REPT: Reverse Debugging of Failures in Deployed Software

Weidong Cui and Xinyang Ge, ***Microsoft Research Redmond;***
Baris Kasikci, ***University of Michigan;*** **Ben Niu,** ***Microsoft Research Redmond;***
Upamanyu Sharma, ***University of Michigan;*** **Ruoyu Wang,** ***Arizona State University;***
Insu Yun, ***Georgia Institute of Technology***

https://www.usenix.org/conference/osdi18/presentation/weidong

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

REPT: Reverse Debugging of Failures in Deployed Software

Weidong Cui[1], Xinyang Ge[1], Baris Kasikci[2], Ben Niu[1], Upamanyu Sharma[2], Ruoyu Wang[3], and Insu Yun[4]

[1]Microsoft Research
[2]University of Michigan
[3]Arizona State University
[4]Georgia Institute of Technology

Abstract

Debugging software failures in deployed systems is important because they impact real users and customers. However, debugging such failures is notoriously hard in practice because developers have to rely on limited information such as memory dumps. The execution history is usually unavailable because high-fidelity program tracing is not affordable in deployed systems.

In this paper, we present REPT, a practical system that enables *reverse debugging* of software failures in deployed systems. REPT reconstructs the execution history with high fidelity by combining online lightweight hardware tracing of a program's control flow with offline binary analysis that recovers its data flow. It is seemingly impossible to recover data values thousands of instructions before the failure due to information loss and concurrent execution. REPT tackles these challenges by constructing a partial execution order based on timestamps logged by hardware and iteratively performing forward and backward execution with error correction.

We design and implement REPT, deploy it on Microsoft Windows, and integrate it into WinDbg. We evaluate REPT on 16 real-world bugs and show that it can recover data values accurately (92% on average) and efficiently (in less than 20 seconds) for these bugs. We also show that it enables effective reverse debugging for 14 bugs.

1 Introduction

Software failures in deployed systems are unavoidable and debugging such failures is crucial because they impact real users and customers. It is well known that execution logs are helpful for debugging [28], but nobody wants to pay a high performance overhead for always-on logging/tracing when most logs or traces would be discarded for normal runs. As a result, only a memory dump is captured upon failures in deployed software to enable post-mortem diagnosis.

Alas, it is challenging for developers to debug memory dumps due to limited information. The result is that a significant fraction of bugs is left unfixed [32,59]. Those that get fixed can take weeks in certain cases [32].

To make matters worse, streamlined software processes call for short release cycles [53], which limits the extent of in-house testing prior to software release. Frequent releases increase the dependency on debugging failures reported from deployed software, because these failure occurrences become the only way to detect certain bugs. Frequent releases also increase the demand for quickly resolving bugs to meet short release deadlines.

There exists a rich literature on debugging failures, which can roughly be classified into two categories:

(1) *Automatic root cause diagnosis* [16, 37–41, 61] attempts to automatically determine the culprit statements that cause a program to fail. Due to various limitations (e.g., requiring code modification [37, 40, 41], inability to handle complex software efficiently [37, 61], or being limited to a subset of failures [37, 39]), none of these systems are deployed in practice. Moreover, even though root cause diagnosis can help a developer determine the reasons behind a failure, developers often require a deeper understanding of the conditions and the state leading to a failure to fix a bug, which these systems do not provide.

(2) *Failure reproduction* for debugging attempts to enable developers to examine program inputs and state that lead to failures. Exhaustive testing techniques such as symbolic execution [22] and model checking [21, 58], or state-space exploration [51] can be used to determine inputs and state that lead to a failure for the purpose

of debugging. Unfortunately, these techniques require heavyweight runtime monitoring [26]. Another popular technique for reproducing failures is record/replay systems [46, 48, 50, 52, 56] that record program executions that can later be replayed to debug failures. This is also known as *reverse debugging* [31, 55] or *time-travel debugging* [44]. On the plus side, reverse debugging allows a developer to go back and forth in a failed execution to examine a program's state (i.e., control and data flow) to truly understand the bug and devise a fix. On the other hand, record/replay systems incur prohibitive overhead (up to 200% for the state-of-the-art system [56]) in multithreaded programs running on multiple cores, making them impractical for use in deployed systems.

Due to the limitations of existing techniques, major software vendors including Apple [17], Google [33], and Microsoft [30] as well as open-source systems such as Ubuntu [54] operate error reporting services to collect data about failures in deployed software and analyze them. To our knowledge, even the most advanced bug diagnosis system deployed in production, namely RETracer [27], is only able to *triage* failures caused by access violations.

To solve the challenge of debugging software failures in deployed systems, we argue that we need a *practical* solution that enables reverse debugging of such failures. To be practical, the solution must (1) impose a very low runtime performance overhead when running on a deployed system, (2) should be able to recover the execution history accurately and efficiently, (3) work with unmodified source code/binary, (4) apply to broad classes of bugs (e.g., concurrency bugs).

In this paper, we present REPT[1], a practical solution for reverse debugging of software failures in deployed systems. There are two key ideas behind REPT. First, REPT leverages *hardware* tracing to record a program's control flow with low performance overhead. Second, REPT uses a novel binary analysis technique to recover data flow information based on the logged control flow information and the data values saved in a memory dump. Consequently, REPT enables reverse debugging by combining the logged control flow and the recovered data flow.

The main challenge faced by REPT is how to *accurately* and *efficiently* recover data values based on the logged control flow and the data values saved in the memory dump. To be *accurate*, REPT must be able to correctly recover a significant fraction of data values in the execution history. To be *efficient*, REPT must incur low runtime monitoring overhead and should finish its analysis within minutes. To solve this challenge, we introduce a new binary analysis approach that combines *forward* and *backward* execution to *iteratively* emulate instructions and recover data values. REPT uses the following two new techniques for its analysis:

First, we design an error correction scheme to detect and correct value conflicts that are introduced by memory writes to unknown addresses. When emulating a memory write instruction, it is too conservative to mark all memory values as unknown if the destination address is unknown. Instead, REPT leaves memory untouched and relies on detecting a conflict later caused by stale values in the destination memory. Unlike previous solutions that use expensive hypothesis tests to decide memory aliases [57], the error correction scheme enables REPT to run its iterative analysis efficiently.

Second, we leverage the timing information provided by modern hardware to determine the order of non-deterministic events such as races across multiple threads. Non-determinism has been a long-standing challenge that hinders the ability of existing record/replay systems to achieve high accuracy with low overhead. REPT can identify the order of accesses to the same memory location in most cases by using fine-grained timestamps that modern hardware provides. When the timing information is not enough, REPT restricts the use of memory accesses whose order cannot be inferred. This stops their values from negatively affecting the recovery of other data.

We implement REPT in two components. The online tracing component is a driver that controls Intel Processor Trace (PT) [36], and has been deployed on hundreds of millions of machines as part of Microsoft Windows. The offline binary analysis component is a loadable library that is integrated into WinDbg [45]. We also enhance Windows Error Reporting (WER) service [30] to control hardware tracing on deployed systems.

To measure the effectiveness and efficiency of REPT, we evaluate it on 16 real-world bugs in software such as Chrome, Apache, PHP, and Python. Our experiments show that REPT can enable effective reverse debugging for 14 of them, including 2 concurrency bugs. We evaluate REPT's data recovery accuracy by comparing its recovered data values with those logged by Time Travel Debugging (TTD) [44], a slow but precise record/replay tool. Our experiments show that REPT can achieve an average accuracy of 92% and finish its analysis in less than 20 seconds for these bugs.

[1] REPT stands for Reverse Execution with Processor Trace and reads as "repeat."

2 Overview

2.1 Problem Statement

The overarching goal of REPT is to enable reverse debugging of failures in deployed software with low runtime overhead. REPT realizes reverse debugging in two steps. (1) REPT uses hardware support to log the control flow and timing information of a program's execution. When a failure occurs, REPT saves an enriched memory dump including both the final program state and the additionally recorded control flow and timing information before the failure. (2) REPT uses a new offline binary analysis technique to recover data values in the execution history based on the enriched memory dump.

REPT needs to recover data values because there is no existing hardware support for efficiently logging all data values of a program's execution. However, there exist hardware features such as Intel PT [36] and ARM Embedded Trace Macrocell [18] that can efficiently log the control flow and timing information.

2.2 Design Choices

When designing REPT, we make three design choices.

Memory Dump Only vs. Online Data Capture: We choose to only rely on the data in a memory dump rather than logging more data during execution to minimize the performance overhead for deployed systems. Furthermore, to do online data capture, we would need to modify the operating system or programs because there is no existing hardware support for that. We choose not to do it to minimize intrusiveness.

Binary vs. Source: We choose to do the analysis at the binary level instead of at the source code level for three reasons. First, by performing analysis at the instruction level, REPT is essentially agnostic to programming languages and compilers. This allows REPT to support native languages (e.g., C/C++) as well as managed languages (e.g., C#). Second, today's applications often consist of multiple modules/libraries from different vendors, and not all source code may be available for analysis [25]. Third, the mapping between the source code and binary instructions is not straightforward due to compiler optimizations and the use of temporary variables, thus converting source-level analysis result back to the binary-level presents a non-trivial challenge.

Concrete vs. Symbolic: One popular approach to reconstructing executions is symbolic execution. In symbolic execution, a program is *executed* with *symbolic* inputs of *unconstrained* values (e.g., a Boolean can initially take any of the true or false values) as opposed to concrete ones. As the program executes, symbolic execution gathers constraints on symbolic values. Whenever an event of interest occurs (e.g., a failure), symbolic execution uses a constraint solver to determine the program inputs that would have led to that failure. Conceptually, symbolic execution may help with recovering data values. We could treat operands such as registers and memory locations referenced by each instruction as variables, and generate constraints among these variables based on the semantics of the instructions. However, given a long execution trace, the constraints gathered on the variables may grow too large (particularly when memory locations are made symbolic) to solve within a reasonable amount of time for even state-of-the-art constraint solvers. Therefore, we choose to do concrete execution instead of symbolic execution. REPT keeps concrete values for registers and memory locations at each position in the instruction sequence and analyzes each instruction to recover concrete values of its operands.

2.3 Challenges

To enable reverse debugging, REPT faces three challenges when recovering register and memory values in the execution history.

2.3.1 Irreversible Instructions

This first challenge for REPT *is handling irreversible instructions*. If every instruction is *reversible* (i.e., the program state before an instruction's execution can be fully determined based on the program state after its execution), then the design of REPT would be straightforward: invert each instruction's semantics and recover data values at each position in the instruction sequence. However, many instructions are *irreversible* (e.g., `xor rax,rax`) and thus information destroying. We solve this challenge by using forward execution to recover values that cannot be recovered in backward execution.

2.3.2 Missing Memory Writes

The second challenge for REPT is *handling memory writes to unknown addresses*. Most memory addresses cannot be determined statically. Since the analysis may not fully recover data values due to irreversible instructions, REPT may not know the destination of a memory write during its analysis. When this happens, one option is to assume that values at *all* memory locations become unknown. This is too conservative because it may cause the analysis to miss many data values that are actually recoverable. If REPT chooses to ignore the memory

write, the analysis will leave an invalid value at the memory location, which may propagate into other registers or memory locations. We solve this challenge by using error correction.

2.3.3 Concurrent Memory Writes

The third challenge for REPT is *correctly identifying the order of shared memory accesses*. In the presence of multiple instruction sequences from different threads, it may not be possible to infer the execution order of concurrent memory accesses despite timestamps provided by hardware. REPT needs to properly handle these memory accesses, otherwise it may infer wrong values for these memory locations. We solve this challenge by restricting in the analysis the use of data values recovered from concurrent memory accesses.

3 Design

In this section, we describe the design of REPT by focusing on how it solves the three key technical challenges discussed in the previous section.

For brevity, we define an instruction sequence as $I = \{I_i | i = 1, 2, ..., n\}$ where I_i represents the i-th instruction executed in the sequence. We assume that the memory dump is available after the n-th instruction's execution. We define a program's state, S, as a collection of all data values in registers and memory locations. We define S_i as the program state after the i-th instruction is executed. Therefore, S_0 represents the program state before the first instruction I_1 is executed, and S_n represents the program state stored in the memory dump. We define a state S_i as *complete* if all the register and memory values are known. We define an instruction I_i as *reversible* if, given a complete state S_i, we can recover S_{i-1} completely; otherwise we say the instruction is *irreversible*. The design of REPT is not limited to a specific architecture, however, in the rest of the paper, we use x86-64 instructions in our examples.

In the rest of this section, we present the design of REPT progressively by describing how it handles increasingly more complex and realistic scenarios.

- A single instruction sequence with only **reversible** instructions (Section 3.1).
- A single instruction sequence with **irreversible** instructions but without memory accesses (Section 3.2).
- A single instruction sequence with irreversible instructions and **with memory accesses** (Section 3.3).
- **Multiple** instruction sequences with irreversible instructions and with memory accesses (Section 3.4).

3.1 Instruction Reversal

REPT's first mechanism assumes that the input is a single instruction sequence with only reversible instructions. Since every instruction is reversible, REPT can reverse the effects of each instruction to completely recover the initial program state from the end of the instruction sequence to the beginning. For instance, if the instruction sequence has a single instruction I_1 = `add rax,rbx` and S_1 = {`rax`=3, `rbx`=1}, then the analysis can recover S_0 = {`rax`=2, `rbx`=1}.

3.2 Irreversible Instruction Handling

REPT's second mechanism assumes that there is a single instruction sequence with irreversible instructions, but the sequence does not include any memory access. In practice, most instructions are irreversible. For instance, `xor rbx,rbx` is irreversible, because `rbx`'s value before the instruction is executed cannot be recovered simply based on this instruction's semantics and `rbx`'s value after the instruction is executed. Therefore, the straightforward backward analysis for reversible instructions is not applicable in general.

The key idea for recovering a *destroyed* value is to infer it in a *forward* analysis. As long as the destroyed value is derived from some other registers and memory locations, and their values are available, we can use these values to recover the destroyed value. Extending this idea, our basic solution is to iteratively perform backward and forward analysis to recover data values until no new values are recovered.

Conceptually, given the instruction sequence I and the final state S_n, we first mark all register values as *unknown* in program states from S_0 to S_{n-1}. Then we do backward analysis to recover program states from S_{n-1} to S_0. After this step, we perform forward analysis to update program states from S_0 to S_{n-1}. We repeat these steps until a *fixed point* is reached: i.e., no state is updated in a backward or forward analysis. When we update a program state, we only change a register's value from unknown to an inferred value. Crucially, this analysis will not produce conflicting inferred values because all the initial values are correct and no step in the analysis can introduce a wrong value based on correct values. This also guarantees that the iterative analysis will converge.

We show an example of handling irreversible instructions in Figure 1. The instruction sequence has three instructions, and two of them are irreversible. Since we do

			Iteration 1	Iteration 2	Iteration 3
		S_0	↑{rax=?, rbx=?} →	↓	↑{rax=2, rbx=?}
I_1	`mov rbx, 1`	S_1	↑{**rax=?**, rbx=?}	↓{rax=?, **rbx=1**}	↑{**rax=2**, rbx=1}
I_2	`add rax, rbx`	S_2	↑{rax=3, **rbx=?**}	↓{**rax=3**, rbx=1}	↑{rax=3, rbx=1}
I_3	`xor rbx, rbx`	S_3	↑{rax=3, rbx=0}	↓{rax=3, rbx=0} →	↑

Figure 1: This example shows how REPT's iterative analysis recovers register values in the presence of irreversible instructions. We use "?" to represent "unknown". Key updates during the analysis are marked in bold face.

			Iteration 1	Iteration 2	Iteration 3
		S_0	↑{rax=?, rbx=?, [g]=3} →		↑{rax=?, rbx=?, [g]=2}
I_1	`lea rbx, [g]`	S_1	↑{rax=?, rbx=?, [g]=3}	↓{rax=?, rbx=g, [g]=3}	↑{rax=?, rbx=g, [g]=2}
I_2	`mov rax, 1`	S_2	↑{rax=?, rbx=?, [g]=3}	↓{rax=1, rbx=g, [g]=3}	↑{rax=1, rbx=g, **[g]=2**}
I_3	`add rax, [rbx]`	S_3	↑{rax=3, rbx=?, **[g]=3**}	↓{**rax=3**, rbx=g, [g]=3}	↑{rax=3, rbx=g, **[g]=?**}
I_4	`mov [rbx], rax`	S_4	↑{rax=3, rbx=?, [g]=3}	↓{rax=3, rbx=g, [g]=3}	↑{rax=3, rbx=g, [g]=3}
I_5	`xor rbx, rbx`	S_5	↑{rax=3, rbx=0, [g]=3}	↓{rax=3, rbx=0, [g]=3} →	

Figure 2: This example shows how REPT's iterative analysis recovers register and memory values when there exist irreversible instructions with memory accesses. We use "?" to represent "unknown", and use "g" to represent the memory address of a global variable. Some values are in bold-face because they represent key updates in the analysis. We skip the fourth iteration which will recover [g]'s value to be 2 due to the space constraint.

not have instructions before the first one, we do not expect to recover `rbx` in S_0. There are three points that are worth noting in this example. First, we recover `rbx`'s value in S_1 based on the forward analysis in the second iteration. Second, we keep `rax`'s value of 3 in S_2 in the second iteration of forward analysis even though `rax`'s value is unknown in S_1. Third, we recover `rax`'s value of 2 in S_1 in the last iteration of backward analysis.

3.3 Recovering Memory Writes

REPT's third mechanism assumes that there is a single instruction sequence with irreversible instructions and with memory accesses. In practice, there are always instructions that access memory. Unlike registers that can be statically identified from instructions, the address of a memory access may not always be known. For a memory write instruction whose destination is unknown, we cannot correctly update the value for the destination memory. A missing update may introduce an obsolete value, which would negatively impact subsequent analysis. A conservative approach that marks all memory as unknown upon a missing memory write would lead to an unnecessary and unacceptable information loss.

Our key insight for solving the missing memory write problem is to use *error correction*. The intuition behind REPT is to keep using the memory values that are possibly valid to infer other values, and to correct the values later if the values turn out to be invalid based on conflicts. Before describing REPT's error correction algorithm, we first use an example to explain the high-level idea.

The example in Figure 2 has five instructions. There are three key updates as marked in bold face. In the first iteration of the backward analysis, since we do not know `rbx`'s value in S_4, we do not change the value at the address `g`. In the second iteration of the forward analysis, there is a conflict for `rax` in S_3. The original value is 3, but the newly inferred value would be 4 (`rax` + `[g]` = 1 + 3 = 4). Our analysis keeps the original value of 3 because it was inferred from the final program state which we assume is correct. In the third iteration of the backward analysis, based on `rax`'s value before and after the instruction I_3, we can recover `[g]`'s value to be 2.

Next, we describe the algorithm that REPT uses to recover missing memory writes. We first introduce the data inference graph in Section 3.3.1, and then explain how we use the graph to detect and correct errors caused by missing memory writes in Section 3.3.2.

3.3.1 Data Inference Graph

When performing the backward and forward analysis, REPT maintains a *data inference graph*. The data inference graph is different from a traditional data flow graph in the sense that it tracks how a data value is inferred in either forward or backward directions while a data flow graph tracks the program's data flow in just one direction.

An example data inference graph is shown in Figure 3. In this example, we use `rcx` to recover `[rax]`, and then use the latter to recover `rbx`. Here we assume that `rax`'s value is not changed between I_1 and I_n.

A node in the data inference graph represents a register or a memory location that is accessed in an executed instruction. A node is called a *use* node if its correspond-

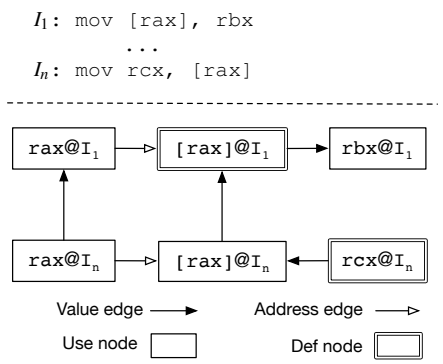

Figure 3: An example data inference graph in REPT. The graph indicates that REPT uses `rcx`@I_n to recover `[rax]`@I_n, which is further used to recover `[rax]`@I_1 and subsequently `rbx`@I_1.

ing register or memory location is for read. Similarly, a node is called a *def* node if it is for write. For instance, `rbx`@I_1 is a use node, and `rcx`@I_n is a def node. If a register or memory location is accessed for both read and write in a single instruction, we create two nodes for it: one use node, and one def node. Finally, REPT treats data in the memory dump as use nodes because their values can be propagated backwards like other use nodes.

There are two kinds of directional edges in the data inference graph: *value edges* and *address edges*. A value edge from node A to node B means that REPT uses A's value to infer B's value. An address edge from A to B means that A's value is used to compute B's address. For instance, the edge from `rcx`@I_n to `[rax]`@I_n is a value edge, and the edge from `rax`@I_n to `[rax]`@I_n is an address edge. To get or set the value of a memory location, its address must be known. When setting a memory node's value, besides value edges, REPT adds address edges from register nodes that are used to compute the address of the memory node. A memory node can have multiple incoming address edges (e.g., a base register and an index register are used together to specify the address).

There are two types of value edges. In the first type of value edges, the connected nodes are from the same instruction and we call them *horizontal* edges. Specifically, in the backward analysis, if a def node's value is known and can be used to infer the value of a use node in the same instruction, we recover the use node's value and add a horizontal edge between the two nodes. Similarly, in the forward analysis, if a use node's value is known and can be used to infer the value of a def node in the same instruction, we recover the def node's value and add a horizontal edge between the nodes as well. It is worth noting that a node may have multiple horizontal incoming value edges. For instance, given `add rax,rbx`, the def node of `rax` can have two incoming value edges from the use nodes of `rax` and `rbx`.

In the second type of value edges, the connected nodes are from different instructions, but they correspond to the same register or memory location. Such value edges are referred to as *vertical* edges. Intuitively, nodes connected via vertical edges belong to the same def-use chain (i.e., a single def with all its reaching uses). In the backward analysis, we recover values from a use node to the preceding use node or the def node along the def-use chain, and add vertical edges in between. Similarly, in the forward analysis, we recover values from a def or use node to its subsequent use node along the def-use chain and add corresponding vertical edges as well. In other words, a def node's value can only be propagated forwardly while a use node's value can be propagated on both directions.

For every node in the data inference graph, REPT also maintains a *dereference level* to aid in error correction (Section 3.3.2). Specifically, all use nodes of values in the memory dump have a dereference level of 0. For any other node, REPT determines its dereference level in three steps: (1) for all incoming value edges, find the maximum dereference level of the source nodes as D_1; (2) for all incoming address edges, find the maximum dereference level of the source nodes as D_2; (3) pick the larger value between D_1 and D_2+1 as the target node's dereference level. We can see that the dereference level actually measures the maximum number of address edges from a value stored in the memory dump to the given node. A node's dereference level reflects the confidence level for its value since data inference errors come from memory due to missing memory writes. A higher dereference level means a lower confidence level.

3.3.2 Error Correction

During the iterative backward and forward analysis, REPT continuously updates the data inference graph and detects and corrects inconsistencies. There are two kinds of inconsistencies: *value* conflict and *edge* conflict. A value conflict happens when an inferred value does not match the existing value. An edge conflict happens when a newly identified def node of a memory location *breaks* the previously assumed def-use relationship between two nodes connected through a vertical edge. Consider the example in Figure 3. If REPT detects another write to the same memory location specified by `rax` between I_1 and I_n, this memory write will cause a conflict on the

vertical edge between `[rax]`@I_n and `[rax]`@I_1.

When REPT detects a conflict, it stops the analysis of the current instruction, identifies the invalid node, then runs the invalidation process. For both types of conflicts, the invalidation process starts with an initial node. In the case of edge conflicts, the initial node is the target node of the broken vertical edge as it no longer belongs to the same def-use chain. In the case of value conflicts, REPT checks if the dereference level of the node of the newly inferred value is less than or equal to that of the node of the existing value (this means a higher or equal confidence for the new value). If so, REPT picks the node of the existing value as the initial node for invalidation. Otherwise, REPT discards the newly inferred value and moves on to the next instruction.

If REPT identifies an initial node for invalidation, it first processes each of its outgoing value and address edges. For a value edge, the target node is marked as unknown. For an address edge, the target node is deleted from the data inference graph since its address becomes unknown and consequently such a def or use on that memory location may no longer exist. Then REPT recursively applies the invalidation process to these target nodes. It is worth noting that the data inference graph is guaranteed *not* to have cycles, because REPT adds a node and edges into the graph only when the node's value is inferred for the first time.

To ensure convergence of the analysis, REPT maintains a *blacklist* of invalidated values for each node. Every time a node is invalidated, its value is added to its blacklist. Once a value is in a node's blacklist, the node cannot take that value any more. This ensures that the iterative analysis process will not enter the conflicting state again and consequently guarantees that the algorithm will eventually converge. However, a correct value can be incorrectly blacklisted for a node if it has a lower confidence level than another incorrect value. This leads to the problem that a value is recoverable but cannot be recovered due to the use of the blacklist. We choose to keep the blacklists to prioritize the convergence of the analysis over the improvement in data recovery.

3.4 Handling Concurrency

When we face multiple instruction sequences executed simultaneously on multiple cores, the problem is seemingly intractable because, without a perfect order of the executed instructions, there could be a large number of ways to order those instructions. We have two insights for tackling this challenge. First, we leverage the timing information logged by hardware tracing to construct a partial order of instructions executed in different threads. Second, we recognize that memory writes are the *only* operations whose orders may affect data recovery.

With timestamps inserted in an instruction sequence, we refer to the instructions between two timestamps as an instruction *subsequence*. We refer to the two timestamps as the start and end time of the subsequence. Given two instruction subsequences from two different instruction sequences, we infer their relative execution order based on their start and end times. If one subsequence's end time is before another subsequence's start time, we say the first subsequence is executed *before* the other subsequence. Otherwise, we say their order cannot be inferred, and the two subsequences are *concurrent*. Note that the order of two subsequences in the same instruction sequence can always be determined based on their positions in the instruction sequence. We say two instructions are *concurrent* if the instruction subsequences they belong to are concurrent. We say two memory accesses are *concurrent* if the corresponding memory access instructions are concurrent.

Given multiple instruction sequences executed simultaneously on multiple cores, REPT first divides them into subsequences, then merges them into a *single conceptual* instruction sequence based on the inferred orders. For two subsequences whose order cannot be inferred, REPT arbitrarily inserts one before the other in the newly constructed sequence. A natural question is whether the data recovery is affected by this arbitrary choice of ordering two concurrent subsequences. Obviously, if we change the order of two subsequences that have concurrent memory accesses to the same location and one of them is write, we may get different values for the memory location. On the other hand, if concurrent subsequences do *not* have any concurrent memory write to the same location, it does not matter in which order REPT places them into the merged instruction sequence.

Since we cannot tell the order of concurrent instruction subsequences, our goal is to *eliminate* the impact of their ambiguous order on data recovery. Specifically, during the iterative analysis, for every memory access (regardless of read or write), REPT detects if it has a concurrent memory write to the same location. If so, REPT takes the following steps to limit the use of the memory access in the data inference graph. First, REPT removes all vertical edges of the node representing the memory access and invalidates the target nodes of outgoing vertical edges. Then, REPT labels the memory access node so that it will not be used in vertical edges. This is because REPT does not know if the memory access happens before or after the concurrent memory write to the same

location. However, REPT still allows horizontal value edges to infer this node's value.

A remaining question is whether picking an arbitrary order for concurrent instruction subsequences would affect the detection of concurrent memory writes to the same location. Our observation is that REPT's analysis works as long as there are no two separate concurrent writes such that one affects the inference of another's destination. We acknowledge that this possibility exists and depends on the granularity of timing information. Given the timestamp granularity supported by modern hardware, we deem this as a rare case in practice [39].

4 Implementation

In this section, we first describe the implementation details of REPT's online hardware tracing and offline binary analysis. Then we describe its deployment.

4.1 Online Hardware Tracing

REPT leverages Intel Processor Trace (PT) to log control-flow and timing information of a program's execution. Intel PT became available when the Broadwell architecture was released in 2014. Intel PT supports various program tracing modes, and REPT currently uses the *per-thread circular* buffer mode to trace user-space execution of all threads within a process. REPT supports configuring the circular buffer size and the granularity of timestamps. We do not configure Intel PT to do whole-execution tracing because that would introduce performance overhead due to frequent interrupts (when the trace buffer gets full) and I/O workload (when the buffer is written to some persistent storage). When a traced process fails, its final state and the recorded Intel PT traces are saved in a single memory dump.

4.2 Offline Binary Analysis

REPT takes a memory dump with Intel PT trace as input, and outputs the recovered execution history of each thread. At first, REPT parses the trace to reconstruct the control flow. Parsing an Intel PT trace requires that the binary code in the dump is the same as the code that was executed when the trace is collected. Therefore, REPT supports jitted code as long as the code was not modified since its execution was logged in the circular trace buffer. Next, REPT converts native instructions into an intermediate representation (IR) that specifies opcodes and operands, and conducts the forward and backward analysis until it converges.

In addition to the final program state and constants, REPT can leverage control dependencies to recover data. For instance, if a conditional branch is executed only if a register's value is 0, then REPT can infer the register's value once it observes that the branch is taken.

Programs invoke system calls to request operating system services, and the operating system may modify certain register and memory values in the process as a response. Upon a system call, REPT will mark all volatile registers as unknown based on the calling convention. REPT currently does not handle memory writes by the kernel, but instead treats those in the same way as missing memory writes and relies on the error correction mechanism to detect and resolve conflicts. We acknowledge that semantic-aware handling of system calls can be done with more engineering effort to help improve the data recovery, but we leave it to future work.

4.3 Deployment

We implement REPT in two components and deploy it into the ecosystem of Microsoft Windows for program tracing, failure reporting, and debugging.

First, we implement the online hardware tracing component as a driver of 8.5K lines of C code. It is responsible for controlling tracing of a target process and capturing the trace in a memory dump when the monitored process fails. We also modify the Windows kernel to support per-thread tracing by swapping the trace buffers upon context switch.

Second, we implement REPT's offline binary analysis and reverse debugging as a library of 100K lines of C++ code, and integrate it into WinDbg [45]. We also implement common debugging functionalities such as code and data breakpoints to facilitate the debugging process.

We enhance the Windows Error Reporting (WER) service [30] to support REPT. Specifically, developers can request Intel PT enriched memory dumps on WER. Then WER selects user machines to trace the targeted program. When a traced program causes a failure, a memory dump with Intel PT trace is captured and sent back to WER. Finally, developers can load the enriched memory dump in WinDbg to do reverse debugging.

5 Evaluation

In this section, we evaluate REPT to answer the following four questions: (1) How accurately can REPT recover data values? (2) How efficiently can REPT recover data values? (3) How effectively can REPT be used to debug failures? (4) What is the deployment status? Next,

Program-BugId	Bug Type	MP	SS
Apache-24483	NULL pointer deref [1]	No	Yes
Apache-39722	NULL pointer deref [2]	No	Yes
Apache-60324	Integer overflow [3]	No	Yes
Nasm-2004-1287	Stack buffer overrun [4]	No	No
PHP-2007-1001	Integer overflow [5]	No	Yes
PHP-2012-2386	Integer overflow [6]	No	No
PHP-74194	Type confusion [7]	No	No
PHP-76041	NULL pointer deref [8]	No	Yes
PuTTY-2016-2563	Stack buffer overrun [9]	No	No
Python-2007-4965	Integer overflow [10]	No	Yes
Python-28322	Type confusion [11]	No	No
Chrome-784183	Integer overflow [12]	No	No
Pbzip2	Use-after-free [29]	Yes	No
Python-31530	Race [13]	Yes	No
Chrome-776677	Race [14]	Yes	No
LibreOffice-88914	Deadlock [15]	Yes	No

Table 1: Software bugs used in our experiments. MP means that the defect and failure threads are different. SS means that the defect is on the same stack as the failure.

we present our experimental setup and describe our experimental results to answer these questions.

We evaluate REPT on failures caused by 16 real-world bugs listed in Table 1. All of these bugs are from open-source software. We focus on open-source software for independent reproducibility. The main constraint that limits us from evaluating REPT on more bugs is that we need to reproduce bugs in open-source software on Microsoft Windows. When reproducing bugs, we try to pick bugs that are from a diverse set of widely-used real-world systems (e.g., Apache, Python, Chrome and PHP) and from a wide spectrum of bug types (e.g., NULL pointer dereference, race, type confusion, use-after-free, integer overflow, and buffer overflow).

In our experiments, we configure Intel PT to use a circular buffer of 256K bytes per thread and turn on the most fine-grained timestamp logging (i.e., TSCEn=1, CYCEn=1, CycThresh=0 and MTCFreq=0; see [36] for more details).

5.1 Accuracy

To evaluate the accuracy of REPT's data recovery, we need to obtain the ground truth. We use Time Travel Debugging (TTD) [44], a slow but precise record/replay tool, to log both control and data flow of a program's execution. With the fully recorded execution, we create inputs to REPT and check the correctness of its output. To evaluate the accuracy of REPT in handling multiple concurrent instruction sequences, we modify TTD to generate the timing information as an approximation to timestamps generated by Intel PT. Finally, we stress test REPT on a highly concurrent program and report how well the timestamps provided by Intel PT can order shared memory accesses under extreme cases.

Program-BugId	# Insts	Cor	Unk	Inc
Apache-24483	49	96.72%	1.64%	1.64%
Apache-39722	1,644	99.30%	0.70%	0.00%
Apache-60324	672	96.47%	1.83%	1.70%
Nasm-2004-1287	67,726	95.95%	3.70%	0.35%
PHP-2007-1001	54,475	99.08%	0.90%	0.02%
PHP-2012-2386	43,813	71.55%	25.40%	3.05%
PHP-74194	78,103	90.88%	7.82%	1.30%
PHP-76041	115	94.96%	3.60%	1.44%
PuTTY-2016-2563	677	99.55%	0.45%	0.00%
Python-2007-4965	1,043	95.04%	4.09%	0.87%
Python-28322	1,062	90.85%	8.60%	0.55%

Table 2: REPT's accuracy on a single instruction sequence. Cor, Unk and Inc represent the percentage of correct, unknown, and incorrect register uses.

5.1.1 Single-Thread Accuracy

In this experiment, we first use TTD to record the execution where each bug is triggered. Then, we replay the recorded execution to construct an instruction sequence without the timing information for the failure thread. Next, we run REPT on the constructed instruction sequence and the final program state provided by the replay engine. Finally, we compare the recovered data values with the data values returned by the replay engine.

When we compare the data values, we only check *register uses* (i.e., a register used as a source operand or the address of a destination memory operand). We do not check *defs* (i.e., a destination operand) because we want to avoid double counting. For instance, given `mov rax,rcx`, both `rax` and `rcx` will be correct or incorrect at the same time. When computing the data recovery accuracy, we do not need to count both of them. We do not check *memory uses* (i.e., a memory used as a source operand) because memory values are usually read into registers before they take on any operations. We analyze the trace of the 16 bugs and find that the destination is a register for 95% of memory reads. Therefore, we can count the uses of these registers to measure the accuracy.

We present our accuracy measurements in Table 2. Column 2 describes the number of instructions executed from the program defect to the program failure. We identify the location of a program defect based on the bug fix. For instance, Apache-24483 is a NULL pointer dereference bug, and its defect is where the NULL pointer check

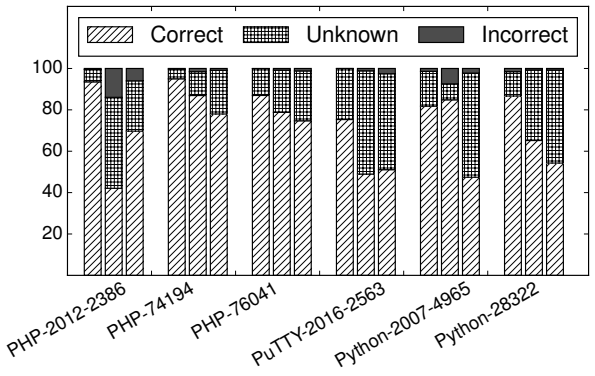

Figure 4: REPT's accuracy on different instruction sequence sizes. For each bug, we limit REPT to analyze 1M instructions, and depict the accuracy for 10K, 100K and 1M instructions away from failure, from left to right.

is added in the bug fix. The rest of three columns show the percentage of correct, unknown and incorrect register uses recovered by REPT in the instruction sequence from the defect to the failure.

We can see that REPT achieves a high accuracy. In most cases, the percentage of correct register uses is above 90% for tens of thousands of instructions; the percentage is still above 80% within 162,208 instructions for the Python-31530 bug. PHP-2012-2386 is an outlier case with the lowest accuracy. This particular bug involves a large number of memory allocation operations right before the program failure. Unfortunately, memory allocation operations are hard to reverse because the metadata information (i.e., chunk sizes) may be completely overwritten by reallocations, resulting in a large percentage of unknowns. We could not obtain the ground truth for Chrome-781483 because TTD does not support Chrome.

We also evaluate how the data recovery accuracy changes as the trace grows. We use instruction sequence sizes of 10K, 100K and 1M, and evaluate 6 bugs, because others have short execution histories. The results are summarized in Figure 4. Overall, the accuracy decreases as the number of instructions increases, and the rate of decrease depends on the program and the workload. It is worth noting that the accuracy does not decrease monotonically as the number of instructions increases. This is expected because REPT's accuracy depends on a program's behavior. For instance, PHP-2012-2386 has the accuracy drop in the case of 100K instructions because these instructions have a large number of memory allocation operations which are hard to reverse.

5.1.2 Multiple-Thread Accuracy

To evaluate REPT's analysis on multiple concurrent executions, we need to emulate the timing information in addition to the control flow from TTD. Currently, TTD supports record and replay of multithreaded programs running on multiple cores by logging timestamps at each system call and synchronization operation (e.g., `cmpxchg`). We extend TTD to log timestamps periodically in a manner similar to Intel PT during recording. When constructing an instruction sequence, we insert TTD's timestamps into the sequence accordingly. We acknowledge that such an approach may not perfectly reflect a multithreaded program's actual behavior on a bare metal machine.We conduct this experiment and report the results as our best estimation of REPT's accuracy for multithreaded programs.

We evaluate REPT on two race condition bugs, Pbzip2 and Python-31530. We do not evaluate Chrome-776677 or LibreOffice-88914 because REPT does not work for them (see Section 5.3). We measure the accuracy on the instructions executed on all threads from the defect to the failure. For Pbzip2, there are 12,496 instructions, and the correct/unknown/incorrect percentages are 95.33%, 4.36%, and 0.31%. For Python-31530, there are 511,289 instructions, and the corresponding percentages are 75.72%, 24.14%, and 0.14%. We attribute the lower accuracy on Python-31530 to the large number of instructions elapsed between the defect and the failure.

Finally, we evaluate how well REPT can use fine-grained timestamps from Intel PT to order memory accesses. We use Racey [34], a stress-testing benchmark that has extremely frequent data races—each thread races with other threads to constantly read/write a shared array for updating a signature. We run Racey with 8 threads for 1000 iterations and instrument it to save the addresses of memory accesses to the shared array. To minimize the instrumentation's impact on timing, we store the memory addresses to a pre-allocated buffer. We measure the fraction of memory accesses that have concurrent memory writes to the same location. We find that 5.5% of accesses to the shared array have concurrent memory writes. Given Racey is an extreme case of concurrent programs, we believe that the granularity of timestamps provided by Intel PT is sufficient for a majority of real-world programs.

5.2 Efficiency

Efficiency of REPT has two prongs, the performance overhead caused by Intel PT when a program is running, and REPT's offline analysis for data recovery. The for-

Program-BugId	# Iters	REPT (s)
Apache-24483	4	5.8
Apache-39722	5	3.0
Apache-60324	2	5.5
Chrome-784183	6	8.2
Nasm-2004-1287	10	18.6
Pbzip2	7	8.2
PHP-2007-1001	5	2.0
PHP-2012-2386	6	3.8
PHP-74194	7	6.3
PHP-76041	6	14.5
PuTTY-2016-2563	5	5.2
Python-2007-4965	12	10.5
Python-28322	18	17.5
Python-31530	6	10.6

Table 3: The number of iterations and the time of REPT's offline analysis.

mer is low and has been well studied. For instance, Figure 8 in [39] shows that the performance overhead with circular buffers and the timing information is below 2% for a range of applications. Furthermore, the deployment of REPT proves that its performance overhead is acceptable in practice, particularly when it is selectively turned on for a program on a user machine.

We test REPT's offline analysis on a machine running an x86-64 Windows 10 on an Intel Core i7-7700K 4.2GHZ Quad-Core CPU with 16GB RAM. In Table 3, we show the analysis time for the 14 bugs REPT can analyze. We can see that REPT finishes its analysis within 20 seconds for all the 14 bugs.

5.3 Effectiveness

To evaluate the effectiveness of REPT, we check if reverse debugging based on recovered data can be used to effectively diagnose a bug. To make this check objective, we say REPT is effective if the values of variables that are involved in the bug fix are correctly recovered. For all the 16 bugs listed in Table 1, REPT is effective for 14 bugs. REPT does not work for Chrome-776677 because the collected trace contains in-place code update for jitted code, which fails Intel PT trace parsing. REPT does not work for LibreOffice-88914, because this is a deadlock bug that triggers an infinite loop, which easily fills up the circular trace buffer and causes the program execution history before the loop to be lost. Out of those 14 bugs, we select three complicated ones to demonstrate the effectiveness of REPT.

Pbzip2. This is a use-after-free bug caused by a race condition. Pbzip2 is a parallel file (de)compressor based on bzip2. Specifically, it divides an input file into chunks of an equal size and spawns multiple child threads to process them in parallel. The main thread synchronizes with child threads using a mutex. Unfortunately, there is a race condition bug where the main thread may free the mutex before all child threads finish, causing the program to crash when a child thread dereferences a pointer field inside the freed mutex. With REPT, a developer can set a data breakpoint on the pointer field, and locate the instruction that overwrites the pointer field in the heap free operation on the main thread by going backwards along the execution.

Python-31530. This is a race condition bug in Python's implementation of its file objects. Python preloads the file content as an optimization for its file operations. To do so, Python allocates a buffer based on the given size `bufsize` and assigns it to a pointer field `f_buf` in the file object. Then, it reads the file content into the buffer, and finally updates another pointer field `f_bufend` so that it points to the end of the buffer (i.e., `f_bufend=f_buf+bufsize`). The race condition happens when two threads preload the file content simultaneously. Specifically, while a thread is reading file content into the buffer, another thread starts preloading and overwrites `f_buf` with a *smaller* buffer. Then, the original thread updates `f_bufend` based on the overwritten `f_buf` and the old `bufsize`, which makes `f_bufend` point to a location beyond the actually allocated buffer. This causes Python to crash when it attempts to read the data outside of the allocated buffer. With REPT, a developer can set data breakpoints on both `f_buf` and `f_bufend`. By going backwards along the reconstructed execution, the developer can see how the race condition bug overwrites `f_buf` and leads to an inconsistent `f_bufend`.

Chrome-784183. This is an integer overflow bug in a validation routine used for image snipping. The validation routine checks if the snipped area is within the original image. For example, given an image represented as a matrix of pixels, one can snip the image by choosing `y` rows from row `x`. The validation routine ensures `x+y` is not greater than the height of the original image. Unfortunately, the routine does not check if `x+y` overflows. Thus, the check is incorrectly passed when a large `y` causes an integer overflow. This results in the subsequent crash when Chrome attempts to access a pixel in the snipped area based on `y`. When the crash happens, the validation function has already returned and more than 500K instructions have been executed afterwards. With REPT, a developer can go back to the validation routine and single step through it to quickly pinpoint the actual arithmetic operation that overflows.

5.4 Deployment

We have received anecdotal stories from Microsoft developers in using REPT to successfully debug failures reported to WER [30]. The very first production bug that is successfully resolved with the help of REPT had been left unfixed for almost two years because developers cannot reproduce the crash locally. The failure occurs in Microsoft Edge when an exception is thrown because a function returns with an error. The bug is hard to fix because there are two possible reasons for the function to fail and it is difficult to tell the actual reason by looking at the memory dump. With the reverse debugging enabled by REPT, the developer is able to step through the function based on the reconstructed execution history and quickly find out the root cause and fix the bug. In summary, a two-year-old bug was fixed in just a few minutes thanks to REPT.

6 Discussion

In this section, we discuss the limitations of REPT and how we plan to address them in future work.

When developers use REPT in practice, they currently have to deal with two main limitations. First, the control flow trace may not be long enough to capture the defect (e.g., the free call is not in the trace for a use-after-free bug). Second, data values that are necessary for debugging the failure are not recovered (e.g., the heap address passed to the free call is not recovered for a use-after-free bug). We cannot simply use a large circular trace buffer to solve this problem because the data recovery accuracy decreases when the trace size increases.

REPT currently does not capture any data during a program's execution. To fundamentally solve these two limitations, we will need to log more data than just the memory dump. It is an open research question to identify a good trade-off between online data logging, runtime overhead, and offline data recovery. A potential direction is to leverage the new `PTWRITE` instruction [36] to log data that is important for REPT's data recovery.

The current implementation of REPT only supports reverse debugging of user-mode executions. While REPT's core analysis is on machine instructions and thus independent of the privilege mode, we need to properly handle kernel-specific artifacts such as interrupts to support reverse debugging of kernel-mode executions.

In addition to reverse debugging, we believe one can leverage the execution history recovered by REPT to perform automatic root cause analysis. The challenge is that the data recovery of REPT is not perfect, so the research question is how to perform automatic root cause analysis based on the imperfect information provided by REPT.

Our evaluation of REPT has been focused on software running on a single machine. When developers debug distributed systems, they usually rely on event logging. It is an interesting research direction to study how program tracing can be combined with event logging to help developers debug bugs in distributed systems. We have not been able to apply REPT to mobile applications because there is no efficient hardware tracing like Intel PT available on mobile devices.

7 Related Work

There is a large body of related work dedicated to debugging failures. More recently, there have been increasing interest in debugging failures in deployed systems. In this section, we discuss some representative examples and describe how REPT differs.

Automatic Root Cause Diagnosis Techniques. A large body of automated root cause diagnosis techniques rely on statistical techniques such as sampling and outlier detection to isolate the key reasons behind a failure and thus help debugging. Cooperative bug isolation [19, 20, 37, 41], failure sketching [40], and lazy diagnosis [39] are state-of-the-art techniques. Unlike these techniques, REPT does not target at a subset of potential bugs or rely on statistical methods to isolate failure causes, but it rather focuses on reconstructing executions. We perceive these techniques as orthogonal and complementary to REPT.

POMP [57] is an automatic root cause analysis tool based on a control flow trace and a memory dump. It handles missing memory writes by running hypothesis tests *recursively*, which significantly limits its efficiency, because the number of hypotheses grows exponentially with the trace size. In contrast, REPT uses a new error correction technique to do forward/backward analysis *iteratively*, which makes its analysis grow linearly with the trace size. We compare their performance on 3 of the 14 bugs (Nasm-2004-1287, PuTTY-2016-2563, and Python-2007-4965) that are evaluated by both. REPT is 1 to 3 orders of magnitude faster than POMP. For instance, POMP takes 30 minutes to analyze the PuTTY-2016-2563 bug, but REPT only takes 5.2 seconds. POMP is evaluated only on how well it works for root cause analysis. There is no instruction-level accuracy reported in the paper, so we cannot directly compare its accuracy with REPT. Furthermore, POMP only supports a single thread, but REPT handles concurrency.

ProRace [62] attempts to recover data values based

on the control flow logged by Intel PT and the register values logged by Intel Processor Event Based Sampling (PEBS) [36]. Unlike REPT, ProRace does not provide solutions for the problems of missing memory writes and concurrent memory writes.

PRES [51] and HOLMES [24] record execution information (e.g., path profiles, function call traces, etc.) to help debug failures. PRES performs state space exploration using the recorded information to reproduce bugs. HOLMES performs bug diagnosis purely based on control flow traces. REPT relies on the lightweight hardware control flow tracing to reconstruct data flows from a memory dump.

"Better Bug Reporting" [23] is a system that performs symbolic execution on a full execution trace to generate a new input that can lead to the same failure. Reporting the generated input instead of the original input can provide better privacy. The main limitation is that it usually introduces high overhead to record a full execution trace. Furthermore, by using a full trace, this bug reporting scheme does not need to handle memory aliasing, but this is not the case for REPT.

Execution Synthesis (ESD) [60] does not assume there is any execution trace. Given a coredump, it relies on heuristics to explore possible paths to search for inputs that may lead to the crash. As recognized in the ESD paper, due to the limitations of symbolic executions for solving complex constraints, ESD may not be able to scale to large programs with long executions.

Delta debugging [61] iteratively isolates program inputs and the control flow of failing executions by repeatedly reproducing the failing and successful runs, and altering variable values. REPT does not make the assumption that failures can be reproduced and operates on a single control flow trace and memory dump.

PSE [42] is a static analysis tool that performs backward slicing and alias analysis on source code to identify potential sources of a NULL pointer. PSE has false positives and is not evaluated on real-world crashes.

Record/Replay Techniques. As we discussed earlier, certain techniques rely on full system record/replay [47–49,56] to help debug failures. REPT does not rely on full system record/replay, which is expensive for deployment usage, but rather reconstructs executions by leveraging lightweight control flow tracing.

Castor [43] is a recent record/replay system that relies on commodity hardware support as well as instrumentation to enable low-overhead recording. Castor works efficiently for programs without data races. In our experience, many programs have data races in practice, which actually make debugging very hard. REPT handles systems with data races.

Ochiai [16] and Tarantula [38] record failing and successful executions and replay them to isolate root causes. REPT does not rely on expensive record/replay techniques nor does it assume bugs can be reproduced.

H3 [35] uses a control flow trace to reduce the constraint complexity for finding a schedule of shared data accesses that can reproduce a failure. H3 does not recover data values, and only applies constraint solving to a small number of shared variables.

State-of-the-Art Techniques in Deployed Systems. Despite extensive prior research, to our knowledge, there are few examples of debugging techniques that are actively used in deployed systems. RETracer [27] is a bug triaging tool that was deployed in Windows Error Reporting [30]. RETracer assigns "blame" to a function for modifying a pointer that ultimately causes an access violation. RETracer performs backward taint analysis based on an approximate execution history recovered by reverse execution. RETracer does not require a control flow trace but can only recover limited data values.

8 Conclusion

We have presented REPT, a practical solution for reverse debugging of software failures in deployed systems. REPT can accurately and efficiently recover data values based on a control flow trace and a memory dump by performing forward and backward execution iteratively with error correction. We implement and deploy REPT into the ecosystem of Microsoft Windows for program tracing, failure reporting, and debugging. Our experiments show that REPT can recover data values with high accuracy in just seconds, and its reverse debugging is effective for diagnosing 14 out of 16 bugs. Given REPT, we hope one day developers will refuse to debug failures without reverse debugging.

9 Acknowledgments

We thank our shepherd, Xi Wang, and other reviewers for their insightful feedback. We are very grateful for all the help from our colleagues on the Microsoft Windows team. In particular, Alan Auerbach, Peter Gilson, Khom Kaowthumrong, Graham McIntyre, Timothy Misiak, Jordi Mola, Prashant Ratanchandani, and Pedro Teixeira provided tremendous help and valuable perspectives throughout the project. We also thank Beeman Strong from Intel for answering numerous questions about Intel Processor Trace.

References

[1] https://bz.apache.org/bugzilla/show_bug.cgi?id=24483.

[2] https://bz.apache.org/bugzilla/show_bug.cgi?id=39722.

[3] https://bz.apache.org/bugzilla/show_bug.cgi?id=60324.

[4] https://www.exploit-db.com/exploits/25005/.

[5] http://ifsec.blogspot.com/2007/04/php-521-wbmp-file-handling-integer.html.

[6] https://www.exploit-db.com/exploits/17201/.

[7] https://bugs.php.net/bug.php?id=74194.

[8] https://bugs.php.net/bug.php?id=76041.

[9] https://github.com/tintinweb/pub/tree/master/pocs/cve-2016-2563.

[10] https://bugs.python.org/issue1179.

[11] https://bugs.python.org/issue28322.

[12] https://bugs.chromium.org/p/chromium/issues/detail?id=784183.

[13] https://bugs.python.org/issue31530.

[14] https://bugs.chromium.org/p/chromium/issues/detail?id=776677.

[15] https://bugs.documentfoundation.org/show_bug.cgi?id=88914.

[16] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim Intl. Symp. on Dependable Computing*, 2006.

[17] Apple Inc. MacOSX CrashReporter. https://developer.apple.com/library/content/technotes/tn2004/tn2123.html, 2017.

[18] Arm Embedded Trace Macrocell (ETM), 2017. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html.

[19] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[20] J. Arulraj, G. Jin, and S. Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[21] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7), July 2011.

[22] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation*, 2008.

[23] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[24] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Intl. Conf. on Software Engineering*, 2009.

[25] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with revgen. In *Proceedings of the 7th Workshop on Hot Topics in System Dependability*, 2011.

[26] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. *SIGOPS Oper. Syst. Rev.*, 2010.

[27] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. RETracer: Triaging crashes by reverse execution from partial memory dumps. In *International Conference on Software Engineering*, 2016.

[28] J. Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, Vienna, Austria, 2012.

[29] J. Gilchrist. Parallel BZIP2. http://compression.ca/pbzip2, 2017.

[30] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *ACM Symp. on Operating Systems Principles*, 2009.

[31] GNU Foundation. GDB and reverse debugging. https://www.gnu.org/software/gdb/news/reversible.html, 2018.

[32] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *Intl. Conf. on Computer Aided Verification*, 2008.

[33] Google Inc. Chrome Error and Crash Reporting. https://support.google.com/chrome/answer/96817?hl=enl, 2017.

[34] M. D. Hill and M. Xu. Racey: A stress test for deterministic execution. http://www.cs.wisc.edu/~markhill/racey.html.

[35] S. Huang, B. Cai, and J. Huang. Towards production-run heisenbugs reproduction on commercial hardware. In *Proceedings of the 2017 USENIX Annual Technical Conference*, Santa Clara, CA, 2017. USENIX Association.

[36] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, 2017.

[37] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[38] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[39] B. Kasikci, W. Cui, X. Ge, and B. Niu. Lazy diagnosis of in-production concurrency bugs. In *ACM Symp. on Operating Systems Principles*, Shanghai, China, October 2017.

[40] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *ACM Symp. on Operating Systems Principles*, 2015.

[41] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[42] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM International Symposium on Foundations of Software Engineering*, 2004.

[43] A. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieŕes, and M. Rosenblum. Towards practical default-on multi-core record/replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2017.

[44] Microsoft Corporation. Time travel debugging. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview.

[45] Microsoft Corporation. Windows Debugger. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/.

[46] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Intl. Symp. on Computer Architecture*, 2008.

[47] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[48] Mozilla Corporation. Mozilla rr. http://rr-project.org/, 2017.

[49] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Intl. Symp. on Computer Architecture*, 2005.

[50] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 2009.

[51] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. PRES: Probabilistic replay with execution sketching on multiprocessors. In *ACM Symp. on Operating Systems Principles*, 2009.

[52] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *IEEE/ACM International Symposium on Microarchitecture*, 2011.

[53] C. Rossi. Rapid release at massive scale. https://code.facebook.com/posts/270314900139291/rapid-release-at-massive-scale/, 2015.

[54] Ubuntu. Ubuntu error. https://wiki.ubuntu.com/ErrorTracker, 2017.

[55] Undo. UndoDB: The interactive reverse debugger for C/C++ on Linux and Android. https://undo.io/, 2018.

[56] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[57] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium*, Vancouver, BC, 2017. USENIX Association.

[58] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[59] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ACM SIGSOFT European Conference on Foundations of Software Engineering*, 2011.

[60] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM European Conf. on Computer Systems*, 2010.

[61] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2002.

[62] T. Zhang, C. Jung, and D. Lee. ProRace: Practical data race detection for production use. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.