EXPOSITOR: Scriptable Time-Travel Debugging with First-Class Traces

Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks
Computer Science Department, University of Maryland, College Park
Email: {khooyp,jfoster,mwh}@cs.umd.edu

***Abstract*—We present EXPOSITOR, a new debugging environment that combines scripting and time-travel debugging to allow programmers to automate complex debugging tasks. The fundamental abstraction provided by EXPOSITOR is the execution *trace*, which is a time-indexed sequence of program state snapshots. Programmers can manipulate traces as if they were simple lists with operations such as map and filter. Under the hood, EXPOSITOR efficiently implements traces as lazy, sparse interval trees whose contents are materialized on demand. EXPOSITOR also provides a novel data structure, the *edit hash array mapped trie*, which is a lazy implementation of sets, maps, multisets, and multimaps that enables programmers to maximize the efficiency of their debugging scripts. We have used EXPOSITOR to debug a stack overflow and to unravel a subtle data race in Firefox. We believe that EXPOSITOR represents an important step forward in improving the technology for diagnosing complex, hard-to-understand bugs.**

I. INTRODUCTION

"...we talk a lot about finding bugs, but really, [Firefox's] bottleneck is not finding bugs but fixing [them]..."
—Robert O'Callahan [1]

"[In debugging,] understanding how the failure came to be...requires by far the most time and other resources"
—Andreas Zeller [2]

Debugging program failures is an inescapable task for software programmers. Understanding a failure involves repeated application of the scientific method: the programmer makes some observations; proposes a hypothesis as to the cause of the failure; uses this hypothesis to make predictions about the program's behavior; tests those predictions using experiments; and finally either declares victory or repeats the process with a new or refined hypothesis.

Scriptable debugging is a powerful technique for hypothesis testing in which programmers write scripts to perform complex debugging tasks. For example, suppose we observe a bug involving a cleverly implemented set data structure. We can try to debug the problem by writing a script that maintains a *shadow data structure* that implements the set more simply (e.g., as a list). We run the buggy program, and the script tracks the program's calls to insert and remove, stopping execution when the contents of the shadow data structure fail to match those of the buggy one, helping pinpoint the underlying fault.

While we could have employed the same debugging strategy by altering the program itself (e.g., by inserting print statements and assertions), doing so would require recompilation—and that can take considerable time for large programs (e.g., Firefox), thus greatly slowing the rate of hypothesis testing. Modifying a program can also change its behavior—we have all experienced the frustration of inserting a debugging print statement only to make the problem disappear! Scripts also have the benefit that they can invoke to libraries not used by the program itself, and may be reused in other contexts.

Background: Prior scriptable debuggers. There has been considerable prior work on scriptable debugging. GDB's Python interface makes GDB's interactive commands—stepping, setting breakpoints, etc.—available in a general-purpose programming language. However, this interface employs a callback-oriented programming style which, as pointed out by Marceau et al. [3], reduces composability and reusability as well as complicates checking temporal properties. Marceau et al. propose treating the program as an event generator—each function call, memory reference, etc. can be thought of as an event—and scripts are written in the style of *functional reactive programming* (FRP) [4]. While FRP-style debugging solves the problems of callback-based programming, it has a key limitation: time always marches forward, so we cannot ask questions about prior states. For example, if while debugging a program we find a doubly freed address, we cannot jump backward in time to find the corresponding malloc. Instead we would need to rerun the program from scratch to find that call, which may be problematic if there is any nondeterminism, e.g., if the addresses returned by malloc differ from run to run. Alternatively, we could prospectively gather the addresses returned by malloc as the program runs, but then we would need to record *all* such calls up to the erroneous free.

Time-travel debuggers, like UndoDB [5], and systems for capturing entire program executions, like Amber [6], allow a single nondeterministic execution to be examined at multiple points in time. Unfortunately, *scriptable* time-travel debuggers typically use callback-style programming, with all its problems. (Sec. VI discusses prior work in detail.)

EXPOSITOR: Scriptable, time-travel debugging. In this paper, we present EXPOSITOR, a new scriptable debugging system inspired by FRP-style scripting but with the advantages of time-travel debugging. EXPOSITOR scripts treat a program's execution *trace* as a (potentially infinite) immutable list of time-annotated program state snapshots. Scripts can create or combine traces using common list operations: traces can be filtered, mapped, sliced, folded, and merged to create lightweight projections of the entire program execution. As

such, EXPOSITOR is particularly well suited for checking temporal properties of an execution, and for writing new scripts that analyze traces computed by prior scripts. Furthermore, since EXPOSITOR extends GDB's Python environment and uses the UndoDB [5] time-travel backend for GDB, users can seamlessly switch between running scripts and interacting directly with an execution via GDB. (Sec. II overviews EXPOSITOR's scripting interface.)

The key idea for making EXPOSITOR efficient is to employ *laziness* in its implementation of traces—invoking the time-travel debugger is expensive, and laziness helps minimize the number of calls to it. EXPOSITOR represents traces as sparse, time-indexed interval trees and fills in their contents on demand. For example, suppose we use the breakpoints combinator to create a trace tr containing just the program execution's malloc calls. If we ask for the first element of tr before time 42 (perhaps because there is a suspicious program output then), EXPOSITOR will direct the time-travel debugger to time 42 and run it *backward* until hitting the call, capturing the resulting state in the trace data structure—the remainder of the trace, after time 42 and before the malloc call, is not computed. (Sec. III discusses the implementation of traces.)

In addition to traces, EXPOSITOR scripts typically employ various internal data structures to record information, e.g., the set s of arguments to malloc calls. These data structures must also be lazy so as not to compromise trace laziness—if we eagerly computed the set s just mentioned to answer a membership query at time t, we would have to run the time-travel debugger from the start up until t, considering all malloc calls, even if only the most recent call is sufficient to satisfy the query. Thus, EXPOSITOR provides script writers with a novel data structure: the *edit hash array mapped trie (EditHAMT)*, which provides lazy construction and queries for sets, maps, multisets, and multimaps. As far as we are aware, the EditHAMT is the first data structure to provide these capabilities. (Sec. IV describes the EditHAMT.)

We have used EXPOSITOR to write a number of simple scripts, as well as to debug two more significant problems. Sec. II describes how we used EXPOSITOR to find an exploitable buffer overflow. Sec. V explains how we used EXPOSITOR to track down a deep, subtle bug in Firefox that was never directly fixed, though it was papered over with a subsequent bug fix (the fix resolved the symptom, but did not remove the underlying fault). In the process, we developed several reusable analyses, including a simple race detector. (Sec. V presents our full case study.)

In summary, we believe that EXPOSITOR represents an important step forward in improving the technology for diagnosing complex, hard-to-understand bugs.

II. THE DESIGN OF EXPOSITOR

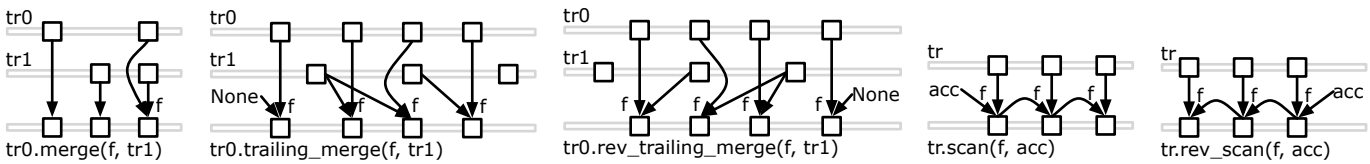
We begin our presentation by describing EXPOSITOR from the perspective of the debugging script writer. Due to lack of space, we defer some details of the design, implementation, and results to our technical report [7].

```
1  class execution:
2    get_at(t):          snapshot at time t
3    breakpoints(fn):    snapshot trace of breakpoints at func fn
4    syscalls(fn):     snapshot trace of breakpoints at syscall fn
5    watchpoints(x, rw):
6           snapshot trace of read/write watchpoints at var x
7    all_calls():        snapshot trace of all function entries
8    all_returns():      snapshot trace of all function exits
9
10   cont():             manually continue the execution
11   get_time():         latest time of the execution
12
13 class trace:
14   __len__():          called by "len(trace)"
15   __iter__():         called by "for item in trace"
16   get_at(t):          item at exactly time t
17   get_after(t):       next item after time t
18   get_before(t):      previous item before time t
19
20   filter(p):          subtrace of items for which p returns true
21   map(f):             new trace with f applied to all items
22   slice(t0, t1):      subtrace from time t0 to time t1
23
24   merge(f, tr):                 see Fig. 2a
25   trailing_merge(f, tr):        see Fig. 2b
26   rev_trailing_merge(f, tr):    see Fig. 2c
27   scan(f, acc):                 see Fig. 2d
28   rev_scan(f, acc):             see Fig. 2e
29
30 class snapshot:
31   read_var(x):        value of variable x in current stack frame
32   read_retaddrs():    return addresses on the stack
33   ... and other methods to access program state ...
34
35 class item:
36   time:               item's execution time
37   value:              item's contents
```

Fig. 1. EXPOSITOR's Python-based scripting API (partial). The get_X and __len__ methods of execution and trace are eager, and the remaining methods of those classes return lazy values.

Fig. 1 lists the key classes and methods of EXPOSITOR's scripting interface (we elide some methods for space), which is written as a library inside UndoDB/GDB's Python environment. The execution class, of which there is a singleton instance the_execution, represents the entire execution of the program being debugged. The trace class represents projections of the execution at points of interest, and items in traces are indexed by their time in the execution. The snapshot class represents a program state at a particular point and provides methods for accessing that state, e.g., reading variables, reading the return addresses on the stack, and so on. There are several methods of execution that produce traces of snapshots. Finally, the item class represents arbitrary values that are associated with a particular point in the execution.

Given a debugging hypothesis, we use the EXPOSITOR interface to apply the following recipe. First, we derive one or more traces from the execution that contain events relevant to the hypothesis; such events could be function calls, breakpoints, system calls, etc. Next, we combine these traces as appropriate, applying filters, maps, and so on to see if our



(a) (b) (c) (d) (e)

Fig. 2. Illustration of complex trace operations.

hypothesis holds. Finally, we query the traces to find the result we are looking for.

A. Example: Reverse engineering a stack-smashing attack

We illustrate the use of EXPOSITOR with a concrete example: reverse engineering a stack-smashing attack, in which malware overflows a stack buffer in the target program to overwrite a return address on the stack, thereby gaining control of the program counter [8].

We develop a reusable script that can detect when the stack has been smashed, which will help pinpoint the attack vector. Our script maintains a *shadow stack* of return addresses and uses it to check that only the top of the stack is modified between function calls or returns; any violation of this property indicates the stack has been smashed.

We begin by using the all_calls and all_returns methods on the_execution to create traces of just the snapshots at function calls and returns, respectively:

```
38 calls = the_execution.all_calls()
39 rets = the_execution.all_returns()
```

Next, we combine these into a single trace so that we can compare consecutive calls or returns. To do so, we use the tr0.merge(f, tr1) method, which creates a new trace containing the events from tr0 and tr1; any items from tr0 and tr1 that occur at the same time are combined with function f (Fig. 2a). Since function calls and returns can never coincide, we can pass **None** for f (as it will not be called):

```
40 calls_rets = calls.merge(None, rets)
```

Now, we map the read_retaddrs method, which returns the list of return addresses on the call stack, over call_returns to create a trace of shadow stacks at every call and return:

```
41 shadow_stacks = calls_rets.map(
42     lambda s: map(int, s.read_retaddrs()))
```

We also use map to cast the return addresses to Python ints.

Then we need to check that, between function calls or returns, the actual call stack matches the shadow stack except for the topmost frame (one return address may be added or removed). We use the following function:

```
43 def find_corrupted(ss, opt_shadow):
44     if opt_shadow.force() is not None:
45         for x, y in zip(ss.read_retaddrs(), opt_shadow.force()):
46             if int(x) != y:
47                 return x # l-value of return address on stack
48     return None
```

Here, find_corrupted takes as arguments a snapshot ss and its immediately preceding shadow stack in opt_shadow; the opt_ prefix indicates that there may not be prior shadow stack (if ss is at the first function call), and we need to call the force method on opt_shadow to retrieve its value (we will explain the significance of this in Sec. III). If there is a prior shadow stack, we compare every return address in ss against the shadow stack and return the first location that differs, or **None** if there are no corrupted addresses. (The zip function creates a list of pairs of the respective elements of the two input lists, up to the length of the shorter list.)

Finally, we generate a trace of corrupted memory locations by applying find_corrupted on calls_rets and shadow_stacks using the tr0.trailing_merge(f, tr1) method. This method creates a new trace by calling f to merge each item from tr0 with the immediately preceding item from tr1, or **None** if there is no preceding item (Fig. 2b). We filter **None** out of the result:

```
49 corrupted_addrs = calls_rets \
50     .trailing_merge(find_corrupted, shadow_stacks) \
51     .filter(lambda x: x is not None)
```

The resulting trace contains exactly the locations of corrupted return addresses at the point they are first evident in the trace.

B. Mini case study: Running EXPOSITOR on tinyhttpd

We used the script just developed on a version of tinyhttpd [9] that we previously modified to include a buffer overflow bug as an exercise for a security class in which students develop exploits of the vulnerability.

As malware, we deployed an exploit that uses a return-to-libc attack [10] against tinyhttpd. The attack causes tinyhttpd to print "Now I pwn your computer" to the terminal and then resume normal operation. Finding buffer overflows using standard techniques can be challenging, since there can be a delay from the exploit overflowing the buffer to the payload taking effect, during which the exploited call stack may be erased by normal program execution. The payload may also erase traces of itself from the stack before producing a symptom.

To use EXPOSITOR, we call the expositor launcher with tinyhttpd as argument, which will start a GDB session with EXPOSITOR's library loaded, and enter the Python interactive prompt from GDB:[1]

```
52 % expositor tinyhttpd
53 (expositor) python-interactive
```

[1]GDB contains an existing python command that is not interactive; python-interactive is a new command that we have submitted to GDB.

Then, we start running tinyhttpd:

```
>>> the_execution.cont() # start running
httpd running on port 47055
```

When tinyhttpd launches, it prints out the port number on which it accepts client connections. On a different terminal, we run the exploit with this port number:

```
% ./exploit.py 47055
Trying port 47055
pwning...
```

At this point, tinyhttpd prints the exploit message, so we interrupt the debugger and use EXPOSITOR to find the stack corruption, starting from the time when we interrupted it:

```
Now I pwn your computer
^C
Program received signal SIGINT, Interrupt
>>> corrupted_addrs = stack_corruption()
                    # function containing Sec. II-A code
>>> time = the_execution.get_time()
>>> last_corrupt = corrupted_addrs.get_before(time)
```

Items in a trace are indexed by time, so the get_before method call above tells EXPOSITOR to start computing corrupted_addrs from the interrupted time backwards, and find the first function call or return when the stack corruption is detected. We can print the results:

```
>>> print time
56686.8
>>> print last_corrupt
Item(56449.2, address)
```

This shows that the interrupt occurred at time 56686.8, and the corrupted stack was first detected at a function call or return at time 56449.2. We can then find and print the snapshot that corrupted the return address with:

```
>>> bad_writes = the_execution \
        .watchpoints(last_corrupt.value, rw=WRITE)
>>> last_bad_write = bad_writes.get_before(last_corrupt.time)
>>> print last_bad_write
Item(56436.0, snapshot)
```

We find that the first write that corrupted the return address occurred at time 56436.0. We can then inspect the snapshot via last_bad_write.value. In this case, the backtrace of the very first snapshot identifies the exact line of code in tinyhttpd, a socket recv with an out-of-bounds pointer, that causes the stack corruption. Notice that to find the bug, EXPOSITOR only inspected from time 56686.8 to time 56436.0. Moreover, had last_corrupt not explained the bug, we would then call corrupted_addrs.get_before(last_corrupt.time) to find the prior corruption event, inspecting only as much of the execution as needed to track down the bug. Notice also that this script can be reused to find stack corruption in any program.

This mini case study also demonstrates that, for some debugging tasks, it can be much faster to search backward in time. It takes only 1 second for corrupted_addrs .get_before(time) to return; whereas if we had instead searched forward from the beginning (e.g., simulating a debugger without time-travel):

```
first_corrupted = corrupted_addrs.get_after(0)
```

it takes 4 seconds for the answer to be computed. Using EXPOSITOR, users can write scripts that search forward or backward in time, as optimal for the task.

C. Additional API methods

The example so far has covered much of the API in Fig. 1. Most of the remaining methods are straightforward. In addition to the methods discussed before, the execution class includes a method get_at to return a snapshot at a particular time, and methods syscalls and watchpoints to return traces at system call entries and when memory values are read or written.

The trace class contains several simple methods for working with traces, such as __len__ for computing the length of a trace. In addition to map and filter, the trace class also includes slice, which returns a subtrace between two times, letting us avoid computation over uninteresting portions of a trace.

The trace class also contains several more complex operations on traces. We have already seen merge and trailing_merge. The rev_trailing_merge method is the same as the latter, except it merges with future items rather than past items (Fig. 2c). The scan method performs a fold- or reduce-like operation for every prefix of an input trace (Fig. 2d). It is called as tr.scan(f, acc), where f is a binary function that takes an accumulator and an item as arguments, and acc is the initial accumulator. It returns a new trace containing the same number of items at the same times as in the input trace tr, where the nth output item out_n is recursively computed as:

$$out_n = \begin{cases} in_n \,ⓕ\, out_{n-1} & \text{if } n > 0 \\ in_n \,ⓕ\, acc & \text{if } n = 0 \end{cases}$$

where f is written infix as ⓕ. The rev_scan method is similar, but deriving a trace based on future items rather than past items (Fig. 2e). rev_scan computes the output item out_n as follows:

$$out_n = \begin{cases} in_n \,ⓕ\, out_{n+1} & \text{if } 0 \le n < length - 1 \\ in_n \,ⓕ\, acc & \text{if } n = length - 1 \end{cases}$$

We will see sample uses of these methods in section IV.

III. LAZY TRACES IN EXPOSITOR

As just discussed, EXPOSITOR allows users to treat traces as if they were lists of snapshots. However, for many applications it would be impractical to eagerly record and analyze full program snapshots at every program point. Instead, EXPOSITOR uses the underlying time-travel debugger, UndoDB, to construct snapshots on demand, and to discard them when they are no longer used (since it is expensive to keep too many snapshots in memory at once). Thus the major challenge is to minimize the demand for snapshots, which EXPOSITOR accomplishes by constructing and manipulating traces *lazily*.

More precisely, all of the trace generators and combinators, including execution.all_calls, trace.map, trace.merge, etc., return immediately without invoking UndoDB. It is only

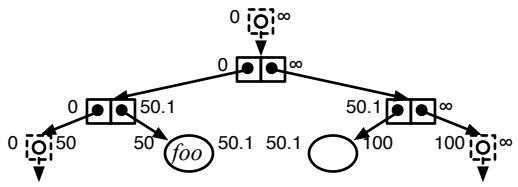
when final values are demanded, with execution.get_at, trace .get_at, trace.get_after, or trace.get_before, that EXPOSITOR queries the actual program execution, and it does so only as much as is needed to acquire the result. For example, the construction of corrupted_addrs in Sec. II-A induces *no* time travel on the underlying program—it is not until the call to corrupted_addrs.get_before(time) in Sec. II-B that EXPOSITOR uses the debugger to acquire the final result.

To achieve this design, EXPOSITOR uses a lazy, interval-tree-like data structure to implement traces. More precisely, a trace is a binary tree whose nodes are annotated with the (closed) lower-bound and (open) upper-bound of the intervals they span, and leaf nodes either contain a value or are empty. The initial tree for a trace contains no elements (only its definition), and EXPOSITOR materializes tree nodes as needed.

As a concrete example, the following trace constructs the tree shown on the right, with a single lazy root node spanning the interval $[0, \infty)$, which we draw as a dotted box and arrow.

```
76 foo = the_execution.breakpoints("foo")
```

Now suppose we call foo.get_before(100). EXPOSITOR sees that the query is looking for the last call to foo before time 100, so it will ask UndoDB to jump to time 100 and then run backward until hitting such a call. Let us suppose the call is at time 50, and the next instruction after that call is at time 50.1. Then EXPOSITOR will expand the root node shown above to the following tree:



Here the trace has been subdivided into four intervals: The intervals $[0, 50)$ and $[100, \infty)$ are lazy nodes with no further information, as EXPOSITOR did not look at those portions of the execution. The interval $[50, 50.1)$ contains the discovered call, and the interval $[50.1, 100)$ is fully resolved and contains no calls to foo. Notice that if we ask the same query again, EXPOSITOR can traverse the interval tree above to respond without needing to query UndoDB.

Likewise, calling get_at(t) or get_after(t) either returns immediately (if the result has already been computed) or causes UndoDB to jump to time t (and, for get_after(t), to then execute forward). These methods may return **None**, e.g., if a call to foo did not occur before/after/at time t.

In the scripts we have written, we find that if we request 30-50% of items in a trace, computing traces lazily takes less time than computing eagerly. The performance varies depending on the query pattern, e.g., get_before is more expensive than get_at or get_after, since the former requires UndoDB to search backward, as well as the kind of computations done, e.g., an expensive map helper function will overshadow the cost of laziness (see our technical report for details [7]).

A. Lazy trace operations

We implement filter, map, and slice lazily on top of the interval tree data structure. For a call tr1 = tr0.map(f), we initially construct an empty interval tree, and when values are demanded in tr1 (by get_X calls), EXPOSITOR conceptually calls tr0.get_X, applies f to the result, and caches the result for future reuse. Calls to tr0.filter(p) are handled similarly, constructing a lazy tree that, when demanded, repeatedly gets values from tr0 until p is satisfied. Note that for efficiency, EXPOSITOR's implementation actually does not directly call get_X on the underlying traces, but instead manipulates their tree data structures directly.

The implementation of tr0.merge(f, tr1) also calls get_X on tr1 also as required. For a call tr.slice(t0, t1) EXPOSITOR creates an interval tree that delegates get_X calls, asking for items from time t0 to time t1 to tr, and returns **None** for items that fall outside that interval.

For the last four operations, [rev_]trailing_merge and [rev_]scan, EXPOSITOR employs additional laziness in the helper function argument f. To illustrate, consider a call to tr.scan(f, acc). Here, EXPOSITOR passes the accumulator to f wrapped in an instance of class lazy, defined as follows:

```
77 class lazy:
78     force():      return the actual value
79     is_forced():  return whether force has been called
```

The force method, when first called, will compute the actual value and cache it; the cached value is returned in subsequent calls. Thus, f can force the accumulator as needed, and if it is not forced, it will not be computed.

To see the benefit, consider the following example, which uses scan to derive a new trace in which each item is a count of the number of consecutive calls to foo with nonzero arguments, resetting the count when foo is called with zero:

```
80 foo = execution.breakpoints("foo") # void foo(int x)
81 def count_nonzero_foo(lazy_acc, snapshot):
82     if snapshot.read_var("x") != 0:
83         return lazy_acc.force() + 1
84     else:
85         return 0
86 nonzero_foo = foo.scan(count_nonzero_foo, 0)
```

Notice that if lazy_acc were not lazy, EXPOSITOR would have to compute its value before calling count_nonzero_foo. By the definition of scan (Fig. 2d), this means that it must recursively call count_nonzero_foo to compute all prior output items before computing the current item, even if it is unnecessary to do so, e.g., if we had called nonzero_foo.get_before(t), and the call to foo just before time t had argument x=0. Thus, a lazy accumulator avoids this unnecessary work. EXPOSITOR uses a lazy accumulator in rev_scan for the same reason.

Likewise, observe that in tr0.trailing_merge(f, tr1), for a particular item in tr0 the function f may not need to look in tr1 to determine its result; thus, EXPOSITOR wraps the tr1 argument to f in an instance of class lazy. The implementation of rev_trailing_merge similarly passes lazy items from tr1 to f. Note that there is no such laziness in the regular merge

operation. The reason is that in tr0.merge(f, tr1), the items from tr0 and tr1 that are combined with f occur at the same time. Thus, making f's arguments lazy would not reduce demands on the underlying time-travel debugger.

B. Tree scan

Finally, EXPOSITOR provides another list combinator, *tree-scan*, which is a lazier variant of scan that is sometimes more efficient. Tree scan is invoked with tr.tscan(f), where f must be an associative function that is lazy and optional in its left argument and lazy in its right argument. The tscan method generates an output trace of the same length as the input trace, where the nth output out_n is defined as:

$$out_n = in_0 \,ⓕ\, in_1 \,ⓕ\, \cdots \,ⓕ\, in_n$$

Notice that there is no accumulator, and EXPOSITOR can apply f in any order, since it is associative. When a value at time t is demanded from the output trace, EXPOSITOR first demands the item in_n at that time in the input trace (if no such item exists, then there is no item at that time in the output trace). Then EXPOSITOR walks down the interval tree structure of the input trace, calling f (only if demanded) on each internal tree node's children to compute out_n. Since the interval tree for the input trace is computed lazily, f may sometimes be called with **None** as a left argument, for the case when f forces an interval that turns out to contain no values; thus for correctness, we also require that f treats **None** as a left identity. (The right argument corresponds to in_n and so will never be **None**.)

Because both arguments of f are lazy, EXPOSITOR avoids computing either argument unnecessarily. The is_forced method of the lazy class is particularly useful for tscan, as it allows us to determine if either argument has been forced and evaluate it first. For example, we can find if a trace contains a true value as follows:

```
87 def has_true(lazyleft, lazyright):
88     return lazyleft.is_forced() and lazyleft.force() \
89         or lazyright.is_forced() and lazyright.force() \
90         or lazyleft.force() or lazyright.force()
91 has_true_trace = some_trace.tscan(has_true)
92 last_has_true = has_true_trace.get_before("inf")
```

The best case for this example occurs if either lazyleft or lazyright have been forced by a prior query, in which case either the first clause (line 88) or second clause (line 89) will be true and the unforced argument need not be computed due to short-circuiting.

EXPOSITOR's rev_tscan derives a new trace based on future items instead of past items, computing output item out_n as:

$$out_n = in_n \,ⓕ\, in_{n+1} \,ⓕ\, \cdots \,ⓕ\, in_{length-1}$$

Here, the right argument to f is optional, rather than the left.

IV. THE EDIT HASH ARRAY MAPPED TRIE

Many of the EXPOSITOR scripts we have written use sets or maps to record information about the program execution. Unfortunately, a typical eager implementation of them could demand all items in the traces, defeating the intention of

```
98  class edithamt:
99      find(k):          Return the value for k or None if not found
100     find_multi(k):    Return an iterator of all values bound to k
101
102     # Static factory methods to create new EditHAMTs:
103     addkeyvalue(lazy_eh, k, v):
104                           Add binding of k to v to lazy_eh
105     remove(lazy_eh, k):
106                       Remove all bindings for k from lazy_eh
107     concat(lazy_eh1, lazy_eh2):
108                 Concatenate lazy_eh2 edit history to lazy_eh1
```

Fig. 3. The EditHAMT API (partial).

EXPOSITOR's lazy trace data structure. For example, consider the following code, which uses Python's standard (non-lazy) set class to collect all arguments in calls to a function foo:

```
93 foos = the_execution.breakpoints("foo") # void foo(int arg)
94 def collect_foo_args(lazy_acc, snap):
95     return lazy_acc.force().union( \
96         set([ int(snap.read_var("arg")) ]))
97 foo_args = foos.scan(collect_foo_args, set())
```

Notice that we must force lazy_acc to call the union method to create a deep copy of the updated set (lines 95–96). Unfortunately, forcing lazy_acc causes the immediately preceding set to be computed by recursively calling collect_foo_args. As a result, we must compute all preceding sets in the trace even if a particular query could be answered without doing so.

To address these problems, we developed the *edit hash array mapped trie* (*EditHAMT*), a new set, map, multiset, and multimap data structure that supports lazy construction and queries, to complement the trace data structure.

A. Using the EditHAMT

From the user's perspective, the EditHAMT is an immutable data structure that maintains the entire history of edit operations for each EditHAMT. Fig. 3 shows a portion of the API. The edithamt class includes find(k) and find_multi(k) methods to look up the most recent value or all values mapped to key k, respectively. (We omit set/multiset operations and some multimap operations for brevity). EditHAMT operations are implemented as static factory methods: calling edithamt.addkeyvalue(lazy_eh, k, v) and edithamt .remove(lazy_eh, k) makes new EditHAMTs by adding or removing a binding from their EditHAMT arguments; we can pass **None** into lazy_eh for an empty EditHAMT. The lazy_eh argument to both methods is lazy so that we need not force it until a call to find or find_multi demands a result.

The last static factory method, edithamt.concat(lazy_eh1, lazy_eh2), concatenates the edit histories of its arguments. For example:

```
109 eh_rem = edithamt.remove(None, "x")
110 eh_add = edithamt.addkeyvalue(None, "x", 42)
111 eh = edithamt.concat(eh_add, eh_rem)
```

Here eh is the empty EditHAMT, since it contains the additions in eh_add followed by the removals in eh_rem. A common EXPOSITOR script pattern is to map a trace to a

sequence of EditHAMT additions and removals, and then use edithamt.concat with scan or tscan to concatenate those edits.

As an example, we present the race detector used in our Firefox case study (Section V). The detector compares each memory access against prior accesses to the same location from any thread. Since UndoDB serializes thread schedules, each read need only be compared against the immediately preceding write, and each write against all preceding reads up to and including the immediately preceding write.

To start, we define a function that uses the EditHAMT as a multimap to track the access history of a given variable v:

```
112 def access_events(v):
113     reads = the_execution.watchpoints(v, rw=READ) \
114         .map(lambda s: edithamt.addkeyvalue( \
115             None, v, ("read", s.get_thread_id())))
116     writes = the_execution.watchpoints(v, rw=WRITE) \
117         .map(lambda s: edithamt.addkeyvalue( \
118             edithamt.remove(None, v), \
119             v, ("write", s.get_thread_id()))
120     return reads.merge(None, writes)
```

In access_events, we create the trace reads by finding all reads to v using the watchpoints method (line 113), and then mapping each snapshot to a singleton EditHAMT that binds v to a tuple of "read" and the running thread ID (lines 114–115). Similarly, we create the trace writes for writes to v (line 116), but instead map each write snapshot to an EditHAMT that first removes all prior bindings for v (line 118), then binds v to a tuple of "write" and the thread ID (lines 117–119). Finally, we merge reads and writes, and return the result (line 120).

We are not done yet, since the EditHAMTs in the trace returned by access_events contain only edit operations corresponding to individual accesses to v. We can get a trace of EditHAMTs that records all accesses to v from the beginning of the execution by using scan with edithamt.concat to concatenate the individual EditHAMTs. For example, we can record the access history of var1 as follows:

```
121 var1_history = access_events("var1").scan(edithamt.concat)
```

We can also track multiple variables by calling access_events on each variable, merging the traces, then concatenating the merged trace, e.g., to track var1 and var2:

```
122 access_history = \
123     access_events("var1").merge(access_events("var2")) \
124         .scan(edithamt.concat)
```

Since trace methods are lazy, this code completes immediately; the EditHAMT operations will only be applied, and the underlying traces forced, when we request a particular access, e.g., at the end of the execution (time "inf"):

```
125 last = access_history.get_before("inf")
```

To see laziness in action, consider applying the above analysis to an execution depicted in Fig. 4, which shows two threads at the top and the corresponding EditHAMT operations at the bottom. Now suppose we print the latest access to var1 at time t_4 using the find method:

```
126 >>> print last.find("var1")
127 ("read", 2)
```

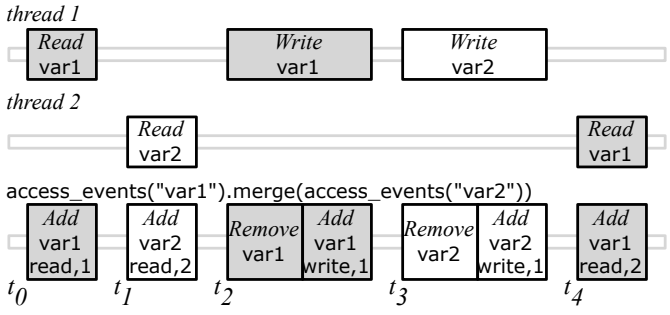


Fig. 4. Example execution with two threads accessing var1 (gray) and var2, and the corresponding EditHAMT operations returned by access_events.

Because "var1" was just added at time t_4, answering this query will only force the EditHAMT and query the time-travel debugger at time t_4, and not before.

As another example, suppose we want to find all accesses to var1 from the last access backwards using find_multi:

```
128 >>> for mem_access in last.find_multi("var1"):
129         print mem_access
130 ("read", 2)
131 ("write", 1)
```

Here since all "var1" bindings added prior to time t_2 were removed at time t_2, the results are computed without forcing any EditHAMTs or querying the debugger before time t_2.

B. Implementation

The EditHAMT is inspired by the *hash array mapped trie* (HAMT) [11]. Like the HAMT, the EditHAMT is a hybrid data structure combining the fast lookup of a hash table and the memory efficiency of a trie. Just as a hash table uses an array of buckets to map keys to values, the HAMT uses an *array mapped trie* (AMT)—a trie that maps fixed-width integer keys to values—for the same purpose; hash collisions are resolved using nested HAMTs with different hash functions.

We developed the EditHAMT by making two changes to the traditional HAMT. First, we replaced the AMT with the *LazyAMT*, which supports lazy, rather than eager, updates. Second, we resolve hash collisions, as well as support remove operations, using *EditList*s, which are lazy linked-lists of nodes tallying edit operations on the EditHAMT; the tails are lazily retrieved from the prior EditHAMT.

By representing a set as a list of edit operations rather than by its elements, the EditList allows us to modify a set by simply appending an add or remove operation to the list. This eliminates the need to know the rest of the set or to force the source EditList. (The same benefit holds for maps, multisets and multimaps.) However, a query on an EditList is slow since it takes $O(n)$ time, where n is the number of edit operations. Instead, we build multiple EditLists, each containing only edits for values with the same hash, and use the LazyAMT to map hashes to EditLists, reducing the cost of lookups to $O(1)$ time (assuming no hash collisions).

Furthermore, it is more memory efficient to make an updated copy of a LazyAMT in the EditHAMT, since only nodes

along the path to the updated binding need to be copied, than it is to make a copy of the bucket array in a hash table, which can be much larger. This makes it viable to store every intermediate EditHAMT as it is created in a trace, as each EditHAMT only requires an additional $O(1)$ memory over the prior EditHAMT. In our current implementation, a trace of EditHAMTs is cheaper than a trace of Python sets (which requires deep copying) if, on average, each EditHAMT or set in the trace has more than eight elements.

Map lookups in EditHAMTs are similar to set membership queries and take $O(1)$ time as well (assuming no hash collisions). Multiset and multimap lookups also take $O(1)$ time per added element or binding on average; however, if we remove individual elements or bindings, then lookups will take $O(n)$ time where n is the number of remove operations.

V. Firefox case study: delayed deallocation bug

To put EXPOSITOR to the test, we used it to track down a subtle bug in Firefox that caused it to use more memory than expected [12]. The bug report contains a test page that, when scrolled, creates a large number of temporary JavaScript objects that should be immediately garbage collected. However, in a version of Firefox that exhibits the bug (revision c5e3c81d35ba), the memory usage increases by 70MB (as reported by top), and only decreases 20 seconds after a second scroll. As it turns out, this bug has never been directly fixed—the actual cause is a data race, but the official fix instead papers over the problem by adding another GC trigger.

Our initial hypothesis for this bug is that there is a problem in the JavaScript garbage collector (GC). To test this hypothesis, we first run Firefox under EXPOSITOR, load the test page, and scroll it twice, temporarily interrupting the execution to call the_execution.get_time() just before each scroll, $t_{scroll1}$ and $t_{scroll2}$, and after the memory usage decreases, t_{end}. Then, we create several traces to help us understand the GC and track down the bug, as summarized in Fig. 5.

We observe the GC behavior using a trace of the calls to (gc_call) and returns from (gc_return) function js_GC (Fig. 5a).[2] Also, we find out when memory is allocated or released to the operating system using mmap2 and munmap traces of the same-named system calls (Fig. 5b). Printing these traces reveals some oddly inconsistent behavior: the GC is called only once after $t_{scroll1}$, but five times after $t_{scroll2}$; and memory is allocated after $t_{scroll1}$ and deallocated just before t_{end}. To make sense of these inconsistencies, we inspect the call stack of each snapshots in gc_call and discover that the first js_GC call immediately after a scroll is triggered by a scroll event, but subsequent calls are triggered by a timer.

We now suspect that the first scroll somehow failed to trigger the creation of subsequent GC timers. To understand how these timers are created, we write a function called set_tracing that creates a trace for analyzing set-like behavior, using EditHAMTs to track when values are inserted or removed, and apply set_tracing to create timer_trace by treating timer creation as set insertion, and timer triggering as set removal (Fig. 5c). This trace reveals that each js_GC call creates a GC timer (between gc_call and gc_return snapshots), except the js_GC call after the first scroll (and the last js_GC call because GC is complete).

To find out why the first js_GC call does not create a GC timer, we inspect call stacks again and learn that a GC timer is only created when the variable gcChunksWaitingToExpire is nonzero, and yet it is zero when the first js_GC returns (at the first gc_return snapshot). Following this clue, we create a watchpoint trace on gcChunksWaitingToExpire and discover that it remained zero through the first js_GC call and becomes nonzero only after the first js_GC returns. It stayed nonzero through the second scroll and second js_GC call, causing the first GC timer to be created after that (Fig. 5d).

We posit that, for the GC to behave correctly, gcChunksWaitingToExpire should become nonzero at some point during the first js_GC call. Inspecting call stacks again, we find that gcChunksWaitingToExpire is changed in a separate helper thread, and that, while the GC owns a mutex lock, it is not used consistently around gcChunksWaitingToExpire. This leads us to suspect that there is a data race. Thus, we develop a simple race detection script, one_lock, that works by comparing each access to a chosen variable against prior accesses from different threads (Section IV-A explains how we track prior accesses), and checking if a particular lock was acquired or released prior to those accesses. For each pair of accesses, if at least one access is a write, and the lock was not held in one or both accesses, then there is a race, which we indicate as an item containing the snapshot of the prior access. We apply this race detector to gcChunksWaitingToExpire and confirm our suspicion that, after $t_{scroll1}$, there is a write that races with a prior read during the first js_GC call when the timer should have been created (Fig. 5e).

To give a sense of EXPOSITOR's performance, it takes 2m6s to run the test page to $t_{scroll2}$ while printing the gc_call trace, with 383MB maximum resident memory (including GDB, since EXPOSITOR extends GDB's Python environment). The equivalent task in GDB/UndoDB without EXPOSITOR takes 2m19s and uses 351MB of memory (some difference is inevitable as the test requires user input, and Firefox has many sources of nondeterminism). As another data point, finding the race after $t_{scroll1}$ takes 37s and another 5.4MB of memory.

The two analyses we developed, set_tracing and one_lock, take only 10 and 40 lines of code to implement, respectively, and both can be reused in other debugging contexts.

VI. Related work

EXPOSITOR provides scripting for time-travel debuggers, with the central idea that a target program's execution can be manipulated (i.e., queried and computed over) as a first-class object. Prior work on time-travel debugging has largely provided low-level access to the underlying execution without consideration for scripting. Of the prior work on scriptable debugging, EXPOSITOR is most similar to work that views the

[2]The index=-1 optional argument to execution.breakpoints indicates that the breakpoint should be set at the end of the function.

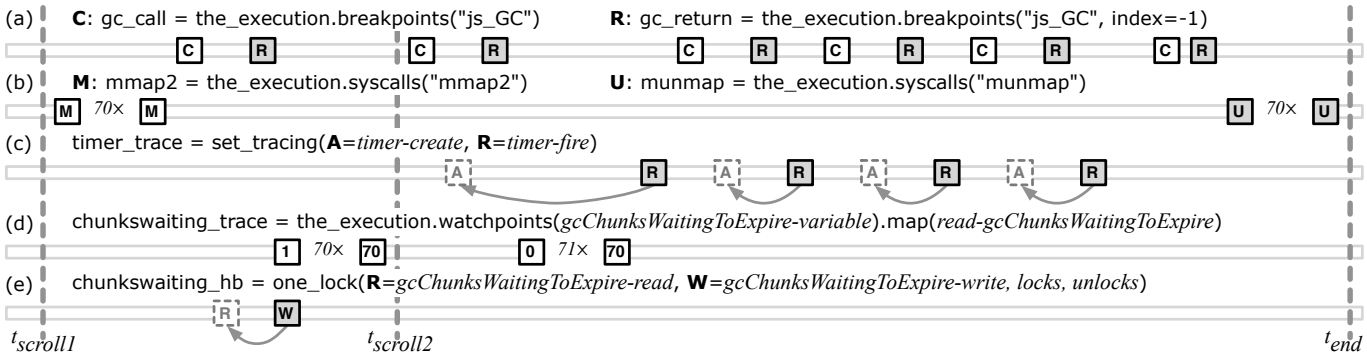


Fig. 5. Timeline of items in traces used to debug Firefox.

program as an event generator—with events seeded from function calls, memory reads/writes, etc.—and debugging scripts as database-style queries over event streams or as dataflow-oriented stream transformers. None of this scripting work includes the notion of time travel.

A. Time-travel debuggers

Broadly speaking, there are two classes of time-travel debuggers. *Omniscient debuggers* work by logging the state of the program being debugged after every instruction, and then reconstructing the state from the log on demand. Some examples of omniscient debuggers include ODB [13], Amber (also known as Chronicle) [6], Tralfamadore [14], and TOD [15]. In contrast, *replay debuggers* work by logging the results of system calls the program makes (as well as other sources of nondeterminism) and making intermediate checkpoints, so that the debugger can reconstruct a requested program state by starting at a checkpoint and replaying the program with the logged system calls. Several recent debuggers of this style include URDB [16] and UndoDB [5] (which we used in our prototype) for user-level programs, and TTVM [17] and VMware ReTrace [18] for entire virtual machines. EXPOSITOR could target either style of debugger in principle, but replay debugging scales much better (e.g., about 1.7× recording overhead for UndoDB vs. 300× for Amber). Engblom [19] provides a more comprehensive survey on time-travel debugging techniques and implementations.

The above work focuses on implementing time travel efficiently; most systems provide very simple APIs for accessing the underlying execution, and do not consider how time travel might best be exploited by debugging scripts.

Similarly, GDB's Python environment simply allows a Python program to execute GDB (and UndoDB) commands in a callback-oriented, imperative style. This is quite tedious, e.g., just counting the number of calls to a particular function takes 16 lines of code and cannot be composed with other scripts (e.g., to refine the count to calls that satisfy predicate p). EXPOSITOR's notion of traces is simpler and more composable: function call counting can be done in one line by computing the length of a breakpoint trace; to refine the count, we simply filter the trace with p before counting.

Tralfamadore [20] considers generalizing standard debugging commands to entire executions, but does not provide a way to customize these commands with scripts.

Whyline is a kind of omniscient debugger with which users can ask "*why did*" and "*why didn't*" questions about the control- and data-flow in the execution, e.g., "*why did this Button's visible = true*" or "*why didn't Window appear*" [21]. Whyline records execution events (adding 1.7× to 8.5× overhead), and when debugging begins, it uses program slicing [22] to generate questions and the corresponding answers (imposing up to a 20× further slowdown). Whyline is good at what it does, but its lack of scriptability limits its reach; it is hard to see how we might have used it to debug the Firefox memory leak, for example. In concept, Whyline can be implemented on top of EXPOSITOR, but limitations of GDB and UndoDB (in particular, the high cost of software watchpoints, and the inability to track data-flow through registers) makes it prohibitively expensive to track fine-grained data-flow in an execution. We plan to overcome this limitation in future work, e.g., using EDDI [23] to implement fast software watchpoints.

B. High-level (non-callback oriented) debugging scripts

EXPOSITOR's design was inspired by MzTake [3], a Scheme-based, interactive, scriptable debugger for Java based on *functional reactive programming*. In MzTake, the program being debugged is treated as a source of *event streams* consisting of events such as function calls or value changes. Event streams can be manipulated with combinators that filter, map, fold, or merge events to derive new event streams. As such, an event stream in MzTake is like a trace in EXPOSITOR. Computations in MzTake are implicitly over the most recent value of a stream and are evaluated eagerly as the target program runs. To illustrate, consider our example of maintaining a shadow stack from Section II-A. In MzTake, when the target program calls a function, a new snapshot event `s` becomes available on the `calls` stream. The `calls_rets` stream's most recent event is the most recent of `calls` and `rets`, so MzTake updates it to `s`. Since `shadow_stacks` is derived from `calls_rets`, MzTake updates its most recent event by executing `map(int, s.read_retaddrs())`.

This eager updating of event streams, as the program executes, can be less efficient than using EXPOSITOR. In particular, EXPOSITOR evaluates traces lazily so that computation

can be narrowed to a few slices of time. In Section II-A, we find the *latest* smashed stack address without having to maintain the shadow stack for the entire program execution, as would be required for MzTake. Also, EXPOSITOR traces are time indexed, but MzTake event streams are not: there is no analogue to tr.get_at(i) or tr.slice(t0, t1) in MzTake. We find time indexing to be very useful for interactivity: we can run scripts to identify an interesting moment in the execution, then explore the execution before and after that time. Similarly, we can learn something useful from the end of the execution (e.g., the address of a memory address that is double-freed), and then use it in a script on an earlier part of the execution (e.g., looking for where that address was first freed). MzTake requires a rerun of the program, which can be a problem if nondeterminism affects the relevant computation.

Dalek [24] and Event Based Behavioral Abstraction (EBBA) [25] bear some resemblance to MzTake and suffer the same drawbacks, but are much lower-level, e.g., the programmer is responsible for manually managing the firing and suppression of events. Coca [26] is a Prolog-based query language that allows users to write predicates over program states; program execution is driven by Prolog backtracking, e.g., to find the next state to match the predicate. Coca provides a retrace primitive that restarts the entire execution to match against new predicates. This is not true time travel but re-execution, and thus suffers the same problems as MzTake.

PTQL [27], PQL [28], and UFO [29] are declarative languages for querying program executions, as a debugging aid. Queries are implemented by instrumenting the program to gather the relevant data. In principle, these languages are subsumed by EXPOSITOR, as it is straightforward to compile queries to traces. Running queries in EXPOSITOR would allow programmers to combine results from multiple queries, execute queries lazily, and avoid having to recompile (and potentially perturb the execution of) the program for each query. On the other hand, it remains to be seen whether EXPOSITOR traces would be as efficient as using instrumentation.

VII. CONCLUSION

We have introduced EXPOSITOR, a novel scriptable, time-travel debugging system. EXPOSITOR allows programmers to project the program execution onto traces, which support a range of powerful combinators including map, filter, merge, and scan. Working with traces gives the programmer a global view of the program, and provides a convenient way to correlate and understand events across the execution timeline. For efficiency, EXPOSITOR traces are implemented using a lazy, interval-tree-like data structure. EXPOSITOR materializes the tree nodes on demand, ultimately calling UndoDB to retrieve appropriate snapshots of the program execution. EXPOSITOR also includes the EditHAMT, which lets script writers create lazy sets, maps, multisets, and multimaps that integrate with traces without compromising their laziness. We used EXPOSITOR to find a buffer overflow in a small program, and to diagnose a very complex, subtle bug in Firefox. We believe that EXPOSITOR is a useful tool for helping programmers understand complex bugs in large software systems.

ACKNOWLEDGMENTS

This research was supported by in part by National Science Foundation grants CCF-0910530 and CCF-0915978.

REFERENCES

[1] R. O'Callahan. (2010) LFX2010: A browser developer's wish list. Mozilla. At 27:15. [Online]. Available: http://vimeo.com/groups/lfx/videos/12471856#t=27m15s
[2] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufman, 2006.
[3] G. Marceau, G. Cooper, J. Spiro, S. Krishnamurthi, and S. Reiss, "The design and implementation of a dataflow language for scriptable debugging," in *ASE*, 2007.
[4] C. Elliott and P. Hudak, "Functional reactive animation," in *ICFP*, 1997.
[5] Undo Software. About UndoDB. [Online]. Available: http://undo-software.com/product/undodb-overview
[6] R. O'Callahan, "Efficient collection and storage of indexed program traces," 2006, unpublished.
[7] Khoo Y. P., J. S. Foster, and M. Hicks, "Expositor: Scriptable Time-Travel Debugging with First Class Traces (Extended Version)," UMD–College Park, Tech. Rep. CS-TR-5021, 2013.
[8] A. One, "Smashing the stack for fun and profit," *Phrack*, no. 49, 1996.
[9] J. D. Blackstone. Tiny HTTPd. [Online]. Available: http://tinyhttpd.sourceforge.net/
[10] S. Designer, "'return-to-libc' attack," *Bugtraq*, Aug. 1997.
[11] P. Bagwell, "Ideal hash trees," EPFL, Tech. Rep., 2001.
[12] A. Zakai. (2011, May) Bug 654028 - 70mb of collectible garbage not cleaned up. Bugzilla@Mozilla. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=654028
[13] B. Lewis, "Debugging backwards in time," in *AADEBUG*, 2003.
[14] G. Lefebvre, B. Cully, C. Head, M. Spear, N. Hutchinson, M. Feeley, and A. Warfield, "Execution mining," in *VEE*, 2012.
[15] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," in *OOPSLA*, 2007.
[16] A.-M. Visan, K. Arya, G. Cooperman, and T. Denniston, "URDB: a universal reversible debugger based on decomposing debugging histories," in *PLOS*, 2011.
[17] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *USENIX ATC*, 2005.
[18] M. Sheldon and G. Weissman, "ReTrace: Collecting execution trace with virtual machine deterministic replay," in *MoBS*, 2007.
[19] J. Engblom, "A review of reverse debugging," in *S4D*, 2012.
[20] C. C. D. Head, G. Lefebvre, M. Spear, N. Taylor, and A. Warfield, "Debugging through time with the Tralfamadore debugger," in *RESoLVE*, 2012.
[21] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *ICSE*, 2008.
[22] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005.
[23] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong, "How to do a million watchpoints: Efficient debugging using dynamic instrumentation," in *CC*, 2008.
[24] R. A. Olsson, R. H. Crawford, and W. W. Ho, "A dataflow approach to event-based debugging," *Software: Practice and Experience*, vol. 21, no. 2, pp. 209–229, 1991.
[25] P. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," in *PADD*, 1988.
[26] M. Ducasse, "Coca: A debugger for c based on fine grained control flow and data events," in *ICSE*, 1999.
[27] S. F. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," in *OOPSLA*, 2005.
[28] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *OOPSLA*, 2005.
[29] M. Auguston, C. Jeffery, and S. Underwood, "A framework for automatic debugging," in *ASE*, 2002.