Alligator in Vest: A Practical Failure-Diagnosis Framework via Arm Hardware Features

Yiming Zhang
Research Institute of Trustworthy Autonomous Systems and Department of CSE, SUSTech, China
The Hong Kong Polytechnic University, China

Yuxin Hu
Southern University of Science and Technology, China

Haonan Li
Southern University of Science and Technology, China

Wenxuan Shi
Southern University of Science and Technology, China

Zhenyu Ning*
Hunan University, China
Southern University of Science and Technology, China

Xiapu Luo*
The Hong Kong Polytechnic University, China

Fengwei Zhang
Southern University of Science and Technology, China

ABSTRACT

Failure diagnosis in practical systems is difficult, and the main obstacle is that the information a developer has access to is limited. This information is usually not enough to help developers fix or even locate the related bug. Moreover, due to the vast difference between the development and production environments, it is not trivial to reproduce failures from the production environment in the development environment. When failures are caused by non-deterministic events such as race conditions or unforeseen inputs, reproducing them is even more challenging.

In this paper, we present INVESTIGATOR, a failure diagnosis framework for practical systems running on Arm. At runtime, INVESTIGATOR leverages the hardware tracing component called Embedded Trace Macrocell (ETM) and a lightweight event capturer to collect information with low overhead. With the collected trace and analysis, INVESTIGATOR identifies the control and data flow related to the cause of a failure, which helps developers in bug fixing. We implemented a prototype of INVESTIGATOR and evaluated it with real-world bugs. The results show that INVESTIGATOR diagnoses these bugs effectively and efficiently while introducing a low performance overhead at runtime.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Software and application security**.

*Corresponding authors

KEYWORDS

ETM tracing, Failure Diagnosis, Debugging

ACM Reference Format:
Yiming Zhang, Yuxin Hu, Haonan Li, Wenxuan Shi, Zhenyu Ning, Xiapu Luo, and Fengwei Zhang . 2023. Alligator in Vest: A Practical Failure-Diagnosis Framework via Arm Hardware Features. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598106

1 INTRODUCTION

Software failures in production environments are unavoidable, and diagnosing them is highly important. It is unlikely that all failures can be revealed before deploying a release to the production environment. Even if developers spend much effort on testing, they merely decrease the probability of failure occurrence [49]. Moreover, non-deterministic events (e.g., race conditions or unforeseen inputs) make concurrency bugs or sequential bugs more likely to manifest [38, 39, 41]. Reproducing and fixing them is even more challenging [63].

To diagnose a failure that occurred in the production environment, developers usually need to analyze the program statements related to the crash and identify what caused the program to crash [71]. To provide sufficient information facilitating the procedure, we argue that a *practical* failure diagnosis system in the production environment can first record the execution flow of the target application (e.g., control and data flow) and then analyze the application states leading to failure according to the execution record. It should fulfill the following three requirements. **R1)** ***Non-invasive***: It should not be intrusive to the production environment and can work without the source code of target programs; **R2)** ***Low overhead***: The runtime overhead incurred should be low enough for the production environment; **R3)** ***Complete***: It should be able to provide a complete execution flow of a target program, which is necessary since using insufficient information may miss the root cause of failures.

Record-and-replay [14, 56, 59, 69] is a typical system to diagnose failures by reverse-execution debugging. However, one of the challenges in these systems [24, 25, 69] is that they do not fulfill R2 because the high performance overhead introduced by recording, especially for a multithreading program [56]. Besides, they do not meet R1, because they require OS modification [12, 20, 51] and custom hardware [32, 53]. Other approaches [14, 29, 36] record the execution flow by pervasive instrumentation of the source code, which requires code modification and introduces high overhead [36, 39]. Using dynamic instrumentation incurs even more overhead [21, 54].

To provide efficient production support, the state-of-the-art failure diagnosis systems [18, 19, 38, 70, 71] usually take advantage of modern hardware (e.g., Intel Processor Trace [60]) to record the control flow of a program with low overhead. However, the standalone control flow is insufficient for failure diagnosis in complex cases involving data. Therefore, the control flow is usually used with a coredump [45] captured upon a program crash to infer the data flow [18, 71]. Unfortunately, relying on the final crashed coredump to diagnose a failure related to non-deterministic events is still difficult in practice, since the memory and registers involved in the failure might be overwritten before the coredump is captured, and thus do not fulfill R3 (an example is given in §2.2). Furthermore, although failure diagnosis has been well studied on traditional x86 platforms [18, 19, 38, 70, 71], it is still an open problem on Arm architecture.

In light of these limitations, we propose Investigator [1], a practical failure diagnosis framework on Arm to fulfill the three requirements. R1: Investigator diagnoses a binary program executing on multi-thread environments by leveraging the off-the-shelf hardware features on Arm, including ETM (Embedded Trace Macrocell) [5] and PMU (Performance Monitoring Unit) [8], so that it requires no modification to any hardware components or binaries in the production environment. R2: Investigator records control flow and timing information of a target program by cooperating with the hardware-tracing component ETM, which imposes a low overhead. Meanwhile, we minimize the size of recorded data generated by ETM to simplify the tracing procedure. R3: With a novel approach that maintains the complete ETM trace, Investigator is capable of reconstructing the entire control flow. Investigator supports recovering the data flow via the collected record, and finally completes the data flow.

To implement the whole process, we faced three technical challenges. C1: The ETM's trace buffer (e.g., Embedded Trace Buffer (ETB) [5]) is limited so that it will get fully occupied very quickly. Furthermore, ETM cannot raise an interrupt when the trace buffer is full. Thus, the trace can be frequently overwritten by subsequent program execution, causing trace loss. To address C1, we design an approach to effectively and efficiently extract traces and transfer them to non-volatile storage in a timely fashion with the help of PMU. C2: The default timestamp generated by the ETM is too coarse-grained to reflect the exact thread interleavings (i.e., non-deterministic event that lead to concurrency bugs) in multi-thread environments. To handle C2, Investigator is equipped with a novel technique that leverages a built-in hardware called Countdown-Counter [5] to maximize the generation of timestamps. Thus, Investigator can use the fine-grained timestamp to determine the thread interleaving causing concurrency bugs. C3: Since ETM does not support data trace as Intel Processor Trace (PT) does (i.e., `ptwrite` instruction), it is non-trivial to accurately recover data flow with low overhead, especially considering the impacts of non-deterministic events (e.g., unforeseen inputs). To tackle C3, we reconstruct the data flow based on the control flow by forward analysis with collected information. Specifically, Investigator employs a lightweight software-based mechanism to collect additional records from non-deterministic events. We carefully classify the non-deterministic events by considering their impact on recording requirements and collect these effects with different strategies to avoid incurring high runtime overhead or sacrificing the accuracy of data flow.

We implemented a prototype of Investigator and evaluated it with the aforementioned requirements in mind. The evaluation results show that Investigator recovers both the control flow and data flow with an accuracy of more than 99%. We also used Investigator to diagnose 17 failed programs (7 code segments reconstructed from applications and 10 real-world applications). The results show that Investigator effectively identifies the root cause of the failures caused by concurrency and sequential bugs with a runtime overhead of up to 3.88% on average.

In summary, we make the following contributions:

- We present a practical failure diagnosis framework named Investigator. It leverages Arm hardware features (e.g., ETM and PMU) to efficiently reconstruct execution pertaining to failures. Investigator works with unmodified binaries on Arm platforms without hardware modification.
- Investigator achieves high performance and performs accurate execution flow recovery, which provides developers with sufficient information for root cause analysis.
- We implement a prototype of Investigator and evaluate it with real-world applications. The results show that Investigator successfully diagnoses various types of failures in these applications with up to 3.88% runtime performance overhead on average.

2 BACKGROUND AND MOTIVATION

2.1 Embedded Tracing Macrocell

Embedded Tracing Macrocell (ETM) is a trace component in Arm architecture [5] that generates instruction trace to describe all the executed instructions on a processor. Moreover, additional information such as timestamps can be captured along with the instruction trace [10]. To facilitate the usage, ETM also provides a range of options such as filters (e.g., process ID and memory region filters) and trace sinks (e.g., Embedded Trace Buffer (ETB), Embedded Trace Router (ETR), and Trace Port Interface Unit (TPIU)). Moreover, ETM only imposes 0.1% overhead in most cases and hardly impacts the execution of applications [55]. As an embedded component inside the processor, ETM is available in almost all Arm Cortex-A and Cortex-M processors [3, 5]. Recent study [55] showed that ETM is used in most popular smartphones and tablets based on Arm. Note that data trace is not supported in both the ETMv4 equipped in the Armv8-A architecture according to the Arm technical manual [5].

[1]What do you call an alligator in a vest? An investigator!

```
1  // Thread 1::
2  char big_buf[64];
3  while(1)
4    read(fd, big_buf, 64);

5  // Thread 2::
6  int total = 0;
7  int len = 0;
8  char buf[15];
9  for (short i=0; i<2; ++i)
10   len = strlen(big_buf);
11   if (len<15)
12     strcpy(buf, big_buf);
13     total += len;
14 assert(total<30)
```

Figure 1: Buffer overflow caused by data race.

2.2 A Motivating Example

Figure 1 demonstrates a typical concurrency bug related to a non-deterministic event (syscall `read`) to show limitations of prior diagnosis systems. Assume that the loop (Line 10 to Line 13) in Thread 2 is executed twice. In the first iteration, the `read` of `big_buf` (Line 4) in Thread 1 is performed after the length check (Line 10 and Line 11) in Thread 2. The following `strcpy` (Line 12) in Thread 2 may lead to a buffer overflow and overwrite variables `len` and `total`. In the second iteration, no data race is involved, but the unpredictable `total` overwritten in the first iteration might be larger than 30 after the summation (Line 13) in Thread 2. This finally fails the `assert` (Line 14) in Thread 2 and crashes the program.

Existing failure diagnosis systems that are purely based on control flow traces [17, 38] fail to diagnose this example, because they do not provide data values, thus only applying at a subset of concurrency bugs. Similarly, the state-of-art techniques in deployed systems, which reconstruct data flow from a control flow trace and a crashed coredump [18, 71], can only recover limited data values in this example, because the memory and registers pertaining to the failure are overwritten by subsequent control flow. Hence, it is impossible to recover the corresponding data to identify root cause.

To overcome the limitations of existing techniques, we design and implement a failure diagnosis framework Investigator. It recovers control and data flow without incurring high overhead during execution or sacrificing the accuracy of data flow. Investigator is different from existing commercial tools (e.g., J-trace [64], and Trace32 [7]) and has no extra hardware requirement (e.g., J-LINK [65]).

3 DESIGN

In this section, we introduce the overview (§3.1) of Investigator. Then, we discuss the encountered main challenges and the corresponding solutions (§3.2).

3.1 Overview

Investigator focuses on facilitating failure diagnosis via providing execution flow of a target program. Investigator requires no modification to the target program in the production environment, because it uses the hardware features and a lightweight software-based mechanism to collect the execution traces of the target program. To minimize the additional overhead, the control and data flow recovery and the root cause analysis are conducted on the host server rather than the Arm platform. More specifically, as shown in Figure 2, Investigator consists of two stages: *recording stage* and *analysis stage*.

(1) Recording Stage (§4). The *recording stage* contains three main modules: An ETM manager, a library hook, and a non-deterministic

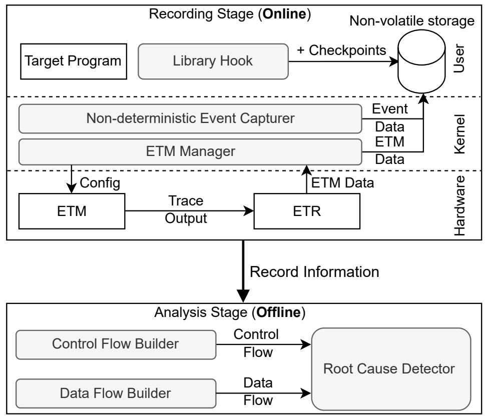


Figure 2: Overview of Investigator.

event capturer. The ETM manager controls the hardware features (e.g., ETM and ETR) to trace the executed instructions. The library hook inspects library functions invoked by the target program. Whenever the target program starts, or a new thread is created, its memory information in the corresponding status is extracted to a coredump [45] (hereinafter referred to as the *checkpoint*). The non-deterministic event capturer, another module, records the non-deterministic events related to the target program. All recorded information is saved to non-volatile storage and transferred to the *analysis stage* on an offline server when a failure occurs.

(2) Analysis Stage (§5). In *analysis stage* performed in host server, a control flow builder and a data flow builder reconstruct the control and data flow with the information collected in *recording stage*, respectively. Meanwhile, a root cause detector is responsible for identifying the root cause of failure from the reconstructed flows during *analysis stage*.

Deployment Scenario. We assume that the usage scenario of Investigator is to deploy it in the production environment for always-on trace and offline analysis. In this scenario, Investigator can gather more data about the reoccurring failure by leveraging frequent failure reoccurrences in large scale deployments, similar to existing production failure-diagnosis systems [28, 38, 39].

3.2 Design Challenges

Recall that Investigator is designed for production environment to fulfill the three practical requirements discussed in §1 (Non-invasive, Low overhead, and Complete). With this goal, Investigator faces the following design challenges.

C1) Loss of Instruction Trace. ETM relies on the capacity of on-chip buffer for trace data storage, but ETM cannot raise an interrupt when the buffer is full. Trace data will be frequently overwritten once the buffer is full, leading to the loss of instruction trace. On the one hand, it is common for a program to use standard libraries. Tracing the library together with the program will increase the size of ETM trace dramatically and consequently make the buffer overwritten even worse. On the other hand, ignoring the library might introduce inaccuracy in data flow recovery.

Solution of C1. Investigator carefully chooses the timing to extract data from the trace buffer so that no trace is overwritten. Moreover, Investigator uses a dedicated ETM configuration in conjunction with software-based library record to collect the impact of library functions instead of tracing them. This design allows us to keep sufficient information while introducing a low performance overhead. More details are provided in §4.1 and §4.2.

C2) Uncertain Order of Executed Instructions. In Arm, each CPU core has its own ETM, and each ETM is designed to trace the instructions executed in its attached core. In a multi-core system, it is not-trivial to determine the instruction order among different cores. However, the instruction order is critical while diagnosing failures caused by race conditions.

Solution of C2. We address this problem by forcing ETM to generate fine-grained timestamps. Although ETM provides a standard configuration for the timestamp generation, the generated timestamp is still too *coarse-grained* in race conditions. We propose a novel approach leveraging an event trigger to increase the timestamp generation frequency. More details are presented in §4.1 and §5.

C3) Accurately Recover Data Flow with Low Overhead. Since ETM does not support data trace as Intel PT does (i.e., `ptwrite` instruction), it is non-trivial to accurately recover data flow with low overhead, especially considering the impacts of non-deterministic events (e.g., unforeseen inputs). The non-deterministic events lead to different execution results in various executions, which makes it difficult to reproduce a failure that arisen in the production environment [62]. Existing solutions focus on functionality and introduce heavy performance overhead [30, 31, 66, 67, 69], which prevents them from being deployed in the production environment. For instance, recent works [39, 56] insert instrumentation into the target program source code or use `ptrace` [47] for the syscall effects. Nevertheless, they bring a considerable overhead (e.g., up to 24× [56]).

Solution of C3. We address this problem by leveraging a lightweight software-based mechanism to collect additional records from non-deterministic events to help data flow recovery offline. Investigator carefully classifies the non-deterministic events according to their impact on record requirements. Then we design a dedicated lightweight non-deterministic event capturer to collect these effects with different strategies to avoid incurring high runtime overhead or sacrificing the accuracy of data flow. The details about the non-deterministic event capturer are presented in §4.3, and the data recovery is described in §5, respectively.

4 RECORDING STAGE

4.1 ETM Manager

The hardware tracing component ETM supports high-performance tracing [23, 55]. ETM Manager controls ETM to trace instructions of the target program. To build a complete and accurate control flow, we keep the ETM always-on for continuous trace to provide sufficient records.

Accurate Tracing. As described in challenge **C1**, ETM cannot raise an interrupt when the trace buffer is full. Thus, the trace can be frequently overwritten by subsequent program execution. To avoid trace loss caused by overwriting, we use timely interrupts raised by hardware Performance Monitoring Unit (PMU) [8] to extract the trace from buffer and save it to disk before the buffer is full. Specifically, Investigator uses the event counter of PMU to count the `PMU_INST_RETIRED` event indicating an instruction executed by the processor to conservatively estimate the current

```
// The wrapper for the non-deterministic function calloc
void* calloc_wrapper (size_t x0, size_t x1){
    void* ret = calloc(x0, x1);
    record(__func__, ret); // save the result
    return ret;}
```

Figure 3: An example of the generated wrapper for the non-deterministic function.

size of the trace buffer while ETM is enabled. Once the number of instructions hits a threshold (close to the buffer size), PMU raises an interrupt (PMI) and notifies Investigator to drain trace from the buffer in PMI handler. To further reduce the overhead of frequent interrupt, we leverage another hardware component Embedded Trace Router (ETR [5]) to allocate a dedicated custom-sized chunk of physical memory (up to 4GB) as a high-capacity buffer to support a tracing. To our empirical knowledge, an Arm instruction can lead to at most 6 bytes of trace data. So a 4GB memory buffer using ETR can support the execution of at least 7 hundred million instructions. Also, considering that the PMI often comes with a skid (less than 10 instructions [55]), the upper bound for PMU is instruction number-10. Thus, the maximal value of the event counter register Investigator sets can be 0xffffffff-upper bound. Note that developers can adjust the size of the ETR and the PMU boundary (e.g., 256MB in our Evaluation §6).

Fine-grained Timestamps. As discussed in the **C2**, to provide accurate ETM trace in a multi-core environment, we configure ETM to generate timestamps, which contain CPU clock timing information, together with the instruction trace. The key challenge is the default generation frequency of the ETM timestamp is too *coarse-grained*, which is insufficient to identify the order of instructions in race conditions. We notice that ETM is allowed to generate timestamps in its trace whenever a special *trace unit event* [5] happens. Based on this observation, we leverage a built-in hardware *Countdown-Counter* [5] within the Arm coresight components [9] as an external source to trigger trace unit events to instruct ETM timestamp generation. Since the *Countdown-Counter* can trigger a trace unit event when the counter value reduces to 0, we configure the initial value and reload value of the *Countdown-Counter* to be 0 to make this event happen all the time to maximize the timestamp generation.

Filtered Tracing. To decrease the size of the trace, Investigator instructs the ETM trace only the target process under analysis by applying ETM's context ID filter. Moreover, Investigator let the ETM just trace the instructions executed in target's code address space via address range filter. Note that it is not trivial to reduce the amount of ETM trace while achieving accurate data flow recovery because reducing the amount of ETM trace may miss the information of non-deterministic events of syscalls and libraries. To address the side-effect of non-deterministic events, we propose a library hook for library functions (§4.2) and a lightweight software capturer for syscalls (§4.3), respectively.

4.2 Library Hook

Library hooks are used to collect the impact of library functions as described by **C1**. It is unnecessary to record all library functions. We divide library functions into two categories: deterministic functions and non-deterministic functions. A deterministic function is a pure function such as `strlen` [27]. Deterministic functions

Table 1: The classification of syscall.

Syscalls Types	Example	Feature	Record Requirement
R*eading* **S***tatus*	getpid	The *RS-Type* syscalls read information related to system status. The results of these syscalls may be transferred by the return value.	We directly record the memory or register they changed.
W*riting* **S***tatus*	epoll_create	The *WS-Type* syscalls change the status of the system, but do not directly change the memory and registers of program.	We ignore them unless they fail and return an error code.
R*eading* **C***ontent*	read	The *RC-Type* read content from an external input.	We choose to truncate the content and record only the first 256 bytes.
W*riting* **C***ontent*	write	The *WC-Type* syscalls write content to an external source.	We consider that they would not affect the execution status of the target program.

are not required to be recorded, as their effects can be inferred by function semantic information. Some functions such as `calloc` and `rand` return a non-deterministic value. We mainly record such a non-deterministic function to fix the gap that the data value cannot be inferred. These non-deterministic functions are categorized manually, INVESTIGATOR uses library hooks to collect their effects.

INVESTIGATOR hooks library functions by modifying the relocation process of dynamic linker `ld.so`. Specifically, INVESTIGATOR changes `_dl_fixup` in `dl-runtime.c` to make the dynamic linker search our wrapper function on relocation. Inside the wrapper (e.g., Figure 3), it executes the original function and saves the execution information.

INVESTIGATOR compiles a *wrapper library* for common non-deterministic functions in the standard library (e.g., `malloc`, `rand`, `calloc`, etc.). INVESTIGATOR also handles the functions that are implemented through VDSO (e.g., `clock_getres` and `gettimeofday`). With library hook, INVESTIGATOR dumps target program's memory information (i.e., checkpoints) in the key status (e.g., program start, or a new thread is created) for later data flow recovery (§5). The checkpoints are generated by `gcore` [46].

Adding wrappers for more library functions is as easy as providing the function name and prototype. INVESTIGATOR automatically generates wrappers and integrates them into the wrapper library.

4.3 Non-deterministic Event Capturer

To record the effects of non-deterministic events, a straightforward approach is to configure ETM both for user space and kernel space. However, this also introduces significant overhead (e.g., generating several gigabytes of ETM trace in seconds). Therefore, we choose to turn off the ETM for kernel space to reduce the performance overhead and design a non-deterministic event capturer to handle these effects with low overhead. We consider primary non-deterministic events from *syscalls*.

Classification of Syscall. It is non-trivial to efficiently capture syscalls as mentioned in **C3**. We classify syscalls into four types shown in Table 1 according to their effect on memory and registers. With classification, we record syscalls with different strategies to significantly reduce capture overhead. In particular, we collect parameters (from the entrance) and impact (from the exit) according to the semantics of each syscall by tracepoints [52].

R*eading* **S***tatus*: The *RS-Type* syscalls read information related to system status. The results of these syscalls may be transferred by the return value (e.g., `getpid`), a pointer (e.g., `getitimer`), or shared memory (e.g., `getrandom`). For syscalls in this category, we directly record the memory or register they changed.

W*riting* **S***tatus*: The WS-Type syscalls change the status of the system. As the WS-Type syscalls do not directly change the memory and registers of the program, we ignore them unless they fail and return an error code. For example, the syscall `epoll_create` in `epoll` API is ignored, but the change introduced by syscall `epoll_wait` is recorded since we classify it as an *RS-Type* syscall.

R*eading* **C***ontent*: The *RC-Type* syscalls read content from an external input, and the handling of *RC-Type* syscalls is similar to that of *RS-Type* syscalls. However, since the content is usually much larger than the status read in the *RS-Type* syscalls, we choose to truncate the content and record only the first 256 bytes due to the performance requirement. Recording the entire content may become impractical in extreme cases (e.g., data center with tremendous uploads) as it results in a huge size of record file. We consider a 256-byte input is already a considerable value. To the best of our knowledge, existing Linux system tool sysdig [66] only keeps the first 80 bytes of input, and there are few cases where the input size exceeds 256 bytes [66]. Although the cases are rare, we still evaluate the impact of the truncation in §6.5.

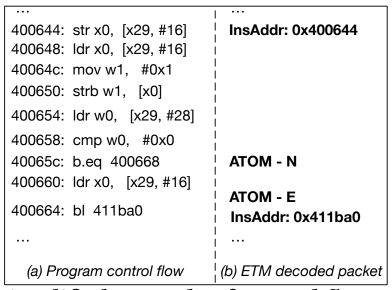
W*riting* **C***ontent*: The *WC-Type* syscalls write content to an external source. We consider that they would not affect the execution status of the target program. For example, the syscall `write` is ignored, and the written content is recorded if the *RC-Type* syscall `read` is used to read from the source again.

In general, INVESTIGATOR provides a basic record of all syscalls by default, including the syscall number, thread ID, and return result. In addition, we manually categorize over 60 system calls covering all cases that emerged from our experiment, and INVESTIGATOR provides additional records based on classification. For example, for the syscall `getrandom (void *buf, ...)`, INVESTIGATOR additionally records the data inside `buf`. Each classification is based on the above consideration with the four types of syscall in Table 1, so new syscalls also can be handled.

5 ANALYSIS STAGE

After receiving information from the *recording stage*, INVESTIGATOR processes control and data flow reconstruction and root cause analysis on another host server rather than the Arm platform.

Control Flow Builder. Control flow builder utilizes the ETM trace result and the binary of the program to reconstruct control flow (i.e., each instruction that the program executes). The ETM trace contains addresses, timestamps, and context ID. Note that ETM only records the addresses for branch and condition instructions. Therefore, INVESTIGATOR combines the ETM trace with the program binary to recover the control flow. Specifically, INVESTIGATOR builds the entire control flow by combining the `ATOM` packet as described in Figure 4 with the instructions between every two consecutive addresses of the ETM trace from the program binary. Figure 4 (b) shows the decoded ETM packet information, which records the branch address (i.e., `InsAddr`) when the program is executed. There is also an `ATOM` packet, which indicates that the program encounters a branch instruction when it executes. The `ATOM-N` indicates that the program did not jump to the branch, while the `ATOM-E` indicates



```
...                                  ...
400644: str x0,  [x29, #16]          InsAddr: 0x400644
400648: ldr x0,  [x29, #16]
40064c: mov w1,   #0x1
400650: strb w1,  [x0]
400654: ldr w0,   [x29, #28]
400658: cmp w0,   #0x0
40065c: b.eq  400668                 ATOM - N
400660: ldr x0,   [x29, #16]
                                     ATOM - E
400664: bl  411ba0                   InsAddr: 0x411ba0
...                                  ...
```

(a) Program control flow
(b) ETM decoded packet

Figure 4: A simplified example of control flow construction

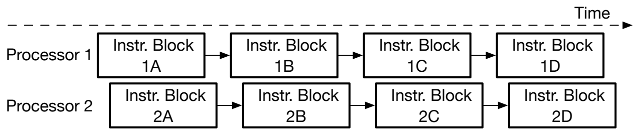


Figure 5: Instruction blocks in a multithreading program.

that the program entered the branch. In this example, the program encounters a conditional comparison at address `40065C` as shown in Figure 4 (a). With the `ATOM-N`, we learn that the program did not enter the address `400668`, which also hints that the content of register `w0` of the previous instruction `400658` is not `0x0`. Similarly, combining with the `ATOM-E` at instruction `400664`, INVESTIGATOR understands that the program will go to branch address `411ba0` to execute subsequent instructions. This subsequent address `411ba0` is also recorded by the ETM as the `InsAddr` packet.

For parallel programs, we utilize context ID to distinguish the instructions of different threads, and employ timestamps to determine the order of instructions with different context IDs. For two instruction blocks with the same context ID, the order of instructions is restored accurately in the constructed control flow. For two instruction blocks with different context ID, we use timestamps to determine the order of blocks. If one block ends before another block starts, the order of instructions can be determined. If two blocks have overlapped timestamps, the order of reconstructed instructions may be inconsistent with actual execution. However, according to the previous research [38] and our evaluation, in a multithreading program, we find that the instructions related to data race are not executed closely. Our precision of fine-grained timestamp can determine the order in which these blocks are accessed (**C2**) and ensure these instructions related to data race are located in a small number of blocks with no overlaps. For example, as shown in Figure 5, the instructions leading to data race between two threads are usually located at 1*B* and 2*C*, or 1*B* and 2*D*, thus we determine the order of data race. Details about the accuracy of timestamps are provided at §6.2 and the experiments in §6.3 show that the accuracy of timestamps in INVESTIGATOR is sufficient for diagnosing failures.

Data Flow Builder. Data flow provides the states of memory and registers after each instruction, which is essential for developers to determine the root cause of failures. Previous works [18, 71] utilize the control flow and coredump captured at crash to infer the data flow with forward-and-backward analysis. However, the backward analysis introduces uncertain factors to data flow because some instructions are irreversible. For example, if the instruction `eor x1, x1, x1` clears register `x1` (e.g., Line 5 of Figure 6), we cannot infer the original value stored in `x1` with the backward analysis. Thus, the necessary data flow for debugging the failure is possibly missing. For example, as aforementioned in Figure 1, it is impossible to recover the data affected by the first invocation of syscall `read` based on the coredump, since it is already overwritten by that of the second invocation.

```
01 // checkpoint.   --->  x29 = 0x7fe3665450   [0x7fe3665488] = 1
[0x7fe3665490] = 2
02     ldr x0, [x29,#56] --->  x0 = 1
03     ldr x1, [x29,#64] --->  x1 = 2
04     add x0, x0, x1    --->  x0 = 3, x1 = 2
05     eor x1, x1, x1    --->  x1 = 0
```

Figure 6: A simplified example of data flow construction

To accurately recover data flow (**C3**), INVESTIGATOR reconstructs the data flow by the forward analysis simulating the execution of each instruction in the restored control flow based on an initial program state (i.e., checkpoint containing the register values and the in-memory data of the target program). For a simplified example shown in Figure 6, it is a restored control flow and data flow. To recover data, INVESTIGATOR first retrieves the register values and the memory data from the checkpoint for the first two initial instructions. Then INVESTIGATOR infers the state of memory and register after the execution of each instruction via semantic information (e.g., the common instruction `add`, `eor`, etc.). In this example, the `eor` instruction erases the data of register `x1`, but it does not affect what `x1` was before. Moreover, for non-deterministic functions and events that cannot be inferred, INVESTIGATOR recovers the state of memory and registers by parsing recorded information (e.g., library hook and syscalls).

Root Cause Detector. We define the *root cause of a failure* as a group of instructions and input that satisfy the following requirements: i) They are involved in the failed execution and correlated with the occurrence of the failure; ii) they contain the necessary instructions where changing these instructions results in a correct execution. We argue that identifying such a small set of instructions and input related to failure produces reasonable efforts for facilitating failure diagnosis procedure. Therefore, INVESTIGATOR uses Root Cause Detector to narrows down the cause of a failure from the reconstructed control-data flow. Note that, we implement the detector by adapting the existing work [38]. The detector first performs an inclusion-based points-to analysis [2] for the reconstructed control flow to obtain *points-to sets*, which builds a transitive relationship between memory addresses.

INVESTIGATOR identifies the memory address and variables involved in crashing or hanging instructions as target nodes. A sequential failure is considered if the failing address are only in the same thread. For sequential failure, INVESTIGATOR filters out a small set of instructions related to the target nodes and their data from the reconstructed execution flow via points-to sets. Subsequently, INVESTIGATOR identifies the corrupted data values and corresponding control flow (e.g., input source, instruction block causing the data to be corrupted, crashed position) from the small set of instructions as the result.

If INVESTIGATOR finds that the write and read of the failing address have arisen in different threads, this failure is considered a concurrency failure. For concurrency failure, INVESTIGATOR first uses the same method to filter out a small set of instructions from execution flow across different threads. Since these instructions have recorded the actual memory access order of data race when a bug occurs, INVESTIGATOR follows the approach [38] that generates patterns as concurrency failure prediction. Specifically, INVESTIGATOR looks for the common atomicity violation patterns (e.g., RWR (Read, Write, Read), WWR, RWW, WRW) and order violation patterns (e.g., WW, WR, RW) in these instructions. Once concurrency failure patterns are gathered, INVESTIGATOR verifies the effects of each pattern following the statistical approach [36, 38, 42, 43]. In particular, INVESTIGATOR identifies the presence of a pattern in the control flow of normal execution without failure, and eliminates these patterns. It is reasonable since INVESTIGATOR gathers more traces from successful or failure executions in large-scale deployment scenario (§3.1), which is similar to existing production failure-diagnosis systems [28, 38, 39]. For the rest patterns that may successfully cause concurrency bug, INVESTIGATOR retains them as the alternative statement orders of root cause. Then INVESTIGATOR identifies the pattern that appears in the failure execution with fewer instructions as the result.

6 EVALUATION

We implement the prototype of INVESTIGATOR with C and Python based on: ETM driver on Linux, that provides interfaces for accessing the functionalities of ETM; ptm2human [33], a tool for decode ETM trace; INVESTIGATOR is deployed on an Armv8 Juno r2 board [6] equipped with 6 cores (2 Cortex-A72 cores and 4 Cortex-A53 cores) and 8GB RAM, and a 256GB SSD is attached to the board. The firmware and OS running on the board is Linaro deliverables Linux 5.3 [44], and we allocate 256 MB ETR circular buffer for ETM tracing. This is the default settings for our experiments, and the ETR buffer size can be adjusted as required.

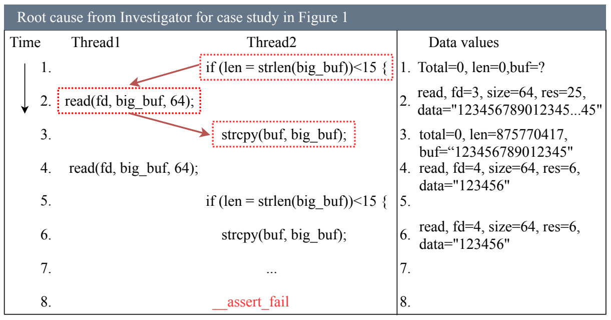
In this section, we first use a case study (§6.1) given in Figure 1 to show the complete workflow of INVESTIGATOR. Next, we focus on evaluating INVESTIGATOR with the practical requirements discussed in §1 (Non-invasive, Low overhead, and Complete). Moreover, the impact of truncation discussed in §4.3 is also evaluated. Specifically, we aim to answer the following research questions:

RQ 1: Can INVESTIGATOR accurately recover control and data flow? (§6.2)
RQ 2: Is the root cause diagnosing in INVESTIGATOR effective? (§6.3)
RQ 3: How efficient is INVESTIGATOR for production environment? (§6.4)
RQ 4: What is the impact of truncation? (§6.5)

6.1 Case Study

We use the program shown in Figure 1 to illustrate how INVESTIGATOR diagnoses its bug. When the program failure occurs, INVESTIGATOR recovers the complete control and data flow according to the collected information from the recording stage. Then, INVESTIGATOR performs root cause analysis based on reconstructed control and data flow.

By identifying crashed memory, INVESTIGATOR first determines that the memory associated with failure is related to variable `len`, and the source of `len` is related to the shared variable `big_buf`. Next, INVESTIGATOR performs alias analysis to obtain instructions related to `big_buf` from the reconstructed control and data flow.



Root cause from Investigator for case study in Figure 1

Time	Thread1	Thread2	Data values
1.		if (len = strlen(big_buf))<15 {	1. Total=0, len=0,buf=?
2.	read(fd, big_buf, 64);		2. read, fd=3, size=64, res=25, data="123456789012345...45"
3.		strcpy(buf, big_buf);	3. total=0, len=875770417, buf="123456789012345"
4.	read(fd, big_buf, 64);		4. read, fd=4, size=64, res=6, data="123456"
5.		if (len = strlen(big_buf))<15 {	5.
6.		strcpy(buf, big_buf);	6. read, fd=4, size=64, res=6, data="123456"
7.		...	7.
8.		__assert_fail	8.

Figure 7: The root cause from INVESTIGATOR for Figure 1.

With the identified instructions containing `big_buf`, INVESTIGATOR suspects that the potential access patterns of this failure are `RWR` (Read, Write, Read) and `WRR`. Finally, INVESTIGATOR verifies whether the patterns appear in the reoccurring failure, and the pattern `RWR` is chosen as output.

The root cause identified by INVESTIGATOR for this failure is illustrated in Figure 7. The arrows between the two threads in dotted rectangles indicate the pattern of data race that causes this failure (`RWR`). We can see that Thread 2 checks the length of `big_buf` (Line 1) before the first `read` (Line 2) in Thread 1. Since the data of `read` is captured, developers can identify that `big_buf` contains a string with length 25. Therefore, the following `strcpy` (Line 3) introduces a buffer overflow, and `len` is unexpectedly changed to `875770417`, which finally leads to the failure of `assert` (Line 8).

Note that the program does not crash immediately after the buffer overflow. In contrast, it executes normally for the second iteration (Line 4 to 6). The second iteration overwrites the values (`buf` and `len`) involved in the buffer overflow. At this point, developers may be confused without the help of INVESTIGATOR, for these values should not cause the failure.

6.2 Accuracy

To evaluate the accuracy of INVESTIGATOR, we employ 8 buggy programs widely used to failure diagnosis systems [18, 39, 73, 74] and measure the accuracy from three aspects: The accuracy of ETM timestamps, the accuracy of control flow recovery, and the accuracy of data flow recovery.

Recall that the generation rate of ETM timestamps might affect the accuracy of control flow recovery (§5). To evaluate whether the ETM timestamps generated by INVESTIGATOR are sufficient for control flow recovery, we test it with a case with extremely heavy data race. Specifically, we make two threads write to a shared variable concurrently for 10, 000 times without any other operations. The result shows that INVESTIGATOR is able to identify the order of 99.34% instruction blocks, and the blocks without timestamp only take a percentage of 0.66%. Thus, we believe that timestamps in INVESTIGATOR are sufficient for most programs in practice.

To further evaluate the accuracy of control flow recovery, we compare the reconstructed control flow with the original binaries and their source code. Note that INVESTIGATOR focuses on binary programs, the source code is used as the ground truth. The result shows that executed instructions from the reconstructed control flow in all tested programs are consistent with the logic of the

Table 2: The data recovery rate of Investigator. We obtain the data recovery rate under the configuration of 256 bytes truncation. #Insts= #instructions; #GT=the number of instrumented coredumps; Data Rec=data recovery rate.

Program-BugID	# Insts	# GT	Data Rec.
shared_counter-NA	172	3	100%
log_proc_sweep-NA	264	9	100%
bank_account-NA	387	10	100%
circular_list-NA	2,108	28	100%
mysql-169	3,867	4	100%
pbzip2-N/A	8,053	11	100%
curl-2017-1000101	9,161	7	97.65%
curl-965	14,412	7	99.30%

source code, and match the original assembly code in the binary. Therefore, we consider the reconstructed control flow is accurate.

For the evaluation of data flow recovery, we need to obtain the ground truth. We derive the ground truth sampling by manually instrumenting a large number of coredumps in the source code. Note that the instrumentation is only used to obtain the ground truth, and Investigator does not require the instrumentation at all. Next, the instrumented program is executed with Investigator, and all the records except for the instrumented coredumps are fed to the Control flow Builder and Data Flow Builder to reconstruct the control flow and data flow. Finally, we compare the reconstructed data flow with the captured coredumps.

As shown in Table 2, Investigator fully recovers the data flow of 6 tested programs, and the data recovery rate of the rest 2 programs is more than 97%. The incomplete data recovery is mainly caused by truncation of non-deterministic event capturer (§4.3). The detailed evaluation of the impact of the truncation is presented in §6.5.

We also compare the accuracy of data flow recovery with REPT [18], a state-of-the-art system targeted at x86. Since the source code of REPT is not public, and we can not reimplement REPT due to different CPU architecture. We did an alternative best-effort qualitative comparison focused on one representative program pbzip2 with an average data recovery rate in REPT. Compared with the 95.33% recovery rate in REPT, Investigator completely restores the data flow of pbzip2. We consider the reason is that REPT does not handle syscall and marks all volatile registers as unknown upon a system call.

6.3 Effectiveness

We collect tested cases from bugbases [11, 34, 35], which were used in previous works [18, 38, 39, 73, 74]. However, due to the Investigator being based on the Arm 64-bit architecture, not all test cases can be compiled and reproduce failures successfully. Therefore, we use 17 representative C/C++ buggy cases to evaluate the effectiveness of Investigator in diagnosing the root cause of failures.

We believe that these evaluated programs have covered wide bug types, which are divided into two groups, i.e., Group E and Group R. As listed in Table 3, Group E contains 7 bugs reconstructed from homebrewed applications, and Group R includes 10 bugs in real-world applications. There are 13 concurrency bugs, of which 7 are single-variable atomicity violation (SAV), 3 are multi-variables atomicity violation (MAV), 2 are deadlock (DL), and 1 is order violation (OV). There are also 4 non-concurrency bugs. These bugs are collected from a diverse set of popular real-world systems (e.g., Mysql, Apache, Curl, Pbzip, and Aget) and wide symptoms (e.g., NULL pointer dereference, use-after-free, and race).

We execute these programs using PoCs separately in our system until the failure occurs and use Investigator to analyze the failure. With the identified root cause and reconstructed control and data flow, we perform extensive experiments in terms of facilitation, diagnosis accuracy, and order accuracy to evaluate how the generated output helps failure diagnosis.

Facilitation. First, we measure the quantity of generated root cause (e.g., #instructions and #variables) to show how Investigator can facilitate developer understand failure. As shown in Table 3, the outputs of Investigator contain only a small number of instructions and variables from a large-scale control and data flow. With Investigator, the number of instructions and variables that need to be understand is significantly reduced. For example, the reconstructed control and data flow on the execution of bug in `pbzip2` include 8,053 instructions and 89 variables, and Investigator reduces the quantity to 6 instructions and 1 variable for developer to learn the failure.

Diagnosis Accuracy. Next, we evaluate the diagnosis accuracy of Investigator via analyzing the related bug-fixing patches and our diagnosis results, similar to previous failure-diagnosis works [38, 39]. By manually analyzing the patches indicating the location that the developers fix the bug and our diagnosis results, we can see whether Investigator can accurately diagnose the root cause. Specifically, we check whether the modified instructions and related variables in the patches are included in the output of Investigator. The result indicates that the output of Investigator contains the fixed instructions and variables for all 17 bugs. For example, the issues with data races identified by developers have the same patterns as what Investigator diagnoses as the root causes of concurrency bugs. We also find that the developer fixes the sequential bugs by limiting illegal input parameters. The illegal input data, variable corrupted position, and crashed position are provided in the Investigator diagnosis results.

Order Accuracy. Moreover, we use a measurement (normalized Kendall tau distance [39] τ) from previous works [38, 39] to evaluate order accuracy. The measurement can analyze the similarity between the ordered instruction list of diagnosis results provided by Investigator and the ordered ground truth list. Specifically, we use the normalized Kendall tau distance to count the order of pairwise disagreements between two instruction lists. The larger distance means the order between the two instruction lists is less similar. Therefore, we define the ordered instruction list in the root cause generated by Investigator as Δ_I and that in the ground truth as Δ_G, then the order accuracy is defined as a percentage $OA = 100 \times (1 - \frac{\tau(\Delta_G, \Delta_I)}{\#\text{ of pairs in } \Delta_G \cap \Delta_I})$. We measure the order accuracy for all the 17 bugs diagnosed in our evaluation. The OA is 100% for each one, which allows us to conclude that Investigator can diagnose the failures with high accuracy. It also confirms that the timestamps generated by ETM are precise enough to determine the accurate control flow that causes the failures.

Comparison of effectiveness. We further compare Investigator with REPT [18] in the types of bugs supported for diagnosis. Investigator recovers the data pertaining to failures caused by non-deterministic input (i.e., `curl-965`, `curl-2017-1000101`,

Table 3: Bugs diagnosed by Investigator.

		Program-BugID	Bug type	Symptom	CDF		Match	Root Cause	
					# I	# V		# I	# V
E	1	shared_counter-N/A	SAV	assertion failure	225	8	Yes	4	1
	2	log_proc_sweep-N/A	SAV	segmentation fault	234	19	Yes	6	1
	3	bank_account-N/A	SAV	race condition fault	366	14	Yes	5	1
	4	string_buffer-N/A	SAV	assertion failure	328	39	Yes	6	1
	5	circular_list-N/A	MAV	race condition fault	2,108	117	Yes	10	2
	6	mysql-169	MAV	assertion failure	3,867	9	Yes	12	2
	7	mutex_lock-N/A	DL	deadlock	64	8	Yes	4	2
R	8	SQLite-1672	DL	deadlock	7,139	84	Yes	12	2
	9	pbzip2-N/A	OV	use-after-free	8,032	89	Yes	4	1
	10	aget-N/A	MAV	assertion failure	7,350	76	Yes	18	2
	11	memcached-127	SAV	race condition fault	10,171	69	Yes	21	1
	12	mysql-3596	SAV	segmentation fault	32,839	97	Yes	10	1
	13	apache-21287	SAV	double free	331,639	268	Yes	22	1
	14	curl-965	SEQ	unhandled input pattern	11,412	74	Yes	20	1
	15	curl-2017-1000101	SEQ	out of bounds read	9,161	57	Yes	18	1
	16	cppcheck-2782	SEQ	unhandled input pattern	232,489	83	Yes	24	1
	17	cppcheck-3238	SEQ	null pointer dereference	280,113	94	Yes	27	1

N/A=bugid is not available; CDF=reconstructed control and data flow; #I=#instructons; #V=#variables; Match=the root cause from Investigator correlated to the bug patches (ground truth).

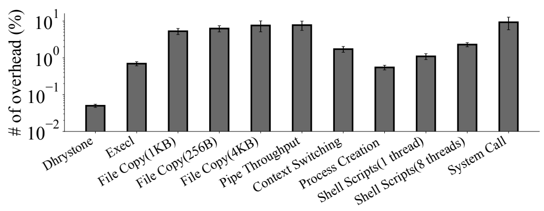


Figure 8: Performance Evaluation with UnixBench.

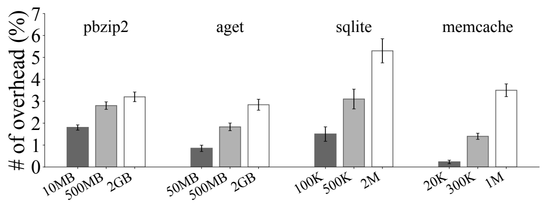


Figure 9: Performance Evaluation with Real-world Programs.

cppcheck-148, and cppcheck-3238) and successfully locates these bugs. In contrast, without handling syscalls for the buggy programs, REPT is hard to recover corresponding data flow affected by the input, as it does not handle and mark the data as unknown. For example, the program curl-2017-1000101 crashes when the system call overwrites the memory buffer. It is critical to find the input to understand the root cause, but it is impossible to know using a crashed coredump without providing system call data flow since the heap (including the value of input) was corrupted by the overflow. As such, REPT would fail to find the root cause from this input data. Therefore, we believe that Investigator provides better accuracy than REPT to the above types of bugs.

6.4 Efficiency

To evaluate the efficiency of Investigator, we run UnixBench 5.1.2 [40] to measure the performance impact. Moreover, to evaluate the overhead of Investigator in practice, we use four real-world programs with various sizes of payloads as benchmarks to measure the performance overhead of normal executions when the programs do not encounter failures. Note that all tests are evaluated when Investigator's entire record modules are enabled, including ETM Manager and non-deterministic event capturer with library hooks.

UnixBench. We run UnixBench on Linux and show the performance results in Figure 8. The performance overhead is 3.88% on average when Investigator is enabled, and the highest performance overhead is 9.3% in System Call. Specifically, File Copy, Pipe Throughput, and System Call introduce a related high overhead. We consider the reason is that these benchmarks involve intensive syscall and I/O operations, which incur larger overhead than others.

Real-world Programs. To measure the performance overhead in practice, we use four real-world programs including Pbzip2, Aget, SQLite, and Memcached. Each program is configured to be executed in three different sizes of input. Specifically, Pbzip2 is used to compress files of 10 MB, 500 MB, and 2 GB size. Aget is tested to download files of 50 MB, 500 MB, and 2 GB size in the same network. SQLite is evaluated by sqlite-bench [61] to write 100, 000, 500, 000, and 2, 000, 000 values in sequential key order in sync mode. A benchmark tool Twemperf [68] was used to test Memcached, which creates 20, 000, 300, 000, and 1, 000, 000 connections to a Memcached server running on localhost.

The result of the experiment is shown in Figure 9. Overall, the average performance overhead of all tests is 2.3%. The highest overhead is 5.3% for SQLite. The performance overhead has an improvement when test stress increases in all four programs. We believe it is caused by non-deterministic event capturer due to I/O operations and syscalls.

Comparison of efficiency. We compare the performance with REPT, the state-of-the-art diagnosis tool that utilizes Intel PT and runs continuously in production environments on x86 architecture. The performance overhead of Unixbench programs from previous REPT public paper [26]. For Intel PT tracing only, REPT's average performance overhead is 3.06%, and its highest performance overhead is 9.68% with a context switch logging enabled. In comparison, Investigator's average performance overhead is 3.88%, with the highest overhead being 9.3%. Based on these results, we believe that Investigator's performance overhead is comparable to the diagnosis system designed for production environments.

6.5 Impact of the Truncation

As discussed in §4.3, non-deterministic event capturer in Investigator truncates the content when recording *RC-Type* syscalls. To learn

Table 4: Impact of the Truncation

Program-BugID	Non-Trun			32 bytes			64 bytes			128 bytes			256 bytes		
	File (KB).	Root.	Over.	File (KB).	Root.	Over.	File (KB).	Root.	Over.	File (KB).	Root.	Over.	File (KB).	Root.	Over.
cppcheck-2782	12.57	Y	3.05%	12.39	N	1.72%	12.41	Y	1.84%	12.47	Y	2.14%	12.56	Y	2.92%
cppcheck-3238	15.93	Y		15.7	N		15.73	Y		15.79	Y		15.92	Y	
curl-965	50.025	Y	2.29%	49.797	N	0.72%	49.831	Y	0.96%	49.903	Y	1.36%	50.023	Y	2.24%
curl-2017-1000101	42.799	Y		42.576	N		42.604	Y		42.661	Y		42.795	Y	

File=The file size of non-deterministic event capturer; Root.=Root cause is identified or not; Over.=Performance overhead.

the impact of the truncation, we perform an extended experiment with the real-world programs `curl` and `cppcheck`. We choose these two programs since the truncation of other programs in Table 3 does not generate perceivable impacts (their causes of the failures are not related to the content of syscalls). In the experiment, we configure Investigator to truncate the content of syscalls (larger than 256 bytes) to different sizes including 32 bytes, 64 bytes, 128 bytes, and 256 bytes. Moreover, the case without any truncation is used as the baseline. For each truncation size, we measure the performance overhead and the record size of non-deterministic event capturer. The impact on the root cause analysis is presented in Table 4.

Impact on Storage. As shown in the `File` column of Table 4, the size of truncation does not affect the size of record files significantly. The reason is that the content of syscall is relatively small. However, the content would be large in specific scenarios (e.g., reading large files), and the solution without truncation would introduce significantly larger record files.

Impact on Efficiency. The `Over` column in Table 4 illustrates the performance of Investigator with various truncation sizes. Similar as the impact to the storage, the larger truncation size introduces a higher performance overhead. Note that the overheads of `Non-Trun` and `256 bytes` are similar since the length of the content is slightly larger than 256 bytes.

Impact on Effectiveness. The `Root` column indicates whether the root cause is successfully identified. Note that the truncation does not affect the accuracy of alias analysis for the control flow used in the root cause detector. Therefore, Investigator can still identify a small number of instructions related to a failure whenever truncation happened. However, Investigator might fail for bugs related to the content of *RC-Type* syscalls if the truncated data cannot provide sufficient information (e.g., truncate 32 bytes). In conclusion, we consider suitable truncation settings (e.g., 256 bytes) can both preserve sufficient information and avoid unaffordable storage or unacceptable performance slowdown.

7 DISCUSSION

Investigator uses non-deterministic event capturer to record the non-deterministic factors. The truncation of our capture may lead to reproduce the actual bug to fail. We can adjust the truncation but also incur more overhead, i.e., trade-off between truncation size and performance overhead (§6.5).

Investigator's deployment assumption is an Arm server. Nevertheless, Investigator is compatible with both the Cortex-A or Cortex-M architecture since ETM is available on almost all processors in both architectures. Therefore, to extend Investigator to other different OSes or bare-metal applications, we need to modify the non-deterministic event capturer to support other systems. However, adapting to mobile devices requires more effort, for example, only logging recent executions. This may weaken Investigator and is part of our future work.

Although Intel PT support data trace, users may instrument programs with the `ptwrite` instructions, which records data values into the Intel PT trace. The instrument causes a violation of non-invasive. In contrast, Investigator is non-invasive for data recovery. Investigator's software components can be complemented for the Intel processor, combining PT's instruction trace to help control-data flow construction and diagnosis.

8 RELATED WORK

Fault Localization Techniques. For closely-related fault localization techniques, Gist [39] and Snorlax [38] use hardware tracing on x86 architecture to statistically infer the root causes of concurrency failures with multiple successful and failing traces. However, our approach differs from the techniques in several ways. First, our approach is able to handle complex and non-deterministic failures, such as concurrency bugs and those caused by non-deterministic input like system calls. As self-acknowledged [38], Snorlax struggles with these types of failures, as it only provides detection of atomicity violation. In contrast, Investigator narrows down the cause of a failure from the reconstructed control and data flow, and is able to perform diagnosis to a failure caused by input. Second, our approach is designed to be efficient, using a combination of hardware and software tracing to minimize overhead. In contrast, Gist requires gathering data flow with instrumentation from multiple runs to find a failure sketch. Finally, our approach does not require access to source code since it solely works on the binary level. In contrast, Gist repeatedly modifies the source code to instrument programs in production to gather information for root cause diagnosis, which may not be possible or desirable in practice. However, while there are many fault localization techniques [15, 77] based on trace analysis, we perceive these techniques as complementary to Investigator's root cause detector.

Hardware-assisted Analysis Techniques. There has been an increasing number of approaches [18, 23, 26, 38, 39, 71, 76] exploiting hardware-assisted features on x86 or Arm. POMP [71] and REPT [18] can reconstruct data flow from a PT trace and a crashed coredump, but has no semantic-aware handling of syscalls, succumbing to recover a limited data. HART [23] focuses on trace for kernel module, but it weakens the multi-core support due to hardware limitations and incurs heavy runtime overhead during tracing. NCScope [78] leverages debug tool DSTEAM[4] with ETM and eBPF to collect data to identify behaviors of Android apps. In comparison, Investigator aims at failure diagnosis in the production environment without extra debug tool requirement.

Record and Replay. There is already a number of studies focusing on record and replay to help failure diagnosis [1, 13, 14, 22, 48, 56–59, 69]. Although useful during development, existing systems are not commonly used in production because the systems usually incur much overhead since they need to perform full record, or require significant number of failure running logs to replay concurrency bugs [1, 50, 57, 69].

Symbolic Execution. To reproduce a particular failure, existing approaches [16, 37, 75] relies on symbolic execution with crashed coredump to reproduce bugs by determining program input states that lead to failures. ARCUS [72] leverages PT to help construct symbolic program states with low overhead, and pinpoint a concise root cause. These approaches are orthogonal to ours. Different from these systems, Investigator focuses on reconstructing the actual control and data flow without modifying or instrumenting to the target programs.

9 CONCLUSION

We propose a failure diagnosis framework Investigator on Arm to satisfy practical requirements. It leverages hardware features and software-based mechanisms to record program execution and then build control and data flow for failure diagnosis. The comprehensive evaluation shows that Investigator accurately recovers control and data flow, and effectively identifies the root cause for various types of bugs with a low runtime overhead.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No. 62002151 and No. 62102175, Shenzhen Science and Technology Program under Grant No. SGDX20201103095408029 and No. ZDSYS20210623092007023, PolyU Grant (ZVG0) and Hong Kong RGC Project (No. PolyU15222320).

REFERENCES

[1] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*.

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Citeseer.

[3] Arm LTD. 2017. ARM CoreSight ETM-M33 Technical Reference Manual. https://developer.arm.com/documentation/100232/latest/.

[4] Arm LTD. 2020. DSTREAM family. https://developer.arm.com/tools-and-software/embedded/debug-probes/dstream-family.

[5] Arm LTD. 2020. Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETM4.6. https://developer.arm.com/documentation/ihi0064/latest.

[6] Arm LTD. 2020. Juno r2 ARM Development Platform SoC. https://developer.arm.com/documentation/ddi0515/f/.

[7] Arm LTD. 2021. Arm TRACE32. https://developer.arm.com/solutions/internet-of-things/tools/trace32.

[8] Arm LTD. 2021. Arm® Architecture Reference Manual. https://developer.arm.com/documentation/ddi0487/latest.

[9] Arm LTD. 2021. CoreSight Components Technical Reference Manual. https://developer.arm.com/documentation/ddi0314/h/trace-port-interface-unit.

[10] Arm LTD. 2021. Learn the architecture: Understanding trace. https://developer.arm.com/documentation/102119/latest/.

[11] BenjaminSchubert. 2015. bugbase. https://github.com/dslab-epfl/bugbase/tree/master/data.

[12] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. 2010. Deterministic Process Groups in dOS.. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.

[13] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments.*

[14] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE'06)*.

[15] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical crash analysis for automated root cause explanation. In *Proceedings of the 29th USENIX Conference on Security Symposium (USENIX Security'20)*.

[16] Ning Chen and Sunghun Kim. 2014. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering* 41, 2 (2014), 198–220.

[17] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. 2009. Holmes: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering (ICSE'09)*.

[18] 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'18)*.

[19] 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 *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*.

[20] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. 2014. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[21] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP'20)*.

[22] Peter Dinges and Gul Agha. 2014. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 31–36.

[23] Yunlan Du, Zhenyu Ning, Jun Xu, Zhilong Wang, Yueh-Hsun Lin, Fengwei Zhang, Xinyu Xing, and Bing Mao. 2020. HART: Hardware-Assisted Kernel Module Tracing on Arm. In *Computer Security (ESORICS'20)*.

[24] GW Dunlap, ST King, S Cinar, M Basrai, and PM Chen. 2002. Enabling Intrusion Analysis through Virtual Machine Logging and Replay. In *Proc. 2002 Symp. Operating Systems Design and Implementation (OSDI'02)*.

[25] Jakob Engblom, Daniel Aarno, and Bengt Werner. 2010. Full-system simulation from embedded to high-performance systems. In *Processor and System-on-Chip Simulation (PSCS'10)*.

[26] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse debugging of kernel failures in deployed systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 281–292.

[27] geeksforgeeks. 2017. Pure Functions. https://www.geeksforgeeks.org/pure-functions/.

[28] 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 (SOSP'09)*.

[29] Chris Gottbrath. 2008. Reverse debugging with the TotalView debugger. In *Cray User Group Conference (CUGC'08)*.

[30] Brendan Gregg. 2019. DTrace Tools. http://www.brendangregg.com/dtrace.html.

[31] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* (1st edition ed.).

[32] Derek R Hower and Mark D Hill. 2008. Rerun: Exploiting episodes for lightweight memory race recording. *ACM SIGARCH computer architecture news* 36, 3 (2008).

[33] hwangcc23. 2021. ptm2human: ARM PTM (and ETMv4) trace to human-readable format. https://github.com/hwangcc23/ptm2human.

[34] jieyu. 2013. concurrency-bugs. https://github.com/jieyu/concurrency-bugs.

[35] jieyu. 2015. maple bug bases. https://github.com/jieyu/maple/tree/master/example.

[36] 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 (OOPSLA'10)*.

[37] Wei Jin and Alessandro Orso. 2012. Bugredux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 474–484.

[38] 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 (SOSP'17)*.

[39] 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 (SOSP'15)*.

[40] kdlucas. 2018. byte-unixbench. https://github.com/kdlucas/byte-unixbench.

[41] Jiashuo Liang, Guancheng Li, Chao Zhang, Ming Yuan, Xingman Chen, and Xinhui Han. 2020. RIPT–An Efficient Multi-Core Record-Replay System. In

Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS'20).

[42] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. 2003. Bug isolation via remote program sampling. *ACM Sigplan Notices* (2003).

[43] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. 2003. Sampling user executions for bug isolation. In *Proceedings of the Workshop on Remote Analysis and Measurement of Software Systems.*

[44] Linaro. 2021. arm-reference-platforms. https://git.linaro.org/landing-teams/working/arm/arm-reference-platforms.git/about/docs/user-guide.rst.

[45] Linux man-pages project. 2021. core(5) — Linux manual page. https://man7.org/linux/man-pages/man5/core.5.html.

[46] Linux man-pages project. 2021. gcore(1) — Linux manual page. https://man7.org/linux/man-pages/man1/gcore.1.html.

[47] Linux man-pages project. 2021. ptrace(2) - Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html.

[48] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. IReplayer: In-Situ and Identical Record-and-Replay for Multithreaded Applications. *SIGPLAN Not.* (2018).

[49] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS'08).*

[50] Nuno Machado, Paolo Romano, and Luís Rodrigues. 2018. CoopREP: Cooperative record and replay of concurrency bugs. *Software Testing, Verification and Reliability (STVR'18)* (2018).

[51] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards practical default-on multi-core record/replay. *ACM SIGPLAN Notices (SIGPLAN'17)* 52, 4 (2017).

[52] Desnoyers Mathieu. 2021. Using the Linux Kernel Tracepoints — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[53] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently. *ACM SIGARCH Computer Architecture News (SIGARCH'08)* 36, 3 (2008).

[54] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP'19).* IEEE.

[55] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Security Symposium (USENIX Security'17).*

[56] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC'17).*

[57] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. 2009. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09).*

[58] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization.* 2–11.

[59] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. 2011. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11).*

[60] James R. 2013. Intel Processor Tracing. https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html.

[61] retrage. 2018. SQLite3 Benchmark. https://github.com/ukontainer/sqlite-bench.

[62] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS'99)* (1999).

[63] Caitlin Sadowski and Jaeheon Yi. 2014. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'14).*

[64] Segger. 2021. J-Trace PRO. https://www.segger.com/products/debug-probes/j-trace/.

[65] Segger. 2021. SEGGER J-Links. https://www.segger.com/downloads/jlink/.

[66] Inc Sysdig. 2021. draios/sysdig. https://github.com/draios/sysdig.

[67] The strace developers. 2021. strace/strace. https://github.com/strace/strace.

[68] thinkingfish. 2014. twemperf. https://github.com/twitter-archive/twemperf.

[69] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M Chen, Jason Flinn, and Satish Narayanasamy. 2012. DoublePlay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS'12)* (2012).

[70] 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 (CCS'16).*

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

[72] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *30th USENIX Security Symposium (USENIX Security'21).*

[73] Jie Yu and Satish Narayanasamy. 2009. A case for an interleaving constrained shared-memory multi-processor. *ACM SIGARCH Computer Architecture News (CAN'09)* (2009).

[74] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA'12).*

[75] Cristian Zamfir and George Candea. 2010. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems (EUROSYS'10).*

[76] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. SHELTER: Extending Arm CCA with Isolation in User Space. In *32nd USENIX Security Symposium (USENIX Security'23).*

[77] 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 (SOSP'19).*

[78] Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Yajin Zhou, Chao Zhang, and Haipeng Cai. 2022. NCScope: hardware-assisted analyzer for native code in Android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).*

Received 2023-02-16; accepted 2023-05-03