RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Sang Kil Cha
KAIST
sangkilc@kaist.ac.kr

Yanick Fratantonio
UC Santa Barbara
yanick@cs.ucsb.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

ABSTRACT

Many software providers operate crash reporting services to automatically collect crashes from millions of customers and file bug reports. Precisely triaging crashes is necessary and important for software providers because the millions of crashes that may be reported every day are critical in identifying high impact bugs. However, the triaging accuracy of existing systems is limited, as they rely only on the syntactic information of the stack trace at the moment of a crash without analyzing program semantics.

In this paper, we present RETracer, the first system to triage software crashes based on program semantics reconstructed from memory dumps. RETracer was designed to meet the requirements of large-scale crash reporting services. RETracer performs binary-level backward taint analysis *without* a recorded execution trace to understand how functions on the stack contribute to the crash. The main challenge is that the machine state at an earlier time cannot be recovered completely from a memory dump, since most instructions are information destroying.

We have implemented RETracer for x86 and x86-64 native code, and compared it with the existing crash triaging tool used by Microsoft. We found that RETracer eliminates two thirds of triage errors based on a manual analysis of 140 bugs fixed in Microsoft Windows and Office. RETracer has been deployed as the main crash triaging system on Microsoft's crash reporting service.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

triaging; backward taint analysis; reverse execution

1. INTRODUCTION

Many software providers, including Adobe [2], Apple [6], Google [16], Microsoft [15], Mozilla [28] and Ubuntu [41], operate crash reporting services that automatically collect crashes from millions of customers and file bug reports based on them. Such services are critical for software providers because they allow them to quickly identify bugs with high customer impact, to file bugs against the right software developers, and to validate their fixes. Recently, Apple added crash reporting for apps in the iOS and Mac App Stores to help app developers pinpoint top issues experienced by their users [5]. Crash reporting is not only being relied upon by traditional desktop platforms, but also by more recent mobile platforms.

One of the most critical tasks in a crash reporting service is *triaging* software crashes, i.e., grouping crashes that were likely caused by the same bug. Large services may receive millions of crash reports every day. It is not possible to have developers inspect more than a small fraction of them. Good triaging can cluster together a potentially large number of crash reports belonging to the same software bug and thus reduce the number of crash reports that have to be inspected. It can also help prioritize the most critical bugs by user impact (i.e., those bugs for which the largest numbers of crash reports arrive) and file them against the right software developers.

Precise crash triaging is a hard problem. A single bug may manifest itself in a variety of ways and give rise to large numbers of dissimilar-looking crash reports. This difficulty is exacerbated by the limited information contained in a crash report. A typical report (or core dump or memory dump [45]) contains at most the context of the crash thread (stack and processor registers) and a subset of the machine's memory contents *at the moment of the crash*. A variety of constraints ranging from minimizing overhead on customer machines to protecting customer privacy prevent software providers from including richer information in their reports. In particular, there is no information about the program execution leading up to the crash. The sheer volume faced by large-scale crash reporting services adds a further problem because there is usually a time budget for analyzing a crash.

Given these challenges, crash reporting services have been limited to triaging crashes by using stack traces in a "syntactic" way. Ubuntu [40] groups crashes by using a crash signature computed based on the top five functions on the stack of the crash thread (we will refer to it as "the stack" in the rest of this paper unless specified otherwise). Microsoft's Windows Error Reporting (WER) service [15] uses a tool called !analyze (bang analyze [23]) to group crashes based on a *blamed* function identified on the stack. !analyze picks the top function as the blamed function by default, but uses a large whitelist of functions and modules and additional heuristics to pick a different function down the stack. Such triaging approaches do not consider the "semantics" of functions, that is, how they contribute to the crash. This significantly limits their triaging accuracy

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884844

and the overall effectiveness of crash reporting services.

In this paper, we present RETracer, the first system to triage software crashes based on program semantics reconstructed from a memory dump. RETracer was designed to meet the requirements of large-scale crash reporting services. Conceptually, RETracer mimics developers which usually debug crashes by *reversely* analyzing the code and the stack to find out how a bad value such as a corrupted pointer was passed. RETracer automates this debugging process by realizing *binary-level backward taint analysis* based only on memory dumps.

Backward taint analysis begins with tainted data items (e.g., a corrupted pointer) at the end of an execution and analyzes instructions reversely to propagate taint backwards and determine where the tainted data originated at an earlier time. It is straightforward to perform this analysis on a recorded execution trace [14, 8, 9]. However, recording execution traces is not an option for crash triaging services because it imposes significant tracing overhead during normal execution [8].

Backward taint analysis *without* an execution trace is challenging. The machine state at an earlier time cannot be recovered completely since most instructions are *information destroying*. For example, after executing `XOR EAX, EAX`, the original value of register `EAX` is lost. With an incomplete machine state, backward taint propagation may have to stop prematurely when the address of a tainted memory location cannot be recovered.

RETracer is the *first* system to successfully perform binary-level backward taint analysis without an execution trace for crash triaging. To recover memory addresses, it combines concrete reverse execution and static forward analysis to track register values.

Given a backward data flow graph inferred from the backward taint analysis, it is not obvious how to find the blamed function. We design an intuitive algorithm to solve this problem. RETracer blames the function where the bad memory address was derived for the first time. The intuition behind it is that the function that was the first one to obtain a bad value should be the one to ensure the value's correctness.

We have developed RETracer into a working system for triaging crashes of x86 and x86-64 native code, and compared it with !analyze, the existing crash triaging tool used by WER. We manually analyzed crashes of 140 bugs fixed in Microsoft Windows 8 and Office 2013. RETracer correctly identified the blamed function for all but 22 bugs, while !analyze failed for 73 bugs. RETracer thus reduces the number of triage errors by two thirds. We evaluated RETracer on 3,000 randomly selected crashes and found that its median, average, and maximum run time is 1.1, 5.8, and 44.8 seconds. RETracer has been deployed as the main crash triaging system on WER. A study of the deployment data shows that RETracer is being used to analyze about half of the reported x86 and x86-64 crashes in Microsoft's software.

In summary, we make the following contributions:

- We have designed a new approach to triaging crashes that performs backward taint analysis without requiring execution traces and identifies blamed functions based on backward data flow graphs (Section 3).
- We have implemented RETracer for triaging crashes of x86 and x86-64 native code (Section 4).
- We have compared RETracer with !analyze, the existing crash triaging tool used by WER, on 140 fixed bugs and studied RETracer's deployment on WER. (Section 5).

2. OVERVIEW

In this section, we present an overview of RETracer's design and discuss its design choices.

RETracer can analyze all crashes caused by access violations, except for those due to stack overflows (i.e., out of stack space), unloaded modules and bit flips [29]. These three exceptions are not a serious limitation because there are effective solutions for each of them. For stack overflows, a straightforward solution is to blame the function that takes up the most stack space. For unloaded modules, a straightforward solution is to blame the function that calls the unloaded module. A solution for detecting bit flips was proposed in [29]. Crashes that are being analyzed by RETracer make up about half of the x86 and x86-64 crashes in Microsoft's software (see Section 5).

RETracer performs backward taint analysis to triage crashes. This essentially mimics how developers analyze crashes. The first question developers often ask is where the corrupted pointer came from. To answer this question, they usually follow the code and the stack backward to find out how the corrupted pointer was passed. The backward taint analysis in RETracer automates this debugging process.

Similar to existing triaging solutions [15, 40], RETracer focuses on the stack of the crash thread: It will blame one of the functions on the stack. A common question is what happens if a crash involves multiple threads. The answer has two aspects. First, a large fraction of bugs involves only the crash thread. For example, an empirical study [36] on the ECLIPSE project showed that more than 60% of bugs were fixed in a function on the crash stack. Second, the triaging goal of RETracer is to group crashes caused by the same bug together, which does not require finding the root cause. For example, assume function A in thread 1 writes a bad value to a pointer. Function B in thread 2 reads the bad pointer and dereferences it, causing an access violation. In this case, the root cause is function A. But if function B is the only one or one of a few functions that use the bad pointer, then grouping crashes based on function B meets the triaging goal.

Backward data flow analysis can, in principle, be performed on source code (e.g., [22, 37]) or on binaries. Source code analysis has a number of desirable properties, including being unencumbered by the complications of processor instruction sets. However, our only source of information about crashes are crash reports. The data they contain (CPU register values, values stored at raw virtual addresses) can be consumed directly by binary-level analysis. While one could consider translating this information into a form that is amenable to source-level analysis, such translation would have to overcome a variety of complications. Registers and memory locations in the crash report would have to be mapped to variables in the source code. The challenge is that there is no debug information for *temporary* variables. For example, given `VarB = VarA->FieldB->FieldC`, there is no debug information indicating in which register or stack location the value of `VarA->FieldB` is stored. Furthermore, compiler optimizations may make the mapping from machine state to source code even harder. Given these complications, we made the design choice to perform our analysis at the binary level.

This choice has the additional benefit of making RETracer independent of the original programming language. This allows us to use a single analysis engine to analyze crashes of both native code (e.g., compiled from C/C++) and jitted code (e.g., compiled from C# or JavaScript). In this paper, we focus on analyzing crashes of native code.

Taint analysis assigns meta-data (taint) to data items (registers or memory locations). Given a sequence of instructions, and an initial set of tainted data items, taint analysis will propagate any taint as-

```
1  f()
2  {
3      T p;
4      p.f = NULL; // RETracer blames this function!
5      g(&p);
6  }
7
8  g(T *q)
9  {
10     int *t = q->f;
11     h(t);
12 }
13
14 h(int *r)
15 {
16     *r = 0; // crash!!
17 }
```

Listing 1: This sample code demonstrates how RETracer makes use of backward taint analysis to blame functions. The crash occurs in function `h` (line 16). RETracer blames function `f` because it originally sets the bad value to pointer `r` (line 4).

sociated with the source operands of an instruction to its destination operand. Backward taint analysis begins with tainted data items at the end of an execution and tries to determine where the tainted data originated at an earlier time. It analyzes instructions reversely to propagate the taint backward. This technique has been applied to recorded execution traces [9], where the complete sequence of executed instructions and a large amount of state that can be derived from it are available.

RETracer is the first system to perform binary-level backward taint analysis without execution traces. The input to RETracer is the memory dump of a crash, binaries of modules identified in the memory dump, and their debug symbols. On Windows, a binary is a PE file, and its debug symbols are stored in a corresponding PDB file. RETracer does not require the memory dump to be complete. Instead, it only requires the stack memory and the CPU context of the crash thread at the moment of crash. Additional memory information in the dump may improve its analysis but is not necessary.

RETracer begins with the corrupted pointer that caused the crash and uses backward taint analysis to find the program location(s) where the bad value originated. The limited amount of information in a memory dump (when compared to an execution trace) is a significant complication that RETracer must overcome. The result of the backward taint analysis is a backward data flow graph that shows how the corrupted pointer was derived. RETracer analyzes the graph to identify the blamed function where the corrupted pointer was derived for the first time.

RETracer's backward taint analysis uses concrete addresses rather than symbolic expressions to represent memory locations. The main limitation of using concrete addresses is that taint cannot be propagated if the address of a tainted memory location cannot be recovered. The main limitation of using symbolic expressions is that taint will be propagated too far if the alias set of the symbolic expression of a tainted memory location is an over-approximation. In RETracer we choose concrete addresses over symbolic expressions because we would rather blame a function close to the crash than a totally irrelevant function. To recover concrete addresses of memory locations, RETracer combines concrete reverse execution with forward static analysis to track values of registers and memory locations.

A simple example is shown in Listing 1. In this case, the crash occurs in function `h` at line 16. With its backward taint analysis, RETracer can find out that the bad pointer `r` was originally set in function `f` at line 4. Therefore, RETracer blames function `f` for this crash.

3. DESIGN

In this section, we describe the design of RETracer in detail. First, we will present the basic scheme of the backward taint analysis. Then we will describe how we use static forward analysis to mitigate the problem of missing register values caused by irreversible instructions. Finally, we will present the algorithm for identifying blamed functions from backward data flow graphs.

RETracer's analysis is performed at the binary level. We describe the design in terms of a very simple assembly language. This allows us to describe RETracer without being encumbered by the idiosyncrasies of any concrete processor. Section 4 will describe how we implemented RETracer for x86-64 processors.

A program in our language is a sequence of instructions of the form `opcode dest,src`, where `opcode` specifies the instruction type (e.g., `mov` or `xor`), and `src` and `dest` are the source and destination operands. An operand is an immediate (a constant), a processor register (numbered $R_0, R_1, \ldots$), or a memory address specified as $[R_b + c \cdot R_i + d]$, where $c \in \{0, 1, 2, 4, 8\}$ is a constant, d is a constant displacement, R_b and R_i are arbitrary registers. We refer to R_b as the *base register* and to R_i as the *index register*.

3.1 Backward Taint Analysis

For taint analysis, we need to answer two questions: how to introduce taint and how to propagate taint. We maintain taint information on both registers and concrete memory locations. For memory locations, we keep the taint at byte granularity. For registers, we keep the taint at register granularity.

Next we first describe taint introduction, taint propagation, and concrete reverse execution. We then explain how the backward analysis is performed inside a function and across functions.

3.1.1 Taint Introduction

The crash report logs both the crash instruction and the access violation type (i.e., read, write, or execution). For write violations, we examine the destination operand of the crash instruction. If it is a memory operand, we taint its base and index registers because the base register may contain a corrupted pointer value, and the index register may contain a corrupted array index. For read violations, we proceed similarly for the source operand. For execution violations, we check the caller to see if it was caused by calling a function pointer. If so, we taint the base and index registers of the memory operand for the function pointer.

3.1.2 Taint Propagation

Starting at the instruction that triggered the crash, we move backward, instruction by instruction. Section 3.1.5 will explain how we incorporate the program's control flow into this analysis. For each instruction, we propagate taint depending on the semantics of the instruction. By default, we first check if the destination operand is tainted. If so, we first untaint the destination operand and then taint the source operand. For example, the `MOV` instruction copies the value of its source operand to its destination operand and can be handled by the default rule. Section 4 contains examples of x86 instructions that require more complex tainting rules.

When propagating taint to registers, we do not require knowing their values. When propagating taint to memory locations, we must know their addresses. Even when the memory address is unknown, we always propagate taint to the base and index registers of the memory operand if they exist. By tainting these registers, we can track how the pointer to a corrupted value was derived recursively

and construct a backward data flow graph of multiple dereference levels. In Section 3.2, we describe how we use such a backward data flow graph to identify a function to blame for the crash.

3.1.3 Concrete Reverse Execution

To compute the address of a memory location, we need to know the values of the base and index registers of the memory operand. In RETracer, we perform concrete reverse execution to track values of registers and memory locations. Similar to taint tracking, we keep values at register granularity for registers and at byte granularity for memory locations.

We can think of an instruction as a function f such that $dst = f(src)$ (e.g., MOV dst, src) or such that $dst = f(src, dst)$ (e.g., ADD dst, src). In the former case, the concrete reverse execution of the instructions attempts to compute the value of $src = f^{-1}(dst)$. This succeeds if we know the value of dst and if f is reversible (i.e., if $f^{-1}(dst)$ is a single value). In the latter case, we attempt to determine the value of dst before the instruction. This is possible if we know the values of src and of dst after the instruction and if f is reversible. In either case, if we can obtain the value, we update the register or memory location associated with the operand and proceed backwards to the next instruction.

The problem is complicated by the fact that many instructions are irreversible. Examples include XOR R_0, R_0 and MOV R_0, $[R_0]$. In these cases, we set the register or memory location's value to be unknown. In cases like MOV R_0, $[R_0]$, we also cannot propagate taint to the memory location $[R_0]$ because the value of R_0 is unknown.

It is worth noting that instructions on stack and frame pointers [43] are mostly reversible. This allows RETracer to track taint on stack memory locations almost completely. On the other hand, the problem of missing register values significantly affects taint propagation to memory locations not controlled by stack or frame pointers. The next subsection describes how we use static forward analysis to mitigate this problem.

3.1.4 Static Forward Analysis

The key idea of our static forward analysis is to perform a binary analysis of individual functions to identify how register values are derived.

MOV R_0, R_1
MOV R_0, $[R_0]$

In the above two-instruction example, our forward analysis will find that, *after* executing the first instruction, we establish the *value relation* that R_0 has the same value as R_1. When we perform backward analysis on the second instruction, we can use this value relation to compute R_0's original value (assuming R_1's value is known) and propagating taint to $[R_0]$. This lookup is done recursively. For example, if we do not know R_1's value, we will check if it can be computed from another register or memory location.

This is similar to traditional use-def analysis. But there is an important difference. Our goal is to find value relations that depend on the *current* register values.

MOV R_0, R_1
MOV R_1, R_2
MOV R_0, $[R_0]$

In the above three-instruction example, after executing the second instruction, we will invalidate all the value relations based on R_1 because its value is changed. We then add a new value relation for R_1 and R_2. Thus, we will not use R_1 to recover R_0's value at the third instruction. In contrast, traditional use-def analysis would maintain that R_0 was defined by R_1 (in the first instruction).

Our static forward analysis is conservative in the sense that we invalidate value relations with all memory locations if the destination of a memory write cannot be decided. To decide the destination of a memory write to the stack, we track how the stack pointer is updated. Specifically, we use two symbolic expressions to represent the values of registers pointing to the stack. The first symbolic expression is *stack0* which represents the stack pointer's value at the entry of a function. The second symbolic expression is *stack1* which represents the stack pointer's value after stack alignment (e.g., to 8 bytes). We need the second symbolic expression because the effect of a stack alignment instruction cannot be determined statically. In our static forward analysis, the stack pointer and other registers derived from it are all represented by *stack0* or *stack1* plus an offset.

Another advantage of tracking registers based on *stack0* and *stack1* is that we can directly tell if a register in a memory operand points to the stack. If so, we do not propagate taint to this register (since RETracer does not triage stack overflows). Without this, incorrect taint could eventually propagate to the stack pointer. While this problem could be fixed, it would complicate our backward taint analysis.

3.1.5 Intra-Procedural Analysis

We have presented the basic ideas of backward taint analysis and static forward analysis. Next we describe how they are being done within a function. Given a function, we first divide it into basic blocks and construct a control flow graph.

Static forward analysis is done only once, before we perform backward taint analysis on a function. Our goal is to compute the value relations for all registers before each instruction. For each block, we first initialize the value relations by merging them from all its *preceding* blocks in the control flow graph. Then we analyze each instruction in the basic block to update the value relations as described in Section 3.1.4. We repeat this process by iterating through all blocks until the value relations converge. When we merge value relations from multiple blocks, we mark a register's value to be invalid if its value relations have a conflict. This ensures that the static forward analysis will converge.

In the concrete reverse execution, we maintain the current values of registers and memory locations for every instruction. We start from the crash instruction in its function. We use the crash report to obtain initial values of registers and memory locations. Then, we execute backward from the crash instruction in its block. For each instruction, we update the concrete values of registers and memory locations as described in Section 3.1.3. We also leverage the value relations inferred from the static forward analysis to recover register values if necessary. For each block, we first merge the values and taint from all its *succeeding* blocks in the control flow graph. If the concrete values of a register or memory location do not agree, we set its value to unknown. We repeat this process by iterating through all blocks that are backward reachable from the crash instruction until the values converge.

If the block of the crash instruction is in a loop, the analysis above will merge values from multiple iterations. This may overwrite the original values that led to the crash. To better capture these important values, we create a new block that contains the instructions up to the crash instruction in its block. We reverse execute this new block and propagate the values to its preceding blocks only once at the beginning.

In the backward taint analysis, we maintain the currently tainted registers and memory locations. We start from the crash instruction in its function. As discussed in Section 3.1.1, we set the initial taint on the base and index registers of one of the operands of the crash instruction. Then, we execute backward from the crash instruction in its block. For each instruction, we propagate the taint

as described in Section 3.1.2. For each block, we first merge the taint from all its *succeeding* blocks in the control flow graph. Since a register or memory location may be tainted at multiple places, we keep its taint as a *set* of program locations where it was tainted. When merging the taint of a register or memory location, we simply update this set. We repeat this process by iterating through all blocks that are backward reachable from the crash instruction until the taint converges. We also apply the same optimization to the block of the crash instruction as in the concrete reverse execution to capture the original taint propagation right before the crash.

After completing the analysis of the crash function, we use its values and taint at the function entry as the initial values and taint for its caller on the stack. Then we perform the backward taint analysis on the caller function from the call instruction where the caller calls the crash function. We apply the optimization described previously to the block of the call instruction to better capture the values and taint right before the function call. After we are done with this function, we continue the analysis to its caller on the stack.

3.1.6 Inter-Procedural Analysis

We have just described how we perform backward taint analysis and static forward analysis within a function. Next we present the details on how we handle function calls in these two analyses.

Our static forward analysis is an intra-procedural analysis. We do not analyze callee functions in this analysis. Instead, given a call instruction, we invalidate value relations for volatile registers which can be modified by the callee based on the calling convention [44] as well as memory locations. We also update the stack pointer if the callee is responsible for cleaning up the stack under the function's calling convention.

Our backward taint analysis is inter-procedural. In addition to analyzing the functions along the stack, as described previously, we apply backward taint analysis to functions called by them. Our inter-procedural analysis is, by necessity, incomplete because we may not be able to find the target functions for all indirect calls during reverse execution. Furthermore, the key difference between functions on the crash stack and those that are not is that the stacks of the latter functions are unavailable since they were already popped. Without the stack, it is difficult to recover lost register values when they are "popped" off the stack at the end of typical functions. This means that we have limited ability to track memory taint in functions that are not on the crash stack.

For these two reasons, we perform the backward taint analysis only for callee functions that either have a tainted return value or change the stack pointer (e.g., exception handling code). Before performing the backward taint analysis on a callee function, we run our static forward analysis on it. We begin the analysis of callee functions at their return instruction(s).

For call instructions whose targets we do not analyze (because we could not find the target or because the target does not meet our conditions), we check if its return value and parameters are tainted based on the function's definition. Since, in general, we have no specification for how a function may change its parameters, our parameter check is only an approximation. Specifically, we assume a callee function may modify the target data structure pointed to by a parameter and check if any memory covered by the data structure is tainted. If the return value or a parameter is tainted, we untaint it and mark that its value was set by the callee function.

3.2 Blamed Function Identification

In this section, we describe how we construct a backward data flow graph based on our backward taint analysis and how we identify a blamed function based on this graph.

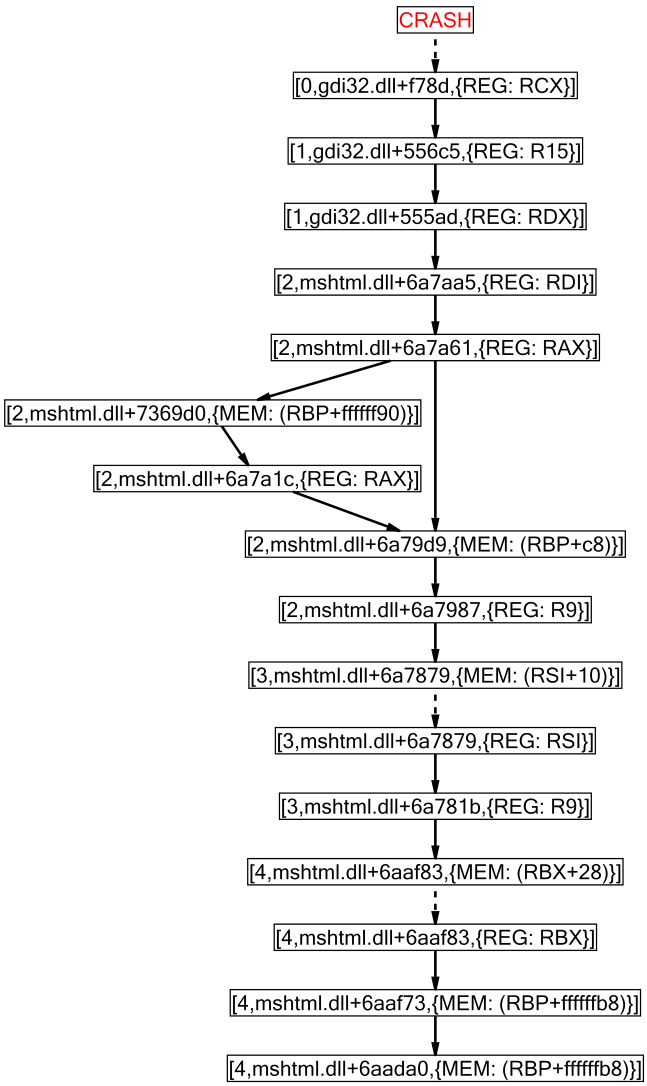

Figure 1: A sample backward data flow graph. Each node is a three tuple consisting of the frame level, the instruction's address, and the tainted operand. Assignment edges are solid and dereference edges are dotted. The graph is simplified to improve readability.

3.2.1 Backward Data Flow Graph

A node in our backward data flow graph is created when a register or memory location is tainted. A node is also created when the taint is stopped at a constant or a call to a callee function. To differentiate instances of the same registers and memory locations at different instructions or at different invocations of the same instruction, we define a node to be a three tuple consisting of the *frame level* of the function on the stack, the instruction's address, and either a register or memory location or constant or a symbol representing a call to a callee function. The crash function has the frame level of zero because it is at the top of the stack. When going down the stack, the frame level increases by one when moving by one frame. For a callee function that is not on the stack, we use the frame level of its caller function on the stack. A sample backward data flow graph is shown in Figure 1.

Our backward data flow graph has two kinds of edges: *assign-*

ment and *dereference*. An assignment edge is added when we propagate taint from a destination operand to a source operand. A dereference edge is added when we propagate taint from a memory location to its base and index registers (i.e., the base memory address and the array index). Both assignment and dereference edges are directed. For assignment edges, the direction is from the destination operand to the source operand. For dereference edges, the direction is from the memory location to the base and index registers.

We add a special node (called the *crash node*) to the graph. We also add dereference edges from the crash node to the base and/or index registers which received the initial taint. We define the *dereference level* of a node to be the minimum number of dereference edges on any path from the crash node to this node. The crash node's dereference level is 0.

3.2.2 Blamed Function Identification

Our identification algorithm works in two steps. The first step is to identify nodes that have *bad* values. The second step is to find a *blamed* function that is the source of these bad values.

For the first step, we pick the nodes of the base and index registers at the crash site to be the nodes of bad values by default. These are the nodes that are connected by dereference edges to the crash node. There is a single bad-value node [0,gdi32+f78d,{REG:RCX}] in Figure 1. We make two exceptions to this rule. One is related to array operations. The other is related to whitelisted modules.

An array operation is usually realized by a compiler as $[R_b + c \cdot R_i + d]$, and we look for operands of this type. If the crash instruction does not have an index register and its base register (i.e., the corrupted pointer) was derived from an array operation, we pick the nodes of the base and index registers of the array operation to be the nodes of bad values. The intuition behind this exception is that, if a corrupted pointer was obtained from an array access, a problem in the array access (due to a bad array base or a bad array index) is more likely than the array containing incorrect values.

We *conditionally* whitelist library modules such as ntdll.dll. On one hand, a crash in such a module is usually caused by a bug in its caller outside the module. On the other hand, simply ignoring functions from a whitelisted module is problematic because it prevents developers from detecting bugs in library modules. Such bugs often have high impact due to the wide use of library modules. Missing them may result in a large number of unresolved crashes.

We solve this dilemma with the help of the backward data flow graph. If all the nodes in the graph are from functions of whitelisted modules, the crash is likely due to a bug in a whitelisted module because the caller outside the whitelisted modules did not pass a parameter that is directly related to the crash. In this case, we handle the whitelisted modules as regular modules. Otherwise, we pick the nodes that are outside whitelisted modules and have the minimum dereference level to be the nodes of bad values. They essentially represent the tainted parameters passed to a whitelisted module.

In the second step of our algorithm, we identify a blamed function as follows. We first identify nodes that are reachable from nodes of bad values by following only assignment edges. All these nodes are at the same dereference level. Then we pick the function that contains such nodes and has the *maximum* frame level as the blamed function. The intuition behind this algorithm is that *the function that was the first one to obtain a bad value should be the one to ensure the value's correctness.*

A bug fix of a bad parameter in a call can either be done in the caller before passing it or in the callee before using it. For this type of bugs, our algorithm *always* blames the caller. We made this choice for two reasons. First, the majority of bug fixes we have observed in practice are in the caller. Second, the call site in the caller function reveals more information than the callee function. A developer can leverage the richer information to decide where the actual fix should be.

By default, we use the name of the blamed function and its module name for triaging. There is an exception. If our analysis stops at a stack frame for which we do not have symbol information, we do not have a function name. In this case, we only use the module name. This allows RETracer to correctly blame third-party modules (that passed a wrong parameter) instead of correct first-party code.

In the example shown in Figure 1, RETracer correctly blames a function in the module mshtml.dll while the crash is in a function in another module (gdi32.dll) three frames above on the stack. The bad value was first obtained from [3,mshtml.dll+6a7879,{MEM:(RSI+10)}].

4. IMPLEMENTATION

We have implemented RETracer for crashes of native x86 and x86-64 Windows binaries. RETracer consumes Windows crash dump (.dmp) files and is implemented as a WinDbg [24] extension [33]. The prototype consists of roughly 66,000 lines of C and C++ code.

RETracer does not support crashes caused by unhandled exceptions and access violations caused by stack overflows, unloaded modules and bit flips. It identifies unhandled exceptions by simply checking the exception code. It identifies stack overflows by checking if the stack pointer passes the stack limit. RETracer identifies unloaded modules by checking if a stack frame is from an unloaded module. It relies on an existing implementation of [29] to detect bit flips.

RETracer uses Windows libraries (MSDIA and MSDIS) for extracting debug information from Windows symbol (.pdb) files and for disassembling machine instructions in Windows *portable executable* (PE) files. MSDIA and MSDIS are released in Microsoft Visual Studio [25]. Given an offset in a PE file, RETracer implements its own logic based on MSDIA and MSDIS to identify the function containing that offset, find the code ranges of the function, disassemble them into instructions, and construct a control flow graph of basic blocks.

RETracer currently relies on the debug information stored in PDB files to reliably disassemble binaries. There exist good disassemblers such as IDA [18] that can disassemble a binary without the debug information based on heuristics. Although perfect disassemly may not be achievable without the debug information, RETracer can potentially work well with such disassemblers if they can correctly disassemble most binaries. Such integration is for a future exploration.

For each instruction, RETracer parses it into an intermediate representation that specifies the opcode and the source and destination operands. RETracer can parse *all* x86/x86-64 instructions. This allows RETracer to apply its default operations in its analysis to all instructions. The default operation for the backward taint analysis is to clear the taint of the destination operand and propagate it to the source operand. The default operation for the concrete reverse execution is to set the value of the destination operand to be unknown. The default operation in the forward static analysis is to invalidate all the value relations based on the destination operand.

In Figure 2, we show the x86/x86-64 instructions that individually handled by RETracer based on their special semantics. Recall that the main goal of RETracer is to recover how a corrupted pointer or array index was derived. This is why RETracer only applies the default operations to bit and floating-point instructions (e.g., SAL and FLD).

Some instructions like CMOV are conditional. We handle such

Category	Instructions
Copy	MOV, LEA, MOVSX, MOVZX, CMOV, MOVS
Arithmetic	ADD, SUB, MUL, DIV, INC, DEC, SBB
Logic	AND, OR, XOR, NEG
Stack	PUSH, POP, PUSHA, POPA, PUSHF, POPF
Exchange	XCHG, CMPXCHG
Control	CALL, RET
Misc	CLD, STD, SETcc

Figure 2: Instructions individually handled by RETracer based on their special semantics.

instructions in a way similar to branches. We assume both conditions are possible, analyze the instruction under each condition, and merge their results. In the case of CMOV, we take the following actions. For the backward taint analysis, we *keep* the taint of the destination operand and propagate the taint to the source operand if the destination operand is tainted. For the concrete reverse execution, we set the destination operand to be unknown and do not update the source operand. For the static forward analysis, we mark the destination operand's value to be invalid.

For function calls that follow the stdcall calling convention [44], we need to unwind the stack on behalf of the callee if we do not analyze it. This is critical for correctly keeping track of the stack. To unwind the stack, we need the call site information to know if we need to unwind the stack and, if so, how much we should adjust the stack pointer. However, PDB files do not contain this information for *all* indirect calls. To mitigate this problem, we developed the following heuristics to infer the stack size to unwind. First, for an indirect call to a function in a different module, we identify the module that implements the function and use that module's PDB file to retrieve the function information. Second, for an indirect call to a virtual function, we construct the virtual function table based on the debug information in the PDB file, and use binary analysis to infer which function in the virtual function table is being called. Third, if both approaches fail, we analyze the PUSH instructions right before the indirect call to infer the stack size to unwind.

Tail call [46] is a common compiler optimization to avoid adding a new stack frame to the call stack. The simple stack walk based on the frame pointers cannot recognize tail calls correctly. If tail calls are not recognized, RETracer's backward analysis will miss the important data flow of parameter passing before the jump to the called subroutine. In RETracer, we detect tail calls by checking if the callee function based on the call instruction in the caller is matched with the callee function on the stack. If not, we further check if the first callee function has a JMP instruction at the end of the function or the first and second callee functions are contiguous (even the JMP instruction is saved in this case). If so, we recognize it as a tail call and analyze the two callee functions together.

In RETracer, we currently whitelist eleven Windows modules and all the functions in the namespace std since they are statically linked into each module. This is far less than the hundreds of whitelisted modules and functions used by !analyze. More importantly, RETracer handles whitelists conditionally as described in Section 3.2.2, which allows us to catch rare bugs in the whitelisted modules.

5. EVALUATION

In this section, we evaluate RETracer. We first present the results on its accuracy. After presenting the results on its performance, we report results from its deployment on Windows Error Reporting (WER) [15] as well as two case studies.

5.1 Accuracy

5.1.1 Identified Blamed Functions

To evaluate RETracer's accuracy in identifying blamed functions, we searched for fixed bugs to use the bug fix as a ground truth. We *exhaustively* searched the bug databases of Windows 8, Windows 8.1, and Office 2013 and the database of WER to identify *all* fixed bugs with associated crash reports that meet the following conditions: (1) The crash was an access violation in native x86 or x86-64 code. (2) The crash was not caused by a stack overflow, an unloaded module or a bit flip [29]. (3) The crash report as well as the PE and PDB files for the module in which the crash occurred are available. Conditions (1) and (2) restrict the search to the domain that our prototype is designed to handle. Condition (3) accounts for the fact that RETracer needs the crash report as well as the PE and PDB files to operate.

Our search identified 140 bugs. This is only a small fraction of the total number of bugs in the databses we searched because our search only includes bugs that were filed due to crash reports and that were fixed. More importantly, WER retains crash reports only for a short period of time. PE and PDB files will also be missing if the crash occurred in a third-party module. Thus, condition (3) eliminates a very large fraction of the eligible bugs.

Figure 3 shows the evaluation results for the 140 bugs. The figure also lists the number of different modules in which these bugs were fixed. The large module count shows that these bugs cover a broad set of code bases. The total module count in Figure 3 is less than the sum of the line items due to overlap.

We manually evaluated the correctness of the blamed functions identified by RETracer and !analyze for each of the 140 bugs using the following criterion. An identified blamed function is *correct* if the bug fix modified the *function* or another member function of its *class* or its *callee* function. We take into account member functions in the same class because they are usually owned by the same developer. A bug filed on the identified blamed function would have been sent to the same developer who made the bug fix. We consider callee functions because caller functions reveal more information from which a developer can decide to fix the bug in the caller or callee.

It is worth noting that !analyze handles a wider range of software failures than RETracer. For instance, !analyze analyzes program hangs. Our comparison between RETracer and !analyze is focused on the kind of crashes supported by RETracer.

We list the number of bugs for which the identified blamed functions are either correct (by *function* or *class* or *callee*) or *wrong* in Figure 3. RETracer is significantly more accurate than !analyze. It reduces the number of triage errors by two thirds. We also evaluated the impact of our static forward analysis by running RETracer without it. This configuration made 36 triage errors, 9 (or 33%) more than full RETracer.

We examined the 22 bugs for which RETracer failed to identify a correct function. For 11 of them, the bug fixes are in functions that are not on the crash stack. Most of these 11 bug fixes corrected errors in the global program logic. For 9 of the bugs, the fixes are in functions below the identified function on the stack (i.e., higher frame levels). Higher dereference levels and loss of taint were the main reasons for RETracer's failure in these cases. For the remaining two bugs, the fixes are in functions above the function blamed by RETracer. In both cases, a pointer check was added right before the bad pointer dereference.

In Figure 4, we show the histogram of the frame levels of functions blamed by !analyze and RETracer. The function at the top of a stack has frame level 0. We can see that !analyze blames the crash

Software	#Bugs	#Modules	#Bugs (!analyze)				#Bugs (RETracer)			
			Function	Class	Callee	Wrong	Function	Class	Callee	Wrong
Windows 8 (user mode)	52	33	19	8	0	25	30	14	1	7
Windows 8.1 (user mode)	47	33	19	1	0	27	33	5	3	6
Office 2013	21	10	8	2	0	11	13	3	1	4
Windows 8 (kernel mode)	10	9	4	0	0	6	7	0	0	3
Windows 8.1 (kernel mode)	10	9	5	1	0	4	6	2	0	2
Total	140	83	55	12	0	73	89	24	5	22

Figure 3: Accuracy based on fixed bugs in different code bases. The counts under *Class* do not include bugs that are counted under *Function* or *Callee*.

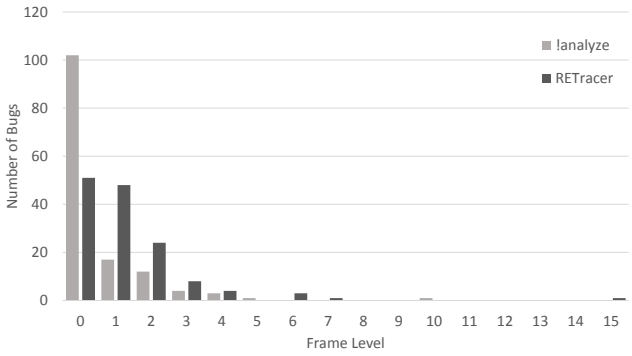

Figure 4: Histogram of the frame levels of blamed functions.

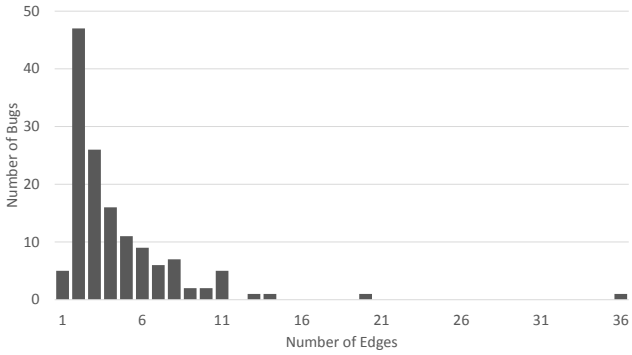

Figure 5: Histogram of the shortest distance from the crash node to the blamed function in the backward data flow graph.

function (which is at the top of the stack) for 100 out of the 140 bugs we studied. On the other hand, RETracer blames a function that is down the stack for 88 bugs. For one bug, !analyze blames the 11th function (with frame level 10) on the stack because it whitelists ntdll.dll. This function is wrong. In contrast, RETracer correctly identifies the 8th function on the stack.

For three bugs, RETracer blames the 7th function on the stack. It is correct in all three cases. RETracer blames the 16th function on the stack for one bug. While this is correct in the sense that the corrupted pointer was returned from a call in the 16th function, it is not correct by our criterion as the developer chose to add a check right before dereferencing the potentially corrupted value.

5.1.2 Backward Taint Analysis

Next we evaluate how well RETracer can recover concrete memory addresses in its backward taint analysis.

In Figure 5, we show a histogram of the shortest distance from the crash node to the blamed function in the backward data flow graph. The shortest distance is measured by the minimum number of edges from the crash node to any node in the blamed function.

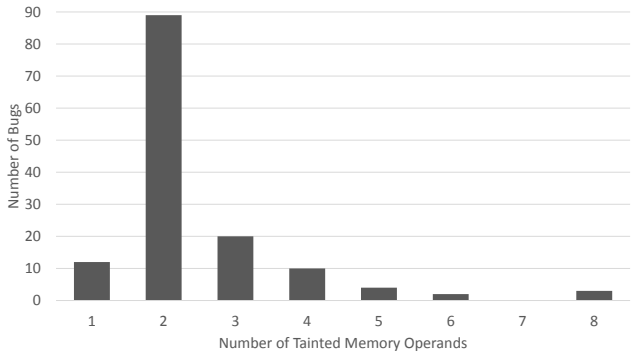

Figure 6: Histogram of the number of memory operands in the shortest path from the crash node to the blamed function in the backward data flow graph.

operand type	count
constant	7
global variable	16
dynamic memory location	78
call site	39

Figure 7: Operand types of the final nodes of 140 taint paths.

We can see that, for the majority of the 140 bugs, RETracer needs to track the taint in less than 11 edges. We checked the two bugs for which RETracer tracks the taint in 20 and 36 edges. !analyze blames the wrong function for the first bug and the correct function for the second bug after ignoring four functions based on its whitelist. For both bugs, RETracer blames the correct function. Interestingly, for the second bug, RETracer reversely traverses 34 of the 36 edges in whitelisted modules to find that the corrupted pointer was passed from a module not in the whitelist.

In Figure 6, we show the histogram of the number of memory operands in the shortest path from the crash node to the blamed function in the backward data flow graph. Since the crash node itself is a memory operand, the number of memory operands is at least one. For the majority of the 140 bugs, RETracer needs to propagate taint over at most four memory operands. We checked the two bugs for which RETracer propagates taint over eight memory operands. For both of them, !analyze is wrong and RETracer is correct.

Next, we examine the taint paths from crash nodes to blamed functions and attempt to analyze why RETracer did not extend the paths beyond the blamed functions. The taint propagation paths may end at a constant, a global variable, a dynamic memory location, or a call site. The last one means that the tainted value was returned from a callee function that RETracer did not analyze.

Figure 7 displays counts for the operand types of the final nodes for the 140 bugs. For the seven bugs with constant operands, RETracer has found the actual end of the path. RETracer found the concrete memory address for 68 of the 78 bugs associated with dynamic

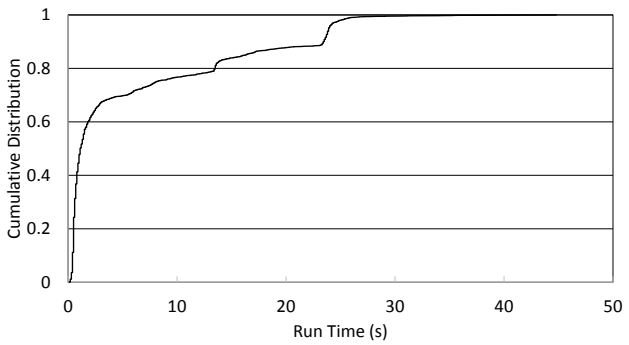

Figure 8: The cumulative distribution of RETracer's run times over 3,000 randomly picked crashes.

memory locations. For these 68 bugs and the 16 global variable bugs, the taint path ended because RETracer did not observe any further write operations to the memory location. This can happen if the memory location was written to by another thread or in a function that completed on the crash thread before the crash and that RETracer did not analyze. The latter is also the explanation for the 39 bugs associated with call sites.

We examined the ten bugs for which RETracer stopped the backward taint propagation at a dynamic memory location without finding a concrete memory address. The values of the registers for computing the memory addresses were missing mainly for one of three reasons: branches (a register may have different values for different branches), self dereference (e.g., MOV EAX,[EAX]), and incomplete memory information (a register's value is from a memory location whose value is not stored in the crash report).

5.2 Performance

To evaluate the performance of RETracer, we randomly picked 3,000 crashes reported to WER in one day. Our experiments were performed on an HP Z420 workstation with a quad-core CPU at 3.6GHz, 16GB RAM, and an OCZ-Vector150 480GB SSD.

We show the distribution of run times over the 3,000 crashes in Figure 8. The median, average, and maximum run time is 1.1 seconds, 5.8 seconds, and 44.8 seconds. The run time is at most 13 seconds for 80% crashes, and at most 25 seconds for 99% crashes.

5.3 Deployment on WER

RETracer has been deployed on WER as the main crash triaging solution since February 2015. If RETracer can analyze a crash and return a blamed function, WER uses it for triaging. Otherwise, WER uses the output from !analyze for triaging.

For a sample period of time, WER received a total of 590,282 x86 and x86-64 crash reports in Microsoft's software (i.e., the PE and PDB files for the crash module exist). 177,325 (30%) crash reports were for an unhandled exception that is not access violation. RETracer returned a blamed function for 285,288 (48%) crash reports. The rest of them were due to stack overflows, unloaded modules or bit flips. For the crashes for which RETracer found a blamed function, the function differed from !analyze's function in 184,756 cases (65%).

Next we present two case studies from RETracer's deployment.

5.3.1 SRWLock

With RETracer, we found that a top bug in x86-64 Internet Explorer is in the function RtlpWakeSRWLock in the module ntdll.dll, the core user-level module in Windows. This bug was ignored previously because ntdll.dll was in the whitelist of !analyze and other modules on the crash stack were blamed for the crashes. Though ntdll.dll is in RETracer's whitelist, RETracer still blames it because the nodes in the backward data flow graph were all from ntdll.dll.

After some investigation, we found that the root cause of the crashes was due to a wide-spread malware [1] not the code of SRWLock [47]. When the malware tries to unload itself, it modifies the return address on the stack to avoid the return to itself after its module win64cert.dll is unloaded. When manipulating the stack, the malware misaligns the stack by mistake. This causes the data structure allocated on the stack by SRWLock to *not* be aligned at the 16-byte boundary. This may lead to a crash non-deterministically because SRWLock requires the data structure to be 16-byte aligned.

The crash may happen on any thread that was using SRWLock when the malware was trying to unload itself because SRWLock uses a linked list to connect its data structures allocated on stacks. Therefore, before RETracer, millions of crashes caused by this malware were wrongly scattered into a large number of groups and were treated as unsolvable. With RETracer, we were able to put all the crashes into one group correctly and revealed it as a top bug in x86-64 Internet Explorer. More importantly, the focus on SRWLock led us to check stack alignment of all threads in the crash process and discover the root cause.

5.3.2 NTFS

With RETracer, we found a high impact bug in ntfs.sys in Windows 10 Preview. The root cause of the bug is that a pointer field used in the crash function may be null and should be checked before being used for dereference. The driver ntfs.sys is the key file system driver in Windows. This bug affected a large number of Windows 10's beta users. The bug was previously ignored by !analyze because the NTFS driver is in its whitelist. Instead, !analyze scattered the crashes into multiple groups by blaming the third-party drivers that called the NTFS driver.

The NTFS driver is also in RETracer's whitelist, but RETracer treated it as a regular module in this case because the backward data flow was contained in the NTFS driver. This allowed RETracer to correctly put all the crashes into a single group under the crash function in NTFS. This raised the priority of the bug and led to the proper fix.

6. THREATS TO VALIDITY

An *external threat to validity* is that our implementation and evaluation were focused on the Windows platform running on x86 and x86-64 architectures. However, our design was general to operating systems and machine instruction sets. So we expect RETracer to work on other architectures such as ARM and other operating systems such as Linux with reasonable engineering efforts.

A *construct validity threat* is that we define our blamed function to be *correct* if it was modified or it was in the same class as a function that was modified in a bug fix. This definition may not truly reflect the actual correctness. The ultimate criteria should be determined by a developer that a blamed function is correct if it helps isolate and fix the bug by grouping related crashes based on the blamed function. However, it is hard to find the relevant developers to verify each bug. Therefore, we used bug fixes as an objective approximation.

An *internal threat to validity* is that we evaluated RETracer on 140 *fixed* bugs. This data set may not be representative. It is possible that we may have different results on open bugs. We made this choice mainly because we need the source code fix as the ground truth. To mitigate this threat, we did an exhaustive search to find all bugs that we can experiment with.

7. RELATED WORK

There is a rich collection of literature on software debugging. In this section, we relate RETracer's design to previous approaches in software debugging, including failure classification, fault localization, and failure reproduction.

7.1 Automated Failure Classification

The goal of failure classification is to group failure reports of the same bug together. This is important for large-scale crash reporting services, including Microsoft [15], Apple [6], Google [16], Adobe [2], Ubuntu [41], Mozilla [28], to prioritize bugs and validate their fixes. Previous work has considered three different types of information that may be included in a failure report: memory dumps, execution logs and failure descriptions. We group our discussion of previous work by these categories.

Memory Dump. In practice, memory dumps are the main source of information used in failure classification. In contrast to execution logs, they do not entail overhead during regular execution. The main disadvantage of memory dumps lies in the limited information they contain. This line of work has focused on using the crash stack to group failures because it is easy to collect and useful for debugging [36].

We have discussed the approaches used in WER [15] and in Ubuntu's crash reporting service [41, 40]. Molnar *et al.* [27] proposed another kind of stack hash for aggregating crashes. The main limitation of stack hashes is that a fixed stack level does not work well with all kinds of crashes. The choice of the crash function of !analyze can be treated as a stack hash of a single stack frame.

Wu *et al.* [48] proposed to leverage information retrieval techniques to identify blamed functions for a crash. Another line of work [13, 21, 26] proposed to cluster crash reports based on the similarity of crash stacks. The main limitation of these approaches is that they require a training set and thus are not effective for newly added functions. Furthermore, they approximate a function's relevance based on syntactic, indirect measures such as a function's depth on the crash stack, which may not correctly reflect the function's involvement in a crash.

The main advantage of RETracer over previous work is that it triages crashes based on an approximated execution history (the backward data flow graph) without requiring execution logs. This enables it to achieve better triaging accuracy by leveraging function semantics as if an execution log were available.

Execution Log. This line of work [30, 39, 11] leverages recorded execution logs and/or checkpoints to classify failures. Runtime recording provides much more information than memory dumps, but also introduces unacceptably high overhead on end-user machines during normal operation.

Failure Description. This line of work [4, 19] treats failure classification as a text categorization problem. It relies on the bug description manually written by various users to classify bugs. However, in most realistic end-user scenarios, such descriptions are not available or not meaningful. In contrast, RETracer works directly on memory dumps that are automatically generated.

7.2 Software Fault Localization

Software fault localization is a technique that locates one or more possible root causes of a bug. The main difference between software fault localization and failure classification is that the former focuses on finding the exact root causes while the latter focuses on grouping failures. There has been a large volume of research on software fault localization [52, 12, 32, 17, 31, 35]. Next we summarize previous work that is most relevant to RETracer in terms of backward analysis.

Postmortem Symbolic Execution (PSE) [22] uses static backward data flow analysis to reconstruct blamed execution traces from a failure site. PSE was the first to use typestate [38] to analyze program failures. Thin slicing [37] is another backward analysis technique that is related to RETracer's design. Unlike traditional slicing [42, 3] that requires all program statements for reaching a seed statement, thin slicing consists only of producer statements for the seed, i.e., those statements that help compute and copy a value to the seed.

Unlike RETracer, these approaches do not analyze crash reports, but try to find bugs by means of static analysis. They are not suitable for triaging crashes in a large-scale crash reporting service.

7.3 Failure Reproduction

Reproducing failures is important for software debugging. Existing failure reproduction techniques [20, 50, 34, 7, 10] cannot be directly used for failure classification on practical services because of path explosion and their overhead on normal execution.

Zamfir *et al.* [51] proposed reverse execution synthesis to address the limitation of path explosion faced by execution synthesis [50]. Instead of finding an input, reverse execution synthesis aims at constructing intermediate memory state near the failure site that can lead to the failure. It does not do instruction-level reverse execution as RETracer. Instead, it analyzes basic blocks backwards and uses forward symbolic execution to analyze each basic block. Evaluations of reverse execution synthesis on three small synthetic concurrency bugs are reported in [49]. It is not clear if it can scale to large programs for crash analysis and triaging.

8. CONCLUSION

We have presented RETracer, the first system that triages crashes based on program semantics extracted from a memory dump. Without requiring an execution trace, RETracer performs backward taint analysis to find out how a bad value such as a corrupted pointer was derived and use it to identify a blamed function for triaging. It combines concrete reverse execution with static forward analysis to track concrete values of registers and memory locations for taint propagation. We have implemented RETracer and deployed it on Windows Error Reporting. Our experiments show that RETracer reduces triage errors by two thirds and triages crashes in just six seconds on average.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. We are grateful to Graham McIntyre for his tremendous help and support in implementing and evaluating RETracer and deploying it into production. We thank Galen Hunt for introducing us to the triaging problem and giving us selfless help throughout the project. We thank Gloria Mainar-Ruiz for helping us collect bug information. We thank Reuben Olinsky for answering numerous questions we had during development. We thank Andrew Baumann and Barry Bond for their help with the case studies. We thank the !analyze team for helping us with the integration and deployment.

10. REFERENCES

[1] 411-spyware.com. Win64cert.dll. http://www.411-spyware.com/file-win64cert-dll.

[2] Adobe Systems Inc. Adobe crash reporter. https://helpx.adobe.com/creative-suite/kb/changing-settings-crash-reporter.html.

[3] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In

Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE), pages 143–151, 1995.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 361–370, 2006.

[5] Apple Inc. Crash logs in xcode 6.3 beta 2. https://developer.apple.com/news/?id=02232015d.

[6] Apple Inc. Technical note TN2123: CrashReporter. https://developer.apple.com/library/mac/technotes/tn2004/tn2123.html.

[7] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP)*, pages 542–565, 2008.

[8] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the Second International Conference on Virtual Execution Environments (VEE)*, pages 154–163, 2006.

[9] BSDaemon. Dynamic program analysis and software exploitation: From the crash to the exploit code. http://phrack.org/issues/67/10.html.

[10] Y. Cao, H. Zhang, and S. Ding. SymCrash: Selective recording for reproducing crashes. In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE)*, pages 791–802, 2014.

[11] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208, 2013.

[12] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 342–351, 2005.

[13] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[14] GDB. Reverse debugging. http://www.gnu.org/software/gdb/news/reversible.html.

[15] 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 *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 103–116, 2009.

[16] Google Inc. Chrome: Send usage statistics and crash reports. https://support.google.com/chrome/answer/96817?hl=en.

[17] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 291–301, 2002.

[18] Hex-Rays. IDA. https://www.hex-rays.com/products/ida/index.shtml.

[19] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Proceedings of the 38th IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 52–61, 2008.

[20] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[21] S. Kim, T. Zimmermann, and N. Nagappan. Crash Graphs: An aggreated view of multiple crashes to improve crash triage. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN)*, 2011.

[22] 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 SIGSOFT 12th International Symposium on Foundations of Software Engineering (FSE)*, pages 63–72, 2004.

[23] Microsoft. The !analyze extension. https://msdn.microsoft.com/en-us/library/windows/hardware/ff562112(v=vs.85).aspx.

[24] Microsoft. Debugging tools for Windows. https://msdn.microsoft.com/en-us/library/windows/hardware/ff551063(v=vs.85).aspx.

[25] Microsoft. Visual Studio. https://www.visualstudio.com/.

[26] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet. Automatically identifying known software problems. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, pages 433–441, 2007.

[27] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th USENIX Security Symposium*, pages 67–82, 2009.

[28] Mozilla. Mozilla crash reporter. https://support.mozilla.org/en-US/kb/mozillacrashreporter?redirectlocale=en-US&redirectslug=Mozilla+Crash+Reporter.

[29] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the Sixth European Conference on Computer Systems (EuroSys)*, pages 343–356, 2011.

[30] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 465–475, 2003.

[31] M. Renieres and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 30–39, 2003.

[32] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference*, pages 432–449, 1997.

[33] A. Richards. Writing a Debugging Tools for Windows extension. In *MSDN Magazine*, March 2011.

[34] J. Robler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 114–123, 2013.

[35] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–152, 2013.

[36] A. Schroter, N. Bettenburg, and R. Premraj. Do stack trace help developer fix bugs? In *Proceedings of the Seventh IEEE Working Conference on Mining Software Repositories (MSR)*, 2010.

[37] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 28th ACM Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[38] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, Jan. 1986.

[39] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 131–144, 2007.

[40] Ubuntu. Apport crash duplicates. https://wiki.ubuntu.com/ApportCrashDuplicates.

[41] Ubuntu. Crash reporting. https://launchpad.net/ubuntu/+spec/crash-reporting.

[42] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[43] Wikipedia. Call stack. http://en.wikipedia.org/wiki/Call_stack.

[44] Wikipedia. Calling convention. http://en.wikipedia.org/wiki/Calling_convention.

[45] Wikipedia. Core dump. http://en.wikipedia.org/wiki/Core_dump.

[46] Wikipeida. Tail Call. http://en.wikipedia.org/wiki/Tail_call.

[47] Windows Dev Center. Slim reader/writer locks. https://msdn.microsoft.com/en-us/library/windows/desktop/aa904937(v=vs.85).aspx.

[48] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. CrashLocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, 2014.

[49] C. Zamfir. *Execution Synthesis: A Technique for Automating the Debugging of Software*. PhD thesis, EPFL, 2013.

[50] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the Fifth European Conference on Computer Systems (EuroSys)*, pages 321–334, 2010.

[51] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea. Automated debugging for arbitrarily long executions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, pages 20–20, 2013.

[52] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 1–10, 2002.