller function and continues to track the register assignment there with the same procedure.

To track the origin of a register, WATCHER employs the combination of binary analysis, identical re-executions, and debugging registers. The binary analysis is employed to identify all assignment instructions with the susceptible register as the target register. After that, breakpoints are installed on these instructions. WATCHER confirms the real assignment using the last-win rule: the last register assignment prior to the failure that assigns the specified value to the failing register.

When a program failure involves multiple registers, a multiple-variable failure, WATCHER will identify the origin of all registers. Although it is possible to track the origins of multiple registers altogether, this method increases the difficulty of handling each trap due to the following reasons: (1) WATCHER should identify which register assignment causes this trap and determines which step to proceed; (2) Each step may require multiple hardware breakpoints, which does not allow the tracking of multiple registers altogether; (3) WATCHER's analysis may have different progress, since the value of the register can be assigned differently, as discussed above. Therefore, WATCHER chooses an easy way to identify the failing addresses: *it only tracks the origin of one register at a time*. This mechanism may increase the number of re-executions for its failure diagnosis, but reduces the implementation difficulty.

3.4 Collecting and Pruning Memory Accesses

Based on the definition of root cause, WATCHER only requires to track instructions issuing memory writes on specific memory units. WATCHER employs hardware watchpoints to track the data flow. By installing watchpoints on the failing addresses before the re-execution, WATCHER collects all information of each memory write within re-executions, including the value after the access, the thread information, and the calling context within the traps.

WATCHER solves the issue caused by memory re-utilization automatically. It tracks the last write operation of each thread, by configuring each thread to handle its own traps correspondingly. A later write operation will always overwrite its prior ones, which eases the happens-before analysis by reducing unnecessary writes. In addition, tracking the last write naturally fixes the address re-utilization issue in which a heap object will be re-utilized after its deallocation.

3.5 Happens-Before Analysis

After collecting the last-write accesses of each thread, WATCHER performs happens-before analysis to infer the order of memory accesses, which is important to determine the root cause for concurrency errors.

For the happens-before analysis, WATCHER mainly focuses on identifying the happens-before relationship for accesses from different threads, as shown in Figure 5. WATCHER utilizes the synchronization order that has been recorded by its record-and-replay framework to perform its happens-before analysis. During the execution, WATCHER maps each memory access to the corresponding synchronization. Based on the order of synchronizations, WATCHER can easily infer the order of memory accesses. If there is a strong happens-before relationship, for example, two memory accesses are protected by the same lock, the later access directly overwrites the previous ones. If there is no happens-before relationship, which may be caused by a data race, WATCHER further employs the value of the write operation to determine the root cause.

A simple example is illustrated in Figure 5, which shows how WATCHER records the order of memory references before the analysis. For this example, $lock1$ was acquired by Thread1, then by Thread2, and $lock2$ was first acquired by Thread1. WATCHER tracks the locks currently held by each thread, which allows it to further differentiate whether a memory access occurs within the

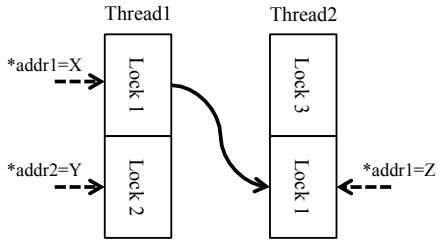

Fig. 5. Happens-before relationship.

protection of locks. In Figure 5, memory references occurring under a lock protection (e.g. $W1\_1$ and $W2\_1$) will be placed within the synchronization event, while memory references outside all locks (e.g. $W1\_2$) will be placed between events. Based on the relationship, WATCHER infers the happens-before relationship between each memory access [Flanagan and Freund 2009; Lamport 1978; Savage et al. 1997]. WATCHER checks whether the corresponding threads are synchronized with the same lock. If yes, then it could draw a happens-before relationship via the order of lock events. For instance, Figure 5 shows that $W1\_1$ happens before $W2\_1$. Based on the happens-before relationship, WATCHER prunes all unnecessary memory references in order to simplify root cause analysis. If there is no happens-before relationship between two accesses on the same memory address, then it can be caused by race conditions.

3.6 Root Cause Analysis and Report

After performing happens-before analysis, WATCHER conducts root cause diagnosis in the following steps: (1) It determines whether the failure is caused by a race condition or not; if the corresponding write is performed by a different thread and there is no happens-before relationship, this failure is reported as a race condition. (2) It determines whether to perform further diagnosis, or generate a failure report. WATCHER performs multi-level diagnosis automatically as further described in Section 3.7.

For the failure report, WATCHER reports the relevant write instructions, their corresponding values, and the explicit synchronizations between threads if a failure is related to a concurrency issue. A failure is a concurrency issue when the last write and the failing read are from different threads. Currently, WATCHER reports root causes of software failures on the screen, which could be extended to report to a persistent file or send a report to programmers in the future.

Note that if the symbol information is not available in the production environment, WATCHER only reports the addresses of instructions (instead of the line number information). Programmers could determine the line number easily using utilities such as `addr2line`, using the same binary with symbol information included.

3.7 Performing Multi-level Diagnosis

As described in Section 2.2, some failures may require multi-level failure diagnosis to identify the final root cause. For instance, if a program crashes due to dereferencing the ptr field of object A, e.g. $A{\rightarrow}ptr$. However, $A{\rightarrow}ptr$ was copied from another object B. In this case, WATCHER will perform a multi-level diagnosis in order to identify dynamic data dependence slices. WATCHER only stops the diagnosis when the memory or register assignment originates from an immediate value, an input parameter of the application, or the result of a system call (e.g. gettimeofday), or when the diagnosis reaches the beginning of the last epoch.

The diagnosis beyond the first level is actually more straightforward, since it only requires identifying the instruction that writes to a memory address. It does not require the determination of

the failing instruction and register, since the last write instruction will be the starting point for its next level diagnosis. Basically, WATCHER simply installs the watchpoint to the specified address, then tries to collect memory writes on that address within a new re-execution.

For multi-level diagnosis, WATCHER performs the diagnosis level-by-level, and only tracks memory references and instructions prior to its previous level. Since one instruction can be executed multiple times, WATCHER further utilizes the sequence number of memory accesses to differentiate these accesses. WATCHER stops the re-execution when the memory unit has been accessed with the expected number of times that is the same as its sequence number.

3.8 Record-and-Replay System

As described before, WATCHER relies on an record-and-replay framework to identify susceptible accesses or instructions within re-executions. Therefore, its record-and-replay module has two requirements. First, the re-execution is identical to the failing execution, since WATCHER employs the watchpoint to track the data flow. Second, the recording overhead should be sufficiently small, in order to be employed in the production environment. Due to these reasons, WATCHER leverages iReplayer as its record-and-replay engine [Liu et al. 2018], since iReplayer supports identical re-executions with 3% recording overhead.

iReplayer divides the execution into multiple epochs, based on the memory requirement and the irrevocable system calls, and only reproduces the last epoch of execution if necessary. This indicates that WATCHER could only identify the root cause of a failure if it is located in the last epoch.

In order to support re-executions, iReplayer periodically takes snapshots of program states, records the results of system calls that cannot be reproduced identically, and records the order of synchronizations. For the performance reason, iReplayer does not record the order of memory accesses, while the handling of race conditions is delayed to its re-execution phase as described below.

iReplayer's re-execution takes multiple steps. It first recovers some changeable regions of the memory, such as global memory and heap memory. Then it lets different threads recover their own stacks before invoking `setcontext()` to reset their local registers. During the re-execution, the order of synchronizations and system calls should be preserved in order to ensure the identical replay. iReplayer assumes that the divergence from the recorded events can be only caused by race conditions, when all other events are reproduced deterministically. Therefore, iReplayer utilizes multiple replays to search for an identical schedule under race conditions. As shown in their paper [Liu et al. 2018], iReplayer has a large probability (over 99%) to reproduce racy executions in the first re-execution.

Since iReplayer does not record the order of memory accesses, it only supports the weak determinism. That is, it may easily lead to a divergence, when there is strong interference in the re-execution. That is the reason why WATCHER does not employ single-step execution or full-instrumentation to get all execution details, as further discussed in Section 2.3. Instead, WATCHER employs hardware debugging points that will provide little interference to the re-execution. When there is only little interference on the execution, WATCHER is expected to achieve the identical re-executions easily.

4 OPTIMIZATIONS

As described in Section 1, WATCHER proposes two optimizations, software breakpoints and hardware tracing, to further reduce the analysis time and ensure the correctness.

4.1 Software Breakpoints

Existing hardware only has four hardware debugging registers, which may lead to a large number of re-executions unnecessarily, as described in Section 3.2. This issue can be greatly reduced with software breakpoints, since an infinite number of software breakpoints can be installed at the same time.

Software breakpoints rely on debugging instructions, e.g. the `INT 3` instruction (with opcode `0xCC`) in X86 machines. Executing such instructions generates a software interrupt (with the `SIGTRAP` signal). To install a breakpoint, the first byte of an instruction is rewritten to `0xCC`, and the original instruction should be recorded. Inside the signal handler, the original instruction needs to be restored. When the instruction and the program counter is reset, the program run as normal. To capture future execution on the target instruction, we need to place the breakpoint back after executing the current instruction. However, the naive implementation may impose race conditions for the multi-threaded programs. If another thread executes the target instruction before the recovery of a breakpoint, WATCHER will fail to capture this execution that may generate a wrong report, since WATCHER's last-win mechanism relies on the order of accesses.

WATCHER proposes the "instruction emulation" to eliminate this issue: instead of removing and recovering `0xCC` in the instruction, WATCHER emulates the results of its original instruction directly by setting registers or memory addresses correspondingly. For instance, if the original instruction is "`mov -0x18(%rcx),%rbx`", WATCHER simply moves the value of the memory unit to `rbx` register directly, without removing and recovering of the breakpoint. After that, it will advance the program counter to the next instruction directly. But the implementation challenge is to support all of the possible instructions. In the implementation, we encountered another issue when an instruction contains segment registers. Segment registers cannot be read directly inside the userspace. Instead, WATCHER obtains the values of segment registers via the `arch_prctl` system call.

4.2 Hardware-Assisted Trace

Multiple re-executions are required to diagnose the root cause of a program failure, even with the help of software breakpoints. If each re-execution is sufficiently long, WATCHER may significantly increase the diagnosis time when a lot of re-executions are needed, since WATCHER's diagnosis time is proportional to each re-execution. Also, software-based approaches cannot ensure the correctness of diagnosis in case of control flow hijacking, as discussed in Section 2.2.

WATCHER further proposes to employ hardware-assisted trace to reduce the number of re-executions, and also ensures the reliability in case of control-flow hijacking. Modern hardware, such as Intel's Processor Trace (PT), could be utilized to record control flow with very low overhead [Cui et al. 2018; Xu et al. 2017]. With this hardware, the control flow information can be obtained once, instead of using the breakpoints to get the control flow multiple times. Beyond the one-time re-execution to collect the control flow, two more executions are required for one level of root cause: one re-execution to determine the failing address, and another re-execution to collect the data flow on each memory address (with the watchpoint). Therefore, WATCHER bounds the number of re-executions to $2X + 1$, where X is the number of levels for its root cause.

WATCHER solves multiple implementation challenges as well. The first challenge is related to its performance, since it will take a long time to decompress the whole trace. WATCHER decompresses the trace on demand, in order to reduce the time spent in decoding the trace. Second, some conditional instructions (e.g., `CMOVEcc`) are not recorded by the PT hardware, which may cause incorrect diagnosis results. WATCHER solves this issue with the assistance of breakpoints, where it confirms these conditions during the next re-execution. Third, one instruction can be executed multiple times, which appears in the trace multiple times. To avoid ambiguity, WATCHER maps the execution of these instructions back to the per-thread synchronization events as described in Section 3.5 to eliminate the ambiguity.

Table 1. Effectiveness evaluation of different techniques of WATCHER

Application	Reference	Description	Type	Level	Var.	Inst.	Time (s)	Software Breakpoint		Hardware Trace	
								Rerun (#)	Time (s)	Rerun (#)	Time (s)
Aireplay-ng	CVE-2014-8322	Stack overflow	Seq.	2	1	2	0.165	13	1.31	5	3.02
Aubio	CVE-2017-17054	Divide-by-zero	Seq.	3	1	3	0.164	16	3.12	6	3.61
Cppcheck-148	Bugbase [Kasikci et al. 2015]	Null pointer	Seq.	1	1	1	0.14	4	0.86	3	1.41
Cppcheck-152	Bugbase [Kasikci et al. 2015]	Null pointer	Seq.	1	1	1	0.157	5	0.93	3	1.48
Curl-721	Bugbase [Kasikci et al. 2015]	Null pointer	Seq.	1	1	1	0.172	4	0.39	3	0.52
Exiv2	CVE-2018-9303	Assertion	Seq.	3	1	4	0.134	11	1.91	7	5.29
Gas	CVE-2005-4807	Stack overflow	Seq.	3	1	3	0.14	13	2.82	7	4.15
Gif2png	CVE-2009-5018	Stack overflow	Seq.	2	1	2	0.141	10	1.49	5	2.13
HTTrack	Issue #20247	Null pointer	Con.	1	1	1	2.167	3	6.26	3	6.56
Libming	CVE-2018-7875	Null pointer	Seq.	1	1	1	0.161	3	0.26	3	0.5
Libpng	CVE-2004-0597	Stack overflow	Seq.	2	1	2	0.134	4	0.87	5	2.89
Libtiff	CVE-2017-7595	Divide-by-zero	Seq.	3	1	3	0.158	18	1.91	7	2.72
Memcached	CVE-2011-4971	Invalid address	Seq.	4	3	11	0.22	62	4.75	21	11.72
Nasm	CVE-2004-1287	Stack overflow	Seq.	3	1	3	0.157	13	2.77	7	3.99
Openjpeg	CVE-2016-7445	Null pointer	Seq.	1	1	1	0.139	8	2.1	3	1.37
Pbzip2	Bugbase [Kasikci et al. 2015]	Null pointer	Con.	1	1	1	0.314	4	0.73	3	1.06
Pfscan	Symbiosis [Machado et al. 2015b]	Assertion	Con.	1	1	1	0.163	9	1.49	3	2.21
Polymorph	Bugbench [Lu et al. 2005]	Stack overflow	Seq.	2	1	2	0.159	8	1.07	5	1.95
Sam2p	Issue #33	Divide-by-zero	Seq.	2	2	3	1.219	10	8.32	7	9.82
Sqlite	Ticket #cfa2c908f2	Assertion fails	Seq.	2	1	2	0.139	8	2.07	5	3.54
Stringbuffer	Symbiosis [Machado et al. 2015b]	Abort	Con.	3	1	3	0.17	23	2.23	7	3.15
Tcpdump	CVE-2017-5205	Invalid address	Seq.	1	1	1	0.14	10	1.26	3	2.38
Transmission	Issue #1818	Assertion	Con.	1	1	1	0.786	5	3.73	3	4.2
Unrar	3LRVS [ZadYree 2011]	Stack overflow	Seq.	2	1	2	0.165	7	1.21	5	2.19
Average						2.3	0.32	11.29	2.24	5.38	3.41

5 EVALUATION

In this section, we evaluate the effectiveness and performance overhead of failure diagnosis (Section 5.1), showcase studies (Section 5.2), and evaluate the overhead on normal executions (Section 5.3) of WATCHER. In the end, we also compare it with existing work (Section 5.5).

All experiments were performed on an 8-core quiescent machine, with an Intel® Xeon® Bronze 3106 processor. The machine is installed with 16GB main memory, and 32KB L1, 1MB L2 and 11MB L3 cache separately. The underlying OS is Ubuntu 18.04.1, installed with Linux-4.15 kernel. GCC-7.3.0, with the `-O2` optimization flag, was used to compile all applications and libraries.

5.1 Effectiveness

We confirmed WATCHER's effectiveness with 24 bugs from 23 real-world programs, where they are fed with buggy inputs. Therefore, their original execution time is very short, as listed in the "Time" column of Table 1. These bugs include a range of program crashes caused by stack overflows, heap overflows, NULL pointer dereferences, divide-by-zeros, assertions, and abort failures, collected from the CWE database, as well as existing work [Kasikci et al. 2017, 2015; Machado et al. 2015b; Xu et al. 2017]. These bugs include 19 sequential bugs and 5 concurrency bugs, indicating that WATCHER is a general tool that could diagnose both concurrency and sequential bugs. Since WATCHER relies on iReplayer, it can only diagnose the bugs when the failing executions are reproducible, which is the reason why it does not include certain bugs of existing work [Cui et al. 2018; Kasikci et al. 2017; Xu et al. 2017]. WATCHER's limitation is further discussed in Section 6.

Table 1 shows the information of these bugs. It describes the type of a bug in the "Type" column, indicating whether this is a concurrency or sequential bug. A concurrency bug results from the outcome of interactions of multiple threads. For example, a wrong value is assigned by one thread, and it is read by another thread and causes a crash. It also shows the total levels of diagnosis in the "Level" column, the number of variables that are reported in the "Var." column, and the number of

total instructions under "Inst.". The total levels indicate how many assignments a wrong value is passed from its root cause to the crash site. WATCHER reports multiple variables sometimes, as shown in "Var." column, as the operand of a crash instruction may involve in multiple registers. The number of total instructions presents how many instructions WATCHER reports for all of these variables, which should be equal to the total levels for each variable. It also displays the original execution time in the "Time" column. For the technique of using "Software Breakpoint" and "Hardware Trace", the table shows the number of re-executions ("Rerun (#)") and the length in seconds of the analysis time ("Time (s)").

From the data listed in Table 1, we have the following **conclusions:** First, WATCHER only requires a short period of time (on average 2.24 seconds when using software breakpoints and 3.41 seconds when using hardware trace separately) for the diagnosis, and is typically less than 12 seconds. This diagnosis time is likely acceptable, if software vendors would provide some financial motivation for normal users that help the diagnosis. Second, WATCHER successfully identifies the final root cause of all of these bugs, where over 50% of bugs will require multiple-level of diagnosis to find the root cause. Third, since WATCHER relies on the replays to reproduce failures for its analysis, the analysis time is proportional to the execution time of the last epoch (but not the whole execution). That is, if a program crashes after running for one day, developers do not need to wait for one day to reproduce the failure, since WATCHER only diagnoses the last epoch (typically seconds or minutes). Last, WATCHER's report is very accurate, and it *only* reports unnecessary variables for two out of 24 bugs, such as Memcached and Sam2p. This indicates that programmers require very minimal manual effort to confirm the bugs. More details are further described as follows.

False Positives and False Negatives: In theory, WATCHER does not have *false negatives* for program crashes if the failure is reproducible and it is caused by an invalid memory write, since it could always find the root cause based on the definitions of Section 2.1. For *false positives*, WATCHER may report unnecessary variables, since it does not have the correct semantics of programs. As we discussed in Section 2.1, WATCHER may report both x and y's value propagation chain if the assertion $assert(x < y)$ is fired. Table 1 reports multiple value propagation chains for 2 out of 24 applications, including `Sam2p` and `Memcached`. For these bugs, programmers may require semantic knowledge to further determine the root cause.

Diagnosis Time: Overall, WATCHER requires a small amount of time to complete its diagnosis. The time shown in Table 1 is the total time for the diagnosis, including its static analysis and multiple re-executions. We also have the following observations: (1) When using software breakpoints, WATCHER requires a shorter diagnosis time, with 2.24 seconds on average, and 8.32 for the worst case. The reason is that WATCHER's re-execution is very efficient. (2) With hardware-assisted trace, the diagnosis time is longer and decoding the trace is the major cause of the slowdown. However, the average number of re-executions is significantly reduced, from 11.29 times for software breakpoints to 5.38 times. Hardware-assisted trace could reduce the analysis time, when each re-execution is very long.

5.2 Case Studies

This section illustrates how WATCHER can identify the root cause of complicated bugs, and how the bug report can help bug fixes. Figure 6 shows the bug report for the `Memcached` bug, where the source code (all in the `memcached.c` file) is shown in Figure 7. Basically, Figure 6 shows the value propagation chain from the network input to the crash point. Based on this report, programmers can easily determine the reason for the crash: the program reads an invalid length, causing `memmove` to touch an invalid address and cause a segmentation fault. WATCHER provides the value of each memory access that helps understand the reason of the failure. The report may not have line numbers,

if the symbol information is not available in the binary. But programmers can determine the line information based on the reported positions of instructions, using the same binary but with the symbol information included.

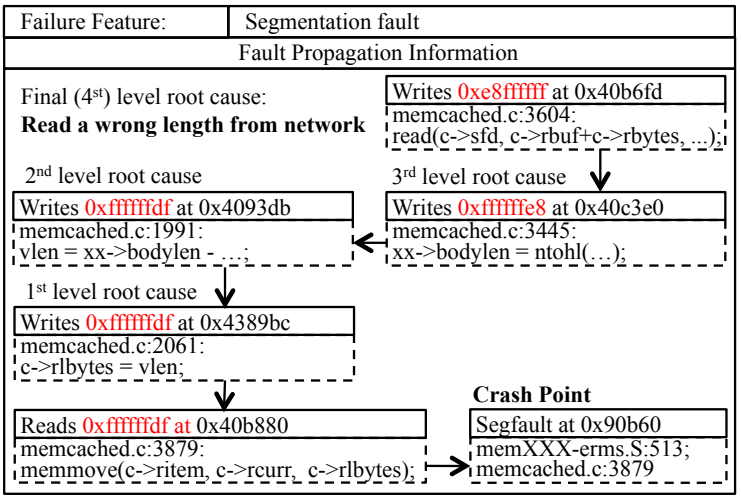

Fig. 6. Failure report for the Memcached bug.

This bug involves multiple variables and requires multi-level root cause diagnosis. `Memcached` crashes at line 3879, where the `tocopy` parameter is a negative value but interpreted as an extremely large number. However, WATCHER cannot identify whether `c->rcurr` or the size (`tocopy`) is wrong, without semantic information. Instead, it reports value propagation chains for both variables.

The diagnosis requires multi-level analysis to report the origin of the `tocopy` variable. The first level root cause is located at line 2061 of `memcached.c`, which sets `c->rlbytes` to the value of `vlen`. But this cannot explain why this value is extremely large. Therefore, WATCHER keeps track of where `vlen` is assigned from. The second level root cause diagnosis can be traced to line 1991 of `memcached.c`, and the third level can be traced to line 3445, where the value of `bodylen` is assigned. But that still did not explain why this program will crash. In fact, it requires four levels of diagnosis to determine the root cause, where the request actually reads from the network input. Overall, WATCHER reports sufficient information for bug fixes, where developers could fix all issues *without additional debugging steps or static analysis*.

5.3 Performance Overhead on Normal Execution

We evaluated the performance overhead of normal executions *when programs do not have failures*, which is very important for the deployment. Since WATCHER's overhead comes from its record-and-replay component–iReplayer, we utilized the same applications as iReplayer to confirm its recording overhead, as shown in Figure 8. Comparing to the default Linux libraries, WATCHER imposes around 1.76% performance overhead on average, which is lower than 3% reported by the iReplayer paper. We believe that the hardware is the major cause of this difference, since we are using an 8-core machine that has a lower contention than the 16-core machine of iReplayer's paper.

We also observed that WATCHER introduces less than 7% runtime overhead for most applications. Two applications (e.g., `dedup` and `raytrace`) perform even better than the default library. We further confirmed that the performance boost is due to iReplayer's custom allocator, since the allocator

```
static void process_bin_update(conn *c) {
      ......
1991: vlen=c->binary_header.request.bodylen-(nkey+c->binary_header.request.extlen);
      ......
2061: c->rlbytes = vlen;
}

static int try_read_command(conn *c) {
      ......
3444: c->binary_header.request.keylen = ntohs(req->request.keylen);
3445: c->binary_header.request.bodylen = ntohl(req->request.bodylen);
      ......
}

static ...... try_read_network(conn *c) {
      ......
3604: res = read(c->sfd, c->rbuf + c->rbytes, avail);
      ......
}

static void drive_machine(conn *c) {
      ......
3875: int tocopy = c->rbytes > c->rlbytes ? c->rlbytes : c->rbytes;
      ......
3879: memmove(c->ritem, c->rcurr, tocopy);
......
}
```

Fig. 7. Code snippet for the Memcached bug.

reduces the synchronization overhead with its per-thread heap design. However, `fluidanimate` introduces significant performance overhead. In order to understand the reason, we collected application data on the number of epochs, system calls, and synchronizations, as shown in Table 2. The number of epochs, synchronizations and system calls can significantly affect its runtime overhead. The `fluidanimate` acquires 1.1 billion locks within 30 seconds, where the recording overhead of these synchronizations is the major reason for the slowdown.

Table 2. Characteristics of applications

Applications	Epochs	Syscalls	Syncs.	Epoch(s)
blackscholes	1	5	22	43.17
bodytrack	1	13602	1947k	35.10
canneal	1	12499	195	30.42
dedup	1	806k	1005k	19.53
ferret	1	527	3725	11.91
fluidanimate	12	4420	1178005k	6.99
raytrace	1	53588	6360	63.89
streamcluster	1	3	1311k	100.32
swaptions	1	1	25	39.56
x264	2	7	207k	25.01
aget	50	249k	125k	0.12
pbzip2	1	352	2592	2.97
pfscan	1	14	39	5.21
sqlite	9	427k	1178k	1.34

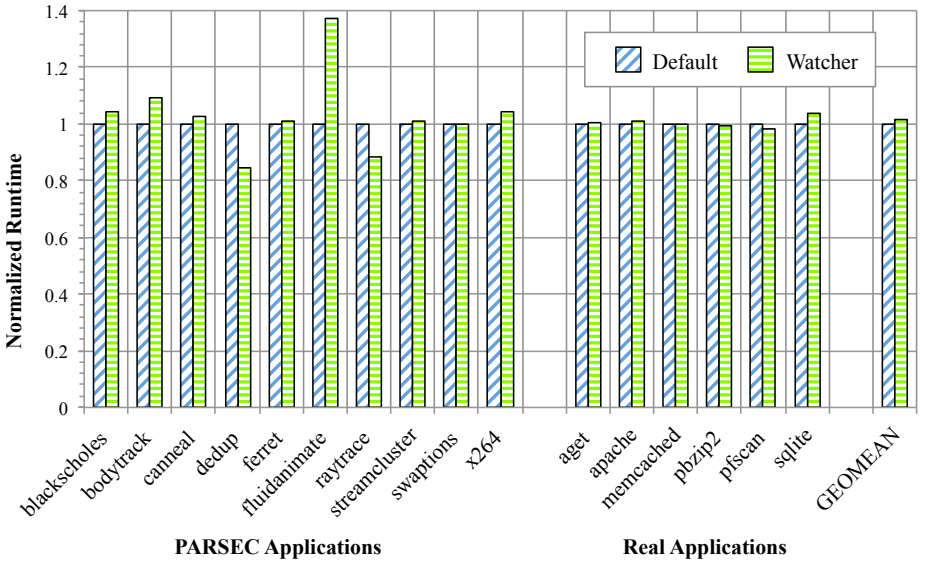

Fig. 8. Performance overhead of WATCHER's recording.

5.4 Memory Overhead

We also collected the memory usage of WATCHER, where the data is omitted due to the space limitation. For applications that will terminate after some period of time, the maximum memory consumption is obtained through the Linux `time` command. For server applications that do not terminate, a script is used to periodically collect the peak memory usage. More specifically, the resident size (`VmHWM`) in the `/proc/PID/status` file is used for the physical memory consumption. Overall, WATCHER's memory consumption is around 4.5×, mainly coming from its record-and-replay framework–iReplayer. We also noticed that small-footprint applications have a larger increase of memory consumption, since iReplayer needs some startup overhead. But we also notice that `fluidanimate` and `x264` contribute a big portion of the memory consumption. *When these two applications are excluded, the total memory overhead is only around 54%.*

We further investigated the reason for the memory consumption, which we believe is primarily coming from the recording of system call results and synchronization order. The characteristics of these applications are further shown as Table 2. Note that server applications are removed since their memory consumption mainly depends on the client's behavior. For `fluidanimate`, this application invokes 19 million lock acquisitions within a second, which is extremely large compared to the other applications. Thus, the recording of all of these synchronizations is the major source of its memory overhead. For `x264`, iReplayer's allocator cannot support memory re-utilization across different threads, while there are more than 1000 threads in the `x264` application, creating the memory blowup issue [Berger et al. 2000]. Overall, we believe that WATCHER's memory overhead is still acceptable due to the following two facts. First, with the evolution of technology, the capacity of memory is not an issue, although the speed of memory remains to be [Alted 2010]. Second, the memory overhead could be largely alleviated, if some execution records could be written to the disk or remote network memory.

Table 3. Memory overhead of WATCHER's recording

Applications	Default	WATCHER
Large Footprint (> 100MB)		
blackscholes	614	624
canneal	851	1,026
dedup	1,562	2,235
fluidanimate	200	4,938
raytrace	1,286	1,843
streamcluster	109	443
x264	264	3,013
pfscan	2,033	1,949
GEOMEAN		3×
AVERAGE		5.8×
Small Footprint (< 100MB)		
bodytrack	33	508
ferret	61	116
swaptions	7	18
aget	3	75
apache	3	84
memcached	6	28
pbzip2	50	195
sqlite	16	149
GEOMEAN		7.4×
AVERAGE		11.3×
Overall GEOMEAN		4.5×
Overall AVERAGE		8.6×

Table 2 also shows the epoch length of these applications, where the epoch length varies between 0.12 and 100.32 seconds, with an average of 27.54 seconds. Since WATCHER is built on top of iReplayer, it could only identify the root cause within the last epoch of execution. *With an epoch length of dozens of seconds,* Watcher *is expected to diagnose more failures than the state-of-the-art—REPT [Cui et al. 2018].* REPT could only reconstruct far less than 1 second of execution. Among these applications, `aget`'s epoch is the shortest. `aget` downloads files from the network, where WATCHER records the content of these files (since they are from the socket), then stops the current epoch when the buffer is full. Frequent stopping creates a large number of epochs, thus making the epoch length very short.

5.5 Comparison with Existing Work

This section compares WATCHER with two state-of-the-art on the effectiveness, diagnosis time, and potential overhead, POMP [Xu et al. 2017] and REPT [Cui et al. 2018]. Both POMP and REPT belong to offline diagnosis that requires memory core-dump and execution trace to diagnose software failures. We collected some common bugs from these papers, and the results are shown in Table 4. Note that the data is directly collected from their corresponding papers, instead of re-evaluating them on the same machine. Some real issues prevent us from evaluating them directly: POMP only supports 32-bit applications, while WATCHER can only support 64-bit applications due to the design of iReplayer's memory allocator. REPT is not publicly available.

Effectiveness: WATCHER provides the most complete coverage compared to existing work, as shown in Table 4. As self-acknowledged [Xu et al. 2017], POMP cannot diagnose all concurrency failures, as well as some sequential failures (e.g., `Aireplay-ng`), if the root cause is related to a system call that cannot be reconstructed offline. REPT also cannot diagnose this bug for the same reason [Cui et al. 2018]. Many bugs are marked with a "?" for REPT, since we do not know whether REPT could reconstruct the values of relevant registers or memory units without the execution of REPT. However, we believe that WATCHER provides better coverage than POMP and REPT due to the following reasons: (1) POMP can only diagnose sequential failures, but not concurrent bugs [Xu et al. 2017]. (2) REPT cannot diagnose the failures when their root cause is related to system calls or un-available third-party binaries. (3) According to their paper [Cui et al. 2018], REPT could reconstruct the execution of up to 78, 103 instructions (e.g., the `PHP-74194` bug), but with 9.12% instructions as unknown or incorrect register uses. In fact, modern computers typically execute billions of instructions per second, such as 3400 Million Instructions Per Second (MIPS) for our evaluation machine. That is, the 78, 103 instructions are only around 0.0003 seconds of execution, and REPT could only diagnose a failure if its root cause locates less than this distance. Since WATCHER could diagnose all failures occurred in the last epoch, around 27 seconds based on our evaluation in Section 5.3, it is **five orders of magnitude** higher than that of 78, 103 instructions of REPT.

Table 4. Comparing WATCHER with other recent work

Application	Type	Effectiveness			Analysis Time			
		POMP	REPT	WA	POMP	REPT	WA(SB)	WA(HT)
Aireplay	Seq.	✗	✗	✓	-	-	1.31s	3.02s
Gas	Seq.	✓	?	✓	40m	-	2.82s	4.15s
Gif2png	Seq.	✓	?	✓	46m	-	1.49s	2.13s
Libpng	Seq.	✓	?	✓	5m	-	0.87s	2.89s
Nasm	Seq.	✓	✓	✓	44s	18.6s	2.77s	3.99s
Openjpeg	Seq.	✓	?	✓	1s	-	2.1s	1.37s
Pbzip2	Con.	✗	✓	✓	-	8.2s	0.73s	1.06s
Unrar	Seq.	✓	?	✓	6h	-	1.21s	2.19s

Note: **?** indicates unknown results; **Seq.** and **Con.** indicate sequential or concurrency bug.

Diagnosis Time: The diagnosis time is also shown in Table 4. For WATCHER's analysis time, both software breakpoints ("WA(SB)") and hardware-tracing ("WA(HT)") are listed in the table. Overall, WATCHER's analysis time is orders of magnitude lower than POMP, and is generally smaller than REPT. For the `Unrar` bug, POMP takes 6 hours to diagnose the bug, while WATCHER only costs less than 3 seconds. For the `Nasm` bug, POMP takes 44 seconds to diagnose, and REPT spends 18.6 seconds, while WATCHER only requires 3.99 seconds to diagnose this bug. That is, Watcher *is* 4.6× *faster than REPT and more than* 10× *faster than POMP*. Based on our understanding, three reasons contribute to this difference: (1) First, WATCHER employs the dynamic analysis, instead of analyzing the instructions statically. As we discussed before, executing one instruction will take much less time than analyzing one instruction statically. (2) WATCHER performs the decoding on demand (only partial instructions), while POMP and REPT typically decode all instructions starting from the crash point. However, the decoding overhead is one major source of the performance overhead. For the `Nasm` application, REPT decodes and analyzes 67, 726 instructions (based on their paper), while WATCHER only decodes 11, 948 instructions.

Overhead for Normal Executions: For performance overhead, since both REPT and POMP record the control flow trace using the Intel Processor Trace, they impose the overhead under 2% [Kasikci et al. 2017; Xu et al. 2017]. That is, their performance overhead is comparable to WATCHER for most applications. WATCHER may impose a higher memory overhead than REPT and POMP, since WATCHER requires to record the order of synchronizations and system call results in order to ensure the identical replay. In contrast, both REPT and POMP only record the trace of control flow, and dump out the memory image when programs crash.

6 DISCUSSION

This section discusses the limitations of WATCHER. We also discuss possible side effects, potential extensions, and potential employment.

Limitations. WATCHER is an in-situ diagnosis system that can identify the root causes of program crashes with the following limitations. First, WATCHER only identifies one of the root causes (e.g., the most basic root cause), even if a failure may have multiple candidates of root causes. Second, WATCHER excels at diagnosing crashes by writing a particular value to a memory unit, but will require additional manual effort for failures caused by not writing the particular value, sharing a similar issue with existing work [Kasikci et al. 2017]. Last, WATCHER relies on a record-and-replay framework that can identically reproduce a failing execution.

Since WATCHER is built on top of iReplayer, it inherits some of iReplayer's limitations: (1) iReplayer only replays the last epoch, which implies that WATCHER may not identify the root cause if it is not located in the last epoch. But we did not observe any such case based on our evaluation of 24 real bugs. (2) iReplayer cannot support applications with self-defined synchronizations or atomic instructions, but supporting other standard synchronizations, which is the major reason why WATCHER cannot run some applications in the related work. If an application cannot be reproduced identically, then it is impossible to utilize WATCHER to diagnosis failures of these applications. (3) iReplayer's memory consumption can occasionally be quite high, due to recording all synchronizations and system calls.

Side Effects. WATCHER will not introduce side effects during its diagnosis. Because system calls are recorded, WATCHER simulates them by only returning the recorded results without invoking real system calls. If a program communicates externally via the socket, all data sent to the socket will be skipped during the re-excutions. All GUI events will be simulated without human intervention, since they also interacted with the operating system via system calls, which can be intercepted by WATCHER's record-and-replay component.

Potential Extensions. Currently, WATCHER is able to diagnose program crashes, without human intervention. But in theory, WATCHER can be extended to diagnose any failures that dynamic analysis can do (assuming only the most recent epoch is needed), not just the specific root cause and propagation chain. As far as there is a way to trigger the analysis, via explicit symptoms or user-defined criteria, then WATCHER is able to perform the dynamic analysis automatically. Such failures include program hangs, deadlocks, and different types of exceptions. Developers are encouraged to place more assertions inside programs to trigger the failure diagnosis. WATCHER can be extended with some APIs that allow users to specify conditions of triggering the diagnosis. Another possible extension is to employ WATCHER to identify issues in a separate process, while the normal process is continuing its execution.

Potential Employment. WATCHER invokes its failure diagnosis automatically upon failures, without requiring any manual effort. As a drop-in library, WATCHER can be deployed easily with the

preloading mechanism. There is no need to modify the underlying operating system, change or recompile source code, or install new hardware. Compared to offline analysis (e.g., REPT), WATCHER may record more information. However, it preserves the privacy by only reporting root causes (instead of the whole memory image), and improves the effectiveness as described in Section 5.5. WATCHER can be utilized in development phases, staging or canary deployment when new features are rolling out, where latent bugs can be diagnosed immediately (reducing the debugging time). According to the report of data-dependence slices, developers can easily fix the corresponding issues without further confirmation. WATCHER could also be shipped with the production software as an option, where software vendors could provide financial motivation to encourage normal users to help diagnose failures.

7 RELATED WORK

This section skips related work that only focuses on specific types of failures [Sanchez-Stern et al. 2018; Serebryany et al. 2012], hardware failures [Bower et al. 2005], OS failures [King et al. 2005], failure detection [Arnold et al. 2008], or failure reproduction [Bell et al. 2013; Yu et al. 2017]. Instead, it only focuses on general failure diagnosis, which can be classified into the following types.

Onsite Failure Diagnosis: Triage [Tucek et al. 2007] and Insight [Nguyen et al. 2014] also perform failure diagnosis at the failure site. However, they do not perform the failure diagnosis in the failing process as WATCHER does. Triage integrates multiple error detectors with delta analysis to enhance its effectiveness [Tucek et al. 2007]. However, Triage's effectiveness highly depends on these integrated detectors, and cannot identify the root cause of concurrency failures as self-acknowledged. Insight is claimed to be in-situ diagnosis [Nguyen et al. 2014]. However, Insight is different from WATCHER in the following aspects. First, Insight performs its diagnosis on a cloned virtual machine, instead of inside the same process. Second, it only reproduces incoming messages, which has no guarantee to reproduce the failure and might introduce false positives. For instance, some internal randomness may actually affect the reproduction. Third, Insight can only diagnose non-crash-related failures.

Core Dump Analysis: Some approaches perform postmortem analysis on core dumps by reconstructing the execution states at an arbitrary instruction [Cui et al. 2016; Glerum et al. 2009; Xu et al. 2016]. This facilitates root cause analysis, when the states of failing instructions can be recovered. However, they cannot diagnose concurrency failures, or they may stop due to incomplete states or information-destroying instructions [Cui et al. 2016]. POMP [Xu et al. 2017] and REPT [Cui et al. 2018] require control flow information to assist the diagnosis, as further discussed in Section 5.5. As offline analysis, they share multiple issues as discussed in Section 1.

Program Slicing: Program slicing finds all possible statements that are relevant to a seed statement or value of interest [Korel and Laski 1988; Musuvathi et al. 2008; Sahoo et al. 2013; Wang et al. 2014; Zhang et al. 2003]. Comparing to WATCHER, static slicing is imprecise [Harman and Hierons 2001], but it provides more relevant instructions that dynamic slicing cannot do, such as some non-executed instructions in the failing execution. In contrast, WATCHER belongs to dynamic slicing. However, different from traditional dynamic slicing, its in-situ environment allows it to perform various types of dynamic confirmation and pruning. Therefore, WATCHER is able to eliminate all irrelevant instructions with its dynamic confirmation, and reports erroneous values of each data-dependence slice that is both necessary and sufficient for fixing the bug. Overall, WATCHER provides an efficient and effective technique to perform dynamic dependence slicing, which can be extended to other dynamic analysis.

Statistical Analysis: SymbioSis [Machado et al. 2015a], Gist [Kasikci et al. 2015], and Snorlax [Kasikci et al. 2017] statistically infer root causes of concurrency failures with multiple successful

and failing executions. However, they require significant numbers of traces that users may not be willing to share, or require back-and-forth between users and programmers for failure diagnosis (e.g. Gist [Kasikci et al. 2015]). Kairux also employs the difference between successful and failing execution to diagnose root cause [Zhang et al. 2019]. Kairux can diagnose sequential failures, and could also reconstruct test cases using existing unit tests. Compared to them, WATCHER does not need successful executions, and can diagnose the root cause in the failing process.

8 CONCLUSION

This paper proposes an in-situ diagnosis that could diagnose software failures in the failing process. Comparing to existing work, WATCHER only reports the root causes to programmers, eliminating the privacy concern. WATCHER proposes a hybrid approach that combines identical replay, debugging methods, binary analysis, and happens-before analysis together to perform the failure diagnosis automatically. WATCHER further proposes software breakpoints to overcome the hardware limitation, and employ the hardware-assisted trace to diagnose failures under control-flow hijacks. Experimental results on 24 bugs demonstrate that WATCHER can diagnose a range of software failures instantly.

ACKNOWLEDGMENTS

We thank anonymous reviewers and Shan Lu, Xu Liu and Wei Wang for their helpful comments on improving this paper. This material is based upon work supported by the National Science Foundation under Award CCF-1566154, CCF-1823004, CCF-2024253, CCF-1919044, CCF-1901242, and CCF-1910300. This research is also supported, in part by ONR N000141712045, N000141410468 and N000141712947, IARPA TrojAI W911NF-19-S-0012, and Sandia National Lab under award 1701331. The work is partially supported by Mozilla Research Grant and UMass Start-up Package as well. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Francesc Alted. 2010. Why modern CPUs are starving and what can be done about it. *Computing in Science & Engineering* 12, 2 (2010), 68.

Matthew Arnold, Martin Vechev, and Eran Yahav. 2008. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. ACM, New York, NY, USA, 143–162. https://doi.org/10.1145/1449764.1449776

Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight Recording to Reproduce Field Failures. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 362–371. http://dl.acm.org/citation.cfm?id=2486788.2486836

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS IX)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. 2007. Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 405–422. https://doi.org/10.1145/1297027.1297057

Fred A. Bower, Daniel J. Sorin, and Sule Ozev. 2005. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (Barcelona, Spain) *(MICRO 38)*. IEEE Computer Society, Washington, DC, USA, 197–208. https://doi.org/10.1109/MICRO.2005.8

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/512529.512560

Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 820–831. https://doi.org/10.1145/2884781.2884844

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Freyja. 2017. How much could software errors be costing your company? https://raygun.com/blog/cost-of-software-errors/.

Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 103–116. https://doi.org/10.1145/1629575.1629586

Patrice Godefroid and Nachiappan Nagappan. 2008. Concurrency at Microsoft: An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*.

Godefroid, Patrice and Nagappan, Nachi. 2008. Concurrency at Microsoft - An Exploratory Survey. http://www.microsoft.com/en-us/research/publication/concurrency-at-microsoft-an-exploratory-survey/.

Mark Harman and Robert Hierons. 2001. An overview of program slicing. *software focus* 2, 3 (2001), 85–92.

Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 141–152. https://doi.org/10.1145/2491956.2462167

Intel. 2017. *Intel XED*. Retrieved December, 2017 from https://intelxed.github.io/

Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 241–255. https://doi.org/10.1145/1869459.1869481

Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland)

(PLDI '09). ACM, New York, NY, USA, 110–120. https://doi.org/10.1145/1542476.1542489

Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 582–598. https://doi.org/10.1145/3132747.3132767

Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412

Samuel T. King, George W. Dunlap, and Peter M. Chen. 2005. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1247360.1247361

B. Korel and J. Laski. 1988. Dynamic Program Slicing. *Inf. Process. Lett.* 29, 3 (Oct. 1988), 155–163. https://doi.org/10.1016/0020-0190(88)90054-3

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: In-situ and Identical Record-and-replay for Multithreaded Applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3192366.3192380

Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A study of interleaving coverage criteria. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. ACM, New York, NY, USA, 533–536. https://doi.org/10.1145/1287624.1287703

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*.

Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015a. Concurrency Debugging with Differential Schedule Projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 586–595. https://doi.org/10.1145/2737924.2737973

Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015b. Concurrency debugging with differential schedule projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 586–595.

Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 693–708. https://doi.org/10.1145/3037697.3037751

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

Hiep Nguyen, Daniel J. Dean, Kamal Kc, and Xiaohui Gu. 2014. Insight: In-situ Online Service Failure Path Inference in Production Computing Infrastructures. In *Proceedings of the 2014*

USENIX Conference on USENIX Annual Technical Conference (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 269–280. http://dl.acm.org/citation.cfm?id=2643634.2643663

Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. ACM, New York, NY, USA, 235–248. https://doi.org/10.1145/1095810.1095833

Quora. 2015. What is a coder's worst nightmare? https://www.quora.com/What-is-a-coders-worst-nightmare.

Swarup Kumar Sahoo, John Criswell, and Vikram Adve. 2010. An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 485–494. https://doi.org/10.1145/1806799.1806870

Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 139–152. https://doi.org/10.1145/2451116.2451131

Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2018. Finding Root Causes of Floating Point Error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/3192366.3192411

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. Address-Sanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing Production Run Failures at the User's Site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/1294261.1294275

Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. 2014. DrDebug: Deterministic Replay Based Cyclic Debugging with Dynamic Slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. ACM, New York, NY, USA, Article 98, 11 pages. https://doi.org/10.1145/2544137.2544152

Paul F Wilson. 1993. *Root cause analysis: A tool for total quality management*. ASQ Quality Press.

Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. ACM, New York, NY, USA, 529–540. https://doi.org/10.1145/2976749.2978340

Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*. 17–32.

Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 26–36. https://doi.org/10.1145/2025113.2025121

Tingting Yu, Tarannum S. Zaman, and Chao Wang. 2017. DESCRY: Reproducing System-Level Concurrency Failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 694–704. https://doi.org/10.1145/3106237.3106266

ZadYree. 2011. *Unrar 3.9.3 - Local Stack Overflow*. Retrieved October 8, 2018 from https://www.exploit-db.com/exploits/17611/

Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical Data Race Detection for Production Use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 149–162. https://doi.org/10.1145/3037697.3037708

Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise Dynamic Slicing Algorithms. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 319–329. http://dl.acm.org/citation.cfm?id=776816.776855

Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, New York, NY, USA, 131–146. https://doi.org/10.1145/3341301.3359650