

MIT Open Access Articles

Partial replay of long-running applications

The MIT Faculty has made this article openly available. **Please share** how this access benefits you. Your story matters.

Citation: Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2011. Partial replay of long-running applications. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 135-145.

As Published: <http://dx.doi.org/10.1145/2025113.2025135>

Publisher: Association for Computing Machinery (ACM)

Persistent URL: <http://hdl.handle.net/1721.1/73450>

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike 3.0



Partial Replay of Long-Running Applications

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden
MIT CSAIL
{akcheung, asolar, madden}@csail.mit.edu

ABSTRACT

Bugs in deployed software can be extremely difficult to track down. Invasive logging techniques, such as logging all non-deterministic inputs, can incur substantial runtime overheads. This paper shows how symbolic analysis can be used to re-create path equivalent executions for very long running programs such as databases and web servers. The goal is to help developers debug such long-running programs by allowing them to walk through an execution of the last few requests or transactions leading up to an error. The challenge is to provide this functionality without the high runtime overheads associated with traditional replay techniques based on input logging or memory snapshots. Our approach achieves this by recording a small amount of information about program execution, such as the direction of branches taken, and then using symbolic analysis to reconstruct the execution of the last few inputs processed by the application, as well as the state of memory before these inputs were executed.

We implemented our technique in a new tool called *bbr*. In this paper, we show that it can be used to replay bugs in long-running single-threaded programs starting from the middle of an execution. We show that *bbr* incurs low recording overhead (avg. of 10%) during program execution, which is much less than existing replay schemes. We also show that it can reproduce real bugs from web servers, database systems, and other common utilities.

Categories and Subject Descriptors

D.2.5 [Software Engineering] – Testing and Debugging

General Terms Reliability, Performance

1. INTRODUCTION

A large amount of research effort has been devoted to the problem of identifying bugs and helping programmers pinpoint their root causes. However, not all bugs are equal. Most development organizations already have more bug reports than resources to fix them. As such, a bug from an important customer is much more critical than a bug dis-

covered by an automated test generator. Unfortunately, pinpointing and fixing such critical bugs in the field can be very challenging. Stack traces and core dumps can be helpful when dealing with some errors, and statistical bug isolation can be used when a bug affects large numbers of users. However, for bugs only manifest as a result of a specific input, these techniques are not useful.

The gold standard for pinpointing such bugs is program replay. If users could provide a log of the entire execution of their system or all of its inputs, then identifying and fixing such bugs would be simple: when the bug is observed, you would have a complete trace of the execution that could be used to reproduce and fix the bug. In fact, replay tools exist that able to reconstruct an execution by either taking periodic snapshots of the system state [33], or by logging all non-deterministic inputs to the target program, such as the return values from non-deterministic functions (e.g., `random`), system calls, and user inputs [17]. The problem with these tools, however, is that while they can reproduce the exact scenario that leads to the buggy behavior (modulo hardware failures), they can significantly slow down the normal execution of the program and produce very large data logs that may grow without bound as long as the program keeps executing. This makes them impractical for debugging systems like databases and web servers that may run for months at a time.

To mitigate this runtime overhead problem, there has been work done on performing software replay using symbolic execution [31, 15, 19]. The idea is to capture a subset of the program state during runtime, thus incurring less overhead, and then use symbolic execution to reconstruct the missing information. Even though the reconstructed state might not be exactly the same as the one that the user experienced, it is nonetheless useful for debugging purposes, since it is usually the case that the same bug can be triggered by multiple execution traces. Thus, as long as the reconstructed trace can still trigger the bug then it is good enough. Unfortunately, all the previous work focuses on replaying programs from the beginning of the execution, and today's symbolic execution engines (which rely on SMT solvers) are not able to replay programs that have been running for days or months.

In this paper, we explore the idea of using symbolic execution to perform *partial* replay from the middle of executions, rather than replaying long-running programs from the very beginning. By doing this, we avoid having to store and transmit very large logs (as required by deterministic replays) and mitigate the very long replay times that can result from symbolic execution. Our tool, called *bbr*, is the first

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

Copyright 2011 ACM 978-1-4503-0443-6/11/09 ...\$10.00.

such partial symbolic replay tool. To use bbr, the developer annotates the code with checkpoints, for instance at places in the program that mark the transition from processing one request or phase to the next, such as at each invocation of the query parser in a database server. bbr collects a log during program execution and, when a checkpoint is reached, discards all logged data that was previously collected, limiting the size of the logs that need to be maintained. When the program terminates, the developer gives the (partial) log to bbr to replay starting from the last checkpoint. bbr symbolically executes the program, generating constraints when the program uses values that were not recorded. At the end of the symbolic execution, the constraints are solved to find the program state at the beginning of the replay. The solved state might not be exactly the same as the one in the original execution, but it is one guaranteed to take the same control flow path when executed with the logged data. We refer to this as a *branch-deterministic* replay, and we show that such a replay is as useful as a fully deterministic replay in debugging a number of real-world bugs, while avoiding the expensive cost of the data-logging approaches.

Unlike the data-logging based schemes, bbr works by recording control flow (one bit per branch) and accessed array indices into a log during program execution, as well the current stack and file offset information for the currently opened files. when a checkpoint is taken. This amounts to much less data than in the data-log based approaches (which can accumulate large logs even in partial replay scenarios).

In summary, this paper makes the following contributions:

- We introduce the idea of partial symbolic replay at an arbitrary program point in the target application, which lets users replay long-running programs without the burden of maintaining large logs. Furthermore, we develop an algorithm to handle pointer aliases that arise during partial replay without using the theory of bit-vector arrays, which, as our results show, gives poor performance.
- We propose the concept of branch-deterministic replay by recording control flow and non-constant array indices, and demonstrate its applicability to data-intensive applications, showing low runtime overhead and small log sizes as compared to other existing replayers.
- We identify several techniques that enable replaying of real-world applications in our bbr prototype, including a new memory model, separating constraints into independent subgroups during solving, and building a parallel solver implementation.
- We show that our technique can scale to replay real programs, including sqlite running on a 10 GB TPC-C database and two different web servers. Our runtime overheads range from 1% to 33%, with reasonable replay and solving times. We also show that we can reproduce several real bugs in these systems.

The rest of this paper is organized as follows. In Section 2, we overview the architecture of bbr and provide a sample usage scenario. In Section 3, we discuss in detail the design of the different components of bbr. Section 4 describes our experimental evaluation of bbr on a number of different real-world applications. Finally, Section 5 surveys related work, followed by conclusions in Section 6.

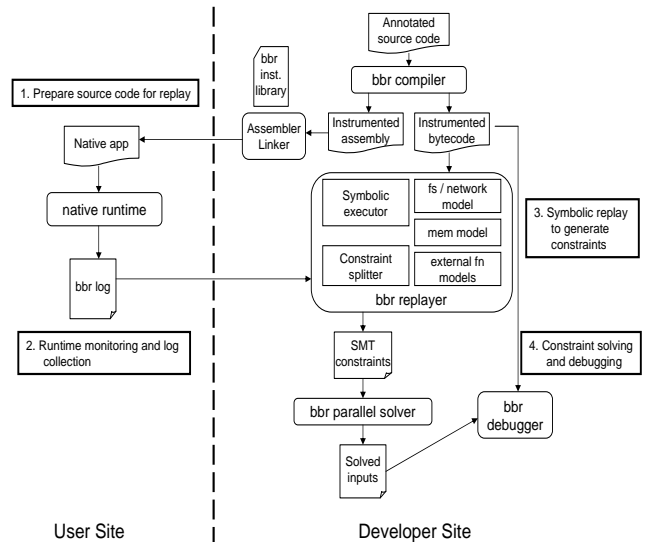


Figure 1: bbr system overview, highlighting steps in the replay process

2. OVERVIEW

bbr consists of five components: a bytecode compiler, a library for the instrumentation routines, a bytecode replayer, a parallel solver, and a debugger. Fig. 1 shows the steps involved in using bbr for replay.

We illustrate the operation of these components through a real-world bug in memcached. In version 1.2.6, a bug was found in decrementing an existing value in the cache [1]. The (slightly abridged) code is as follows:

```

char *do_add_delta (item *it, const int64_t delta) {
2622  int64_t value = ITEM_data(it);
    ...
2631  if (incr) {
    ...
2634  } else {
2635     value -= delta;
2636     if (value < 0) {
        value = 0;
    }
    MEMCACHED_COMMAND_DECR(ITEM_key(it), value);
}

```

The function first obtains the existing value for a key from the cache at line 2622 before performing the decrement. Then, if the resulting value is negative, the code simply sets the new value to be 0. This is a bug since `value` is treated as a signed entity, contrary to memcached’s specifications. If the existing value is a large number with the most significant bit set, then decrementing it will result in 0. In the following, we describe how a developer can use bbr to replay and debug this problem.

2.1 Preparing for Replay

To use bbr, the developer first prepares the source code by inserting the bbr checkpoint annotations. The annotation can be inserted in two ways: at a program point for which the developer would like to start replay from (e.g., at the point that processes each incoming requests), or configured as a timing parameter (e.g., take a checkpoint every five minutes). In our example, the developer can put the annotation at the program point that calls different functions (e.g., `do_add_delta`) based on the incoming request type.

The developer then uses the *bbr compiler* to compile the code into LLVM bytecode. The *bbr compiler* links the user code with a modified version of the *uClibc* [2] library (which we use because it can be easily compiled into LLVM bytecode). During compilation, the compiler inserts instrumentation routines to record two types of data. First, *bbr* records the values of conditional branches, in the example the branch conditions at lines 2631 and 2636. Next, *bbr* records the values of non-constant indices into arrays (none in this case). The instrumented code is then compiled to native assembly and linked with the *bbr instrumentation library* to produce the final native memcached distribution.

2.2 Runtime Monitoring

The user starts using the instrumented memcached server in her application. As the application issues operations against the memcached server, the *bbr* library embedded in the server intercepts the branch outcomes and array indices from the running program and stores them in an in-memory buffer, which is periodically flushed to disk when it becomes full. When the checkpoint annotation is hit, the library takes a lightweight snapshot of the running program. It also discards any logs that are previously written to disk. In our example, the user has been running her application normally until one day she observes that memcached returns 0 for a decrement and causes her application to return an incorrect result. Believing that it is a problem in the server, she files a report to the memcached developer along with the log generated by *bbr*.

2.3 Symbolic Replay

The developer would like to find out why 0 was returned in the decrement operation, which is not obvious by just looking at the code. With the collected log and the instrumented bytecode, the developer uses the *bbr replayer* to replay. The replayer symbolically executes the bytecode from the last checkpoint. In our example, at line 2622, `ITEM_data` loads the data from the heap pointed to by `it`, whose contents are unknown since it was stored prior to the checkpoint. When that happens, the replayer creates a fresh symbolic variable v to represent the contents at the memory pointed to by `it` and assign that to `value`. At the branch point in line 2631, the replayer reads the branch log and finds that the branch was not taken, and jumps to line 2634. At line 2635, `value` is assigned to be $v - d$, where the symbolic variable d represents the value of the program variable `delta`. Finally, at line 2636, from the branch log the replayer learns that the branch was taken, and because the branching condition involves symbolic variables, it generates the constraint $v - d < 0$ and continues execution.

2.4 Solving and Debugging

The replayer saves the generated constraints to a file at the end of the replay, which are then given to the *bbr parallel solver* to solve. In our example, the solver solves for possible values of v and d in order to produce a branch-deterministic execution from the last checkpoint. Suppose the solver returns $2^{63} + 1$ for v , and 1 for d . To map these results back to the program, *bbr* comes with a *debugger*. The debugger allows the developer to specify the values to be printed when an instruction is reached during symbolic execution. A sample session is as follows (italic text represents user inputs):

```
bbr-db> print delta at 2622; print value at 2622;
bbr-db> run
at 2622: delta = 1
at 2622: value = 9223372036854775809 // 263 + 1
```

With the output above and by following the program logic, the developer realizes the cause of the bug, and fixes her code accordingly. Even though the solved values might not be exactly the same as those that the user originally saw (in fact, the insertion of the key into the cache could have happened a long time ago with a different value in the original execution), the developer is still able to use *bbr* to diagnose the problem by finding an execution that leads to the behavior observed by the user.

The debugger also allows the developer to request a new feasible variable assignment if she isn't satisfied with a given assignment by appending extra constraints. In the example above, perhaps the developer knows that the value of `delta` was not 1, and thus the solution does not represent a feasible program state. In that case, she can tell the debugger that `delta` does not equal to 1, and ask the debugger to generate another set of values for `delta` and `value`.

In contrast, in order to use existing tools to replay the bug, the developer would either have to rely on the user to provide a detailed use case that illustrates the bug, or to collect data logs that have recorded the values of all non-deterministic data since memcached started (which could have been a long time ago), or to write assertions and rely on test generators to hit upon the bug, all of which are costly and error-prone.

In the next section, we discuss the details of the design of the *bbr* replayer.

3. BBR DESIGN

We begin by explaining the implementation of the recording mechanism, followed by that of the replayer and parallel constraint solver.

3.1 Instrumentation for Recording

As discussed in Section 2, calls to the instrumentation functions are added to the user code during compilation with the *bbr compiler* to collect the necessary data for replay. These functions are implemented in a separate library that is linked with the instrumented assembly when producing the native executable. Each conditional branch generates 1 bit of data, and other types of data (such as array indices) are 4 bytes each. To reduce disk overhead, we use a double-buffering mechanism where we log data in one buffer in memory. When the buffer fills (or the program terminates), we flush it to disk in the background and start recording into the other buffer. Section 4.1 reports the overhead of data collection. When a checkpoint annotation is reached, *bbr* records the current stack and the state of open files (see Section 3.2.4 for details) so that replay can start at the appropriate program point. The contents of registers and heap memory are not saved, making the operation very fast.

3.2 Replayer Design

Once data logs are collected, the developer feeds them to the replayer, which uses them to generate the constraints used to solve for inputs and program state consistent with the log. Using the branch log, the stream of bytecode instructions (Fig. 2) in the original execution is reconstructed,

PExpr (e_p) ::= $n \mid r \mid \text{unop } e_p \mid e_{p1} \text{ binop } e_{p2} \mid b ? e_{p1} : e_{p2}$
 CExpr (e_c) ::= $e_p \mid \text{BitVector} \mid (l, n)$
 Cond (b) ::= $\text{True} \mid \text{False} \mid \neg b \mid e_{p1} \text{ comp } e_{p2} \mid b_1 \wedge b_2 \mid b_1 \vee b_2$
 Stmt (s) ::= $r = \text{malloc}(n) \mid r = e_p \mid *r = e_p \mid r = *e_p \mid \text{assert } b$
 Program (p) ::= $s; p$
 $n \in \mathbf{N}, r \in \{\text{program variables}\}$
 $(l, n) \in \{\text{memory locations}\}, \text{unop} \in \{-, !\}$
 $\text{binop} \in \{+, -, *, /, \text{mod}, \text{band}, \text{bor}, \text{xor}, \text{shl}, \text{lshr}, \text{ashr}\}$
 $\text{comp} \in \{=, >, <, \geq, \leq, \neq\}$

We denote a memory location as (l, n) , where l represents an allocation, and n is the offset from the base of allocation.

Figure 2: Language of bytecode instructions

and instructions are emulated to generate constraints. The constraint generation process mirrors the symbolic execution done in concolic analysis (e.g., [18]); the main difference is that instead of getting our concrete data from the execution itself, we get it from the log. This subtle difference requires tradeoffs in terms of how much we can rely on symbolic vs. concrete reasoning, and has implications for the design of the symbolic state and constraint generation.

3.2.1 Modeling State

The most interesting aspect of the state maintained by the replayer is memory representation. Traditionally, the easiest way to model memory has been by using the theory of arrays, modeling memory as a giant array that gets updated with every write operation. This approach is very general, but puts too much strain on the SMT solver [35]. At the other extreme, if we know the concrete address of every load and store, we can model the heap as a map that gets modified by read and write operations [32, 18, 23, 34]. This is very efficient because the solver doesn't have to reason about aliasing, but it also requires a lot of concrete knowledge about the execution. In our system, we know the addresses for all memory accesses except those involving memory allocated before the start of replay, and the key to our approach is to leverage the concrete information we have rather than relying on the theory of arrays to discover it.

The first important element of our representation is that it models the heap as a collection of independent buffers produced by `malloc` rather than as a flat array. In this view, a pointer is a pair (l, n) , where l is the base address of a buffer assigned by `malloc`, and n is an offset into the buffer. The benefit of this is that during runtime we only need to record n and only when it is statically unknown in LLVM array element access instructions.¹ The main downside of this approach is that we assume pointer arithmetic to be able to change only n but not l , which prevents the system from reproducing errors caused by unsafe memory operations. In exchange for this, we are able to scale to large, long-running programs and to reproduce real bugs that are often much harder to discover than memory errors (see Section 4.2).

In addition to the heap $h : (l, n) \rightarrow e_c$, the replayer maintains an environment $\sigma : r \rightarrow e_c$ that maps variables to the symbolic values currently assigned to them, as well as an alias map $a : l \rightarrow \{l_0, \dots, l_k\}$ that stores may-alias locations

¹Field in records are accessed in the same way, however n is always statically known in such cases.

$\llbracket r = \text{malloc}(n) \rrbracket(\sigma, h, b, a) = (\sigma[r \rightarrow (l, 0)], h, b, a[l \rightarrow \{\}])$
 where l is a fresh memory location
 $\llbracket r = e_p \rrbracket(\sigma, h, b, a) = (\sigma[r \rightarrow \llbracket e_p \rrbracket], h, b, a)$
 $\llbracket \text{assert } b_1 \rrbracket(\sigma, h, b, a) = (\sigma, h, b \wedge b_1, a)$

Figure 3: Semantics for non-memory instructions

for a given allocation l . Finally, b represents the set of constraints accumulated so far in the execution. Together, the four entities (σ, h, b, a) make up the state of the replayer.

3.2.2 Symbolic Execution of Memory Operations

Our treatment of non-memory operations is similar to standard symbolic execution [32], as illustrated by Fig. 3. However, memory operations present new challenges. First of all, a read or a write to an address (l, n) is easy to handle if we know the concrete values of l and n , since we can just read or update the corresponding entry in our model of the heap. The challenge comes when we lack information about an address as a consequence of partial replay. The reason partial replay makes this difficult is because when we read a location that was written to before the start of the log, we know nothing about its contents, and if it contains an address, we have no information about either l or n , or about its aliasing relationship with other addresses.

Because of this, our system does not know the invariants the non-replayed execution enforced on memory contents. For the replayed portion of the program, bbr ensures that for every location (l, n) , and all its potentially aliased locations (l_i, n) , the following invariant is maintained:

$$\text{SolverValue}(l, n) = \text{SolverValue}(l_i, n) \Rightarrow \text{SolverValue}(h[(l, c)]) = \text{SolverValue}(h[(l_i, c)])$$

In other words, if the solver decides that l_i is actually aliased to l , then the symbolic expression in $h[(l, c)]$ should evaluate to the same value as the symbolic expression in $h[(l_i, c)]$.

In the following, we show this invariant is preserved in memory operations, and describe how we handle memory locations initialized before the start of partial replay.

Address Initialization: Consider reading from or writing to memory whose address is stored in variable r . If $\sigma[r]$ maps to some location (l, n) , then the operation proceeds as described below. Otherwise, r holds a value that was written before the start of partial replay (i.e., $\sigma[r] = \perp$). When this happens, the system assigns a fresh symbolic variable bv to represent r . Furthermore, it assigns bv a brand new base address l_r that is different from all the l 's currently in use by the program, updates $\sigma[r \rightarrow (l_r, 0)]$, and generates the constraint $bv = (l_r, 0)$. To account for the possibility of aliases, the system updates the alias map a to keep track of all the other l 's that may be aliased with l_r . Our current implementation assumes that two memory locations can be aliased to each other only if the program variables for which they were originally allocated to have the same static type, and that a memory location cannot be aliased to some offset within another location (e.g., array variables a and b can be aliased to each other, but a cannot be aliased to the addresses of $b[1]$, $b[2]$, etc, and vice versa). Notice that this is just one possible mechanism for computing alias information, which we chose as it proved sufficient for replaying a

variety of programs. Users can supply more precise alias information obtained elsewhere, e.g., from the output of an alias algorithm, or by logging all memory addresses in load and store instructions.

Memory Writes: Memory writes are processed according to the following rule:

$$\frac{h' = h[((l_j, c) \rightarrow (l_i) ? e_c : h[(l_j, c)]) \wedge ((l_i, c) \rightarrow b_i ? e_c : h[(l_i, c)])] \quad \{(l_j)\} = a[(l_i)] \quad \{(l_i, c, b_i)\} = \text{GetLoc}(\sigma[r])}{\langle *r = e_c, (\sigma, h, b, a) \rangle \rightarrow (\sigma, h', b, a)}$$

This rule conditionally updates every address that is potentially aliased with the address stored in r . The rule uses the function `GetLoc` to account for the fact that the address in r might itself have been the result of some conditional updates, so instead of a single address, `GetLoc` returns a guarded list of the possible addresses stored in $\sigma[r]$. If $\sigma[r]$ is a simple address (l_r, c) , `GetLoc` $(\sigma[r])$ returns (l_r, c, True) , but in general, $\sigma[r]$ can be a complicated expression evaluating to different (l_i, c_i) under different conditions b_i . However, since we logged the value of the offsets, the rule can assume that those are fixed for all choices. For each memory location (l_i, c) that is returned by `GetLoc`, we store a conditional expression that evaluates to e_c only if $*r$ does indeed evaluate to (l_i, c) ; i.e., if b_i is true. For each memory location l_j that is aliased with each l_i , we update $h[(l_j, c)]$ to a conditional that reduces to e_c if l_j is indeed aliased to l_i .

An important assumption in the rule above is that since we always know the offsets for memory accesses, we only have to track aliases among the base addresses. In the special case that an address is allocated with a call to `malloc` after the start of replay, there is no ambiguity as to what address is contained in a memory location, `GetLoc` simply returns one possible location for $*r$, and that location has no aliases, so the rule simply reduces to standard heap update:

$$\frac{\{(l_i, c_i, \text{True})\} = \text{GetLoc}(\sigma[r]) \quad h' = h[(l_i, c_i) \rightarrow e_c]}{\langle *r = e_c, (\sigma, h, b, a) \rangle \rightarrow (\sigma, h', b, a)}$$

Memory Reads: Loading from memory presents similar complications and is processed using the following rule:

$$\frac{\{(l_i, c_i, b_i)\} = \text{GetLoc}(e_c) \quad h[(l_i, c_i)] \neq \perp \quad \sigma' = \sigma[r \rightarrow \forall_i b_i \Rightarrow h[(l_i, c_i)]]}{\langle r = *e_c, (\sigma, h, b, a) \rangle \rightarrow (\sigma', h, b, a)}$$

The notation $\forall_i b_i \Rightarrow e_i$ above should be read as a switch statement that produces e_i if b_i is true. Note that we do not load from any of the may-alias locations of $*e_c$ since the store rule above ensures that their values will be equivalent if the alias turns out to be real.

In summary, in the case where we have the log from the start of execution, our strategy for memory operations reduces to the very efficient modeling of memory as a map. Conversely, when we have no information about aliasing and all our locations are unknown, our strategy is just an inefficient implementation of the theory of arrays. For our purposes, however, this is ideal because it exploits all the concrete information available with the system, and allows us to deal with aliasing uncertainty while still exploiting any aliasing information that we can provide.

3.2.3 External Function Calls

A common issue in the symbolic execution of real-world programs is handling external functions such as `libc` and system calls. The typical solution, and the one we follow, is to provide a model of those functions. `bbr` follows the approach used in `Klee` [18] by linking a modified version of the `uClibc` library with the target program, which greatly reduces the number of `libc` functions that we need to model. We model a subset of system calls using bit vector values. For instance, we use a 4 byte bit vector to model the user ID of the calling process. Thus calls to `getuid` will return that bit vector. We currently model over 100 Linux system calls but do not model the state of devices or the kernel.

3.2.4 File Descriptors and Network Sockets

`bbr` implements a simple symbolic file system where each file is modeled as follows:

- An array of symbolic expressions, where each entry in the array represents one byte in the file.
- A 4 byte integer for the current file offset.
- A 4 byte integer representing the file flags (read / write / both, and append / overwrite mode).
- A 4 byte integer representing the file permissions.
- A 4 byte integer representing the file size.

The file descriptor itself is modeled as a 4 byte bit vector variable, and the file system is a map from the file descriptors to the structure listed above. `bbr` currently supports typical file system calls, and network sockets are modeled similarly, except that they are append-only and do not support seek.

Because the file metadata are modeled as real integers (rather than bit vectors), `bbr` records values from file system operations (as mentioned in Section 2) in order to maintain each file's internal state during symbolic execution. For instance, the return values from `read` calls are recorded so that the current file offset can be correctly updated.

We experimented with using bit vectors and the theory of arrays to represent the file contents, but this did not scale, while the approach above did. This is because the programs we experimented with operate on large files, but do not necessarily process each byte within those files (e.g., databases may only read a few tuples from a loaded page). Modeling these files with the theory of arrays caused a large number of bit vector variables to be created for indices that are not referenced. Our approach, which records additional information during runtime (4 bytes per each invocation of `read`, `write`, `lseek`, etc), allows us to implement our own file model that generates less complicated constraints, which saves a substantial amount of constraint solving time while adding relatively little additional recording state.

3.3 Soundness and Completeness

For partial replay, we define soundness as the fact that all program executions (i.e., the starting heap state and execution path) solved by the replayer are indeed feasible traces of the target program although not necessarily the same as the original execution, and completeness as the ability to solve for program executions given the input logs, provided that the inputs were from an actual execution.

`bbr` is unsound since the initial state of the heap generated at the beginning of the partial trace might not correspond to a feasible heap from the original program. This is because the system might miss some of the invariants that the non-recorded portion of the program enforces on

the heap, so it might generate a heap that violates these unstated invariants. On the other hand, bbr is incomplete since we might not be able to replay executions that violate our assumptions, namely the independent allocation and non-overlapping assumption, the memory alias assumptions, and the fact that we do not model the hardware or the kernel state. In practice, that means we cannot determine the memory locations that are overwritten in a buffer overflow attack (although we can still detect writing out of allocated boundaries errors), or replay an execution that is caused by malfunctioning hardware.

3.4 Parallel Constraint Solver

Symbolic execution produces a set of constraints. The bit vector variables in the constraints correspond to heap contents before the start of replay, the non-deterministic inputs that were read, as well as return values any external function invocations during the execution. The constraints need to be solved in order to re-create the values that would cause the execution to take the same control flow path as in the original run. A straight-forward implementation would send all the generated constraints to a constraint solver to solve in one bundle. We found, however, that this is quite slow as the constraint sets can be very large.

Instead, we subdivide the constraints into independent portions that can be solved separately. Specifically, we found that although there are a large number of symbolic variables, it is frequently the case that each variable is only related to a handful of other variables. For instance, in replaying a web server, each byte that is read from the incoming socket is modeled as a bit vector variable as mentioned in Section 3.2.4. While it is likely that there will be constraints that bind the variables generated from reading the same socket (e.g., a constraint that the first three bytes read from the socket must be the characters ‘GET’), it is unlikely that there will be constraints that bind variables from one socket read to those from reads from different sockets, where each socket corresponds to a unique HTTP request.

Thus, bbr includes a constraints analyzer that separates the constraints into independent groups. Formally, the constraints form a graph $G(V, E)$, where each vertex $v \in V$ represents a bit vector variable, and each edge $e \in E$ between two vertices v_1 and v_2 represents a constraint binding the variables represented by v_1 and v_2 . Given G , the constraint analyzer processes E and splits G into its independent components. At the end of the analysis, each of the independent components are written to a file. The parallel solver in bbr then solves the components individually. Because most solvers are single-threaded, this also allows us to parallelize solving by running each component in a separate instance of the solver with a separate thread.

In addition to reducing solving time, splitting constraints into independent components can help identify components that may not need to be solved. For example, the constraints generated from replaying the `thttpd` web server (see Section 4) separate nicely into independent components as expected, with each component corresponding to an HTTP request. Most components take a relatively short time to solve, but one component takes substantially longer. Examining constraints involved in that component reveals that they involve not the bytes that are read from the request sockets, but instead bind the return values from multiple invocations of `gettimeofday` together. Looking at the code for

the server, we found that it implements a number of timers for house-keeping tasks such as clearing lingering connections. The timer structure maintains a field `time` that stores the next time that the timer should be fired. After processing each incoming connection, the server iterates through all timers to check if any of them should be run by passing in the current time (by calling `gettimeofday`). The check routine then compares the current time with the `time` field in the timers and either fires the timer or reschedules, creating dependencies between each invocation of the check (and thereby `gettimeofday`) routine. This routine runs many times per minute, and the solver is quite slow at solving it.

In this case, however, these constraints represent a small portion of the program which is unlikely to contain bugs, and not solving them reduces solving time substantially. In general, it is possible to use the outputs of any subset of the constraints to attempt to reconstruct a bug, thanks to our parallel solver. If one portion of the constraints is taking a long time to solve, the programmer can run the program on the values from the constraints that have been solved so far with the debugger to see if the bug is reproduced. In the case of the `gettimeofday` constraints described above, they don’t impact the results of interest (namely the contents of the incoming HTTP requests), so we do not need to wait for the solver to finish solving them.

3.5 Using the output of the solver

The output of the solver is an assignment to each variable involved in the constraints. bbr provides a debugger that takes in the instrumented bytecode and the solved outputs. Similar to the replayer, the debugger performs symbolic execution from the replay starting point. When an instruction that generates symbolic data is encountered, instead of creating symbolic variables, the debugger substitutes values from the solved outputs. Like `gdb`, the user can specify a list of values to be printed on the screen when they are executed.

For files and sockets that were used during execution, bbr further provides a utility that assembles the contents that were read from the solved output. bbr is able to do that due to the naming conventions of symbolic variables when they are created. Thus, if the developer is interested in viewing what values were read from a specific socket, she can first use the bbr debugger to identify the file descriptor number that the socket was assigned during replay, and then use the utility to assemble all the contents that were read from that descriptor number, rather than printing out the values of the descriptor number and the contents of the read buffer after each `recv` call.

As discussed in Section 2, if the user is not satisfied with the solved outputs, say because she has extra information about the original program state, she can provide that to the debugger in terms of extra constraints (e.g., some variable must have a certain value). The debugger then appends those constraints with the existing ones, and solves for another plausible program state if one exists.

4. EXPERIMENTS

In this section, we describe our evaluation of the three central claims of this paper:

- First, we compare the runtime overhead of bbr to other replay schemes. We measure both the overhead in terms of slowdown to the running program and the size of the logs generated.

sqlite	We ran 20k TPC-C [4] transactions on top of a 10G database representing 10 warehouses and measured the time to complete all transactions. Checkpoints are taken every 5 transactions.
memcached	We inserted or updated 10k keys into the cache, with values of size 800k each, and measured the time to complete all operations. Checkpoints are taken every 10 operations on the server.
tcpdump	We asked tcpdump to print the details from a packet dump of 1G in size, and measured the time to process all the packets. Checkpoints are taken every 5 packets that are processed.
betaftpd	We fetched 50 files from the server, each of size 1G, and measured the time to complete all operations. Checkpoints are taken every 5 fetches.
thttpd	(same as betaftpd)
ghttpd	(same as betaftpd)

Figure 4: Description of benchmarks

- Next, we chose different types of bugs from real-world applications and examine bbr’s ability to replay them.
- Finally, we measure the effectiveness of the alias-aware memory model and constraint partitioning in bbr in replaying programs.

bbr is implemented with 11k LOC of C++ using LLVM version 2.6. We experimented with several different SMT solver implementations, but we only report numbers from those using Z3 [3] as that was the solver that we found to scale the best. For these experiments, replay was run on a 32-bit Ubuntu machine with 2GB of RAM, while the constraint solving was done on a 64-bit 4-core machine running Z3 version 2.14 on Windows Server 2003 with 24GB of RAM.

4.1 Overhead Experiments

In the first experiment, we measured the time and space overhead that different programs incur while collecting logs with periodic checkpoints. The programs we replayed are listed in Fig. 4. Fig. 5 shows the results of the time overhead experiment, in terms of slowdown relative to an uninstrumented binary, while Fig. 6 and Fig. 7 show the log sizes that were generated by the different recording schemes.² All data collections were done using the bbr instrumentation library. We report the results from running five different versions of each application, namely:

- **[Native]**: Native uninstrumented version.
- **[bbr]**: bbr instrumented version with periodic checkpoints at frequencies shown in Fig. 4.
- **[non-det]**: Instrumented version that collects all non-deterministic data from the beginning of program execution. This models the overhead of a whole-program, fully deterministic replayer.
- **[snapshot]**: Same as non-det, except that every time a checkpoint annotation is reached, we fork a process to create a core dump, and all previously collected data are deleted. This models the overhead of a partial replayer that takes periodic snapshots of memory and logs non-deterministic inputs that were received between snapshots.
- **[loads]**: Instrumented version that records the values of all memory loads. To reduce the amount of logging,

²The checkpoints were placed at sensible program points, such as where the web server starts to process a new HTTP request. We tried placing the annotation at different but semantically similar locations and did not notice much difference in replay and solving times.

we use the technique from iDNA [17] where we only record loads whose values are not predictable at runtime (i.e., the store to a location was not an explicit instruction in the application but rather a side-effect of a system call, such as storing to the buffer in read).

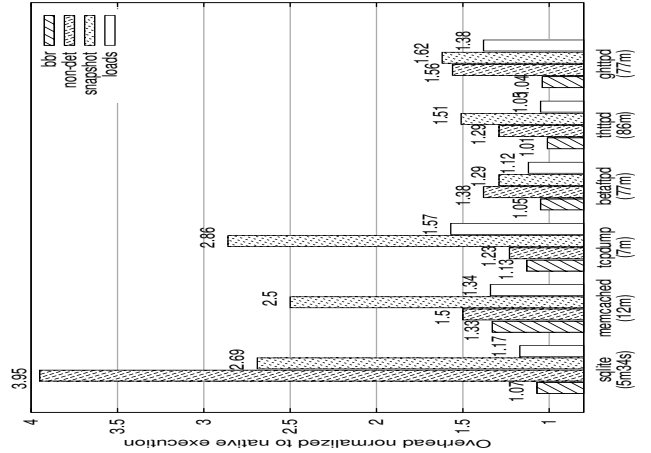


Figure 5: Normalized execution time (relative to uninstrumented binary=1) for different applications, with uninstrumented times shown at bottom.

Benchmark	bbr	non-det	snapshot	loads
sqlite	7k	20G	1G	250M
memcached	16k	8G	600M	569M
tcpdump	3k	2.6G	150M	200M
betaftpd	72M	50G	5.1G	550M
thttpd	100M	50G	5.1G	300M
ghttpd	99M	50G	5.2G	2.2G

Figure 6: Log size experiment results

The timing overhead experiment shows that bbr incurs the least overhead as compared to the other replay modes, varying from 1% for thttpd to 33% for memcached, with an overall average of 10%, which is significantly faster than the other schemes. This is due to the smaller amount of data that bbr needs to write as compared to the data-logging schemes. The more computationally intensive applications (e.g., sqlite) incur more overhead since they are more likely to hit branch instructions during their executions.

In terms of log sizes, bbr outperforms the other schemes by a significant amount. This is expected because for applications such as betaftpd, the majority of non-deterministic data comes from the contents of the files that are read and sent back to the user. So in the case of non-det, the size of the log is roughly equal to the size of files that were read. The core dump snapshots help somewhat, since all previously recorded non-deterministic data are discarded when a snapshot is taken. As the size of the core dump is directly proportional to the amount of memory that is currently being used by the program, for a computationally-intensive application such as web servers, the log size is dominated by the size of the core dump, whereas in data-intensive applications such as web servers, the log size is dominated by the non-deterministic data log. Finally, loads reduces log sizes by only recording the load values that are affected by system calls and subsequently read by the application. However, the log sizes are still larger than those

Benchmark	bbr	non-det	snapshot	loads
sqlite	1.4k/op	1M/op	0.2G/op	12.5k/op
memcached	1.6k/op	0.8M/op	60M/op	56.9k/op
tcpdump	0.6k/op	0.3M/op	30M/op	20k/op
betaftpd	14.4M/op	1G/op	1G/op	55M/op
thttpd	20M/op	1G/op	1G/op	6M/op
ghttpd	19.8M/op	1G/op	1G/op	44M/op

Figure 7: Log growth rates in overhead experiment

# Requests	bbr	non-det	snapshot	loads
10	46M	10G	5.1G	47M
30	46M	30G	5.3G	150M
50	46M	50G	5.6G	300M

Figure 8: Log sizes with varying # of requests

from bbr, since for some applications (such as sqlite and tcpdump), most of the values that are affected by system calls are indeed loaded by the application afterwards, for instance the read buffers in `read` and `recv` calls.

Since the number of operations stored in the various logs are not the same (e.g., bbr takes periodic checkpoints to reduce log sizes whereas non-det keeps the entire log from the beginning of execution), Fig. 7 presents the same results by showing the log growth rates instead. The results show that the bbr logs grow much slower as compared to others. Thus, even if the user decides to keep around all the bbr log files rather than having the system truncate them at each checkpoint (perhaps due to uncertainty as to which checkpoint the replay should start), bbr will still have the smallest log file sizes as compared to other schemes.

Next, we reran the experiment for betaftpd, varying the number of requests issued in order to characterize the size of the data logs as the running time of the application increases (the results show that the execution overhead is relatively insensitive to the number of requests). The results are shown in Fig. 8. Observe that the log sizes for the non-det and loads schemes grow proportionally with execution time due to the lack of log truncation. And while the log sizes are roughly the same for bbr and snapshot, bbr incurs much less runtime overhead because it writes less data to disk, as shown in Fig. 5.

To understand how these numbers compare to other replay tools, we compared our time overheads to those reported for other replay tools on similar data-intensive programs. PRES [31] reports an overhead of 43% when recording all basic blocks while running MySQL, and 1977% when recording PBZip2. R2 [24] reports an overhead of 200% when logging all data from win32 system calls while running MySQL. iTarget [37] reports a slowdown of 58% when performing whole program replay on BerkeleyDB. ODR [15] reports a slowdown of nearly 400% on MySQL. Other tools designed to collect data during normal execution on different benchmarks report similar or larger overheads. For example, program shepherding [28] reports up to 760% and iDNA [17] reports an average of 1189%. In summary, the overheads of bbr are generally low enough (with an average of 10%) to be deployed as an online tool during normal program execution.

4.2 Reproducing Bugs

In our next experiment, we used bbr to replay different types of bugs from real-world programs. Fig. 9 shows the list of bugs that we tried to replay. We describe a few representative bugs in detail in Section 4.2.1 below.

We set up the applications in a similar way as in the overhead experiment, except that we input a request that would expose or trigger the bug. We then terminated the appli-

cation (if it had not already terminated due to errors), and collected the logs. Given the log, we asked bbr to reproduce a plausible execution of the program from the last checkpoint. Fig. 10 shows some statistics about the bbr executions, specifically:

- **[LOC]**: Number of LLVM instructions emulated.
- **[# br]**: Number of conditional branch values recorded.
- **[Replay]**: Time taken for bbr’s symbolic execution.
- **[Split]**: Time taken to split the generated constraints into independent components.
- **[# constr]**: Total number of SMT constraints generated. Note that constraints that reduce to `True` are not included, thus the number of constraints can be smaller than number of conditional branches.
- **[# groups]**: Number of independent groups created.
- **[# vars]**: # of bit vector variables in the constraints.
- **[Solve]**: Time taken for parallel solving.
- **[Debug?]**: Whether the solved inputs can be used to identify the original bug.

These experiments show that bbr is able to symbolically execute a wide-range of real-world data-intensive applications and generate constraints that can be solved within a reasonable amount of time. Most of the replay time was spent in symbolic execution. Note that the number of conditional branch values recorded is typically much fewer than the number of constraints generated. That is because of peephole optimizations on the constraint expressions that bbr performs to eliminate constraints that reduces to `True`, and also because some of the conditional branches are not based on non-deterministic values in the program, so no constraints were generated.

4.2.1 Bug Walkthrough

Here we discuss a few bugs in detail. As discussed in Section 2, the decrement bug in memcached was caused by the incorrect treatment of the stored values as signed entities. Using bbr, we were able to solve for inputs and state of the value store that would trigger the problem. Notice that even though bbr did not record the contents of the request from the user, it was nonetheless able to reconstruct it given the branching decisions recorded during parsing of the request.

For the sqlite bugs, rather than a complete database file, the output of the solver is a combination of heap values and a partial database file that was read while processing the transactions since the last checkpoint. As described in Section 3.5, these outputs cannot be used directly to run sqlite and reproduce the bug, since the solved input file is incomplete. However, these solved values can still help developers debug. For instance, the collation bug was caused by an error in the implementation of the comparison operator between two expression trees, causing it to return the wrong cached value rather than performing the actual comparison. Using the solved values from the partial replay and the branch log, we were able to observe the error by tracking the solved values that were used as inputs to the comparison function, thanks to the bbr debugger. When invoked with those values, the function returns the incorrect value that triggers the bug.

For the bugs in tcpdump, bbr solved for the contents of the incoming packets that cause the infinite loops. While the contents were not exactly identical to the original ones, we have verified that they cause the same problems when fed into tcpdump.

sqlite cast	Error in the processing of queries with multiple selection predicates when one of the predicates involves a cast. The code mistakenly reuses the results from the predicate without the cast for those with the cast [5].
sqlite join	Error in processing natural self-joins. Mistake in identifying the table to join leads to returning extra rows [6].
sqlite collate	In the processing of aggregates, the code mistakenly treats aggregates with collation and without collation as equivalent. As a result, instead of executing each aggregate the cached results are returned [7].
memcached decr	Error in treating unsigned entities as signed leads to the wrong value being returned as a result of decrement [1].
memcache CAS	When a part of an existing value is updated, the code forgets to update its CAS (a unique identifier for the value), leading to the same CAS being returned both before and after the update, violating the specification [8].
tcpdump BGP	The code for displaying BGP packets contains an error in the checking of the loop condition that always returns true. As a result, the loop never terminates [9].
tcpdump ISIS	In printing ISIS packets, the code does not update how many bytes have been processed. Because of that, the printing code goes into an infinite loop. This is a different bug than the one above [10].
tcpdump RSVP	The code for displaying RSVP packets does not check if the end of packet has been reached. As a result, the printing code goes into an infinite loop. This is a different bug than those above [11].
betaftpd char	Betaftpd does not properly handle non-Latin characters (such as ‘á’). As a result, it returns not found when a user requests a file with such characters as filename, even though the file exists.
thttpd defang	When returning an error message to a http request, the code expands ‘>’ and ‘<’ characters into their HTML equivalents. However, it forgets to check if it has already reached the end of the buffer allocated to hold the escaped error message, thus leading to a write out of bounds error [12].
ghttpd CGI	When generating the pathname for a CGI, the code simply concatenates the names of the CGI directory and the CGI executable together, without checking whether the length of the result exceeds the preallocated buffer [13].
ghttpd log	The request logging code in ghttpd does not check the length of the filename in the GET request. As a result, it can write into memory that is beyond the size of the buffer preallocated to hold the log message [14].

Figure 9: Description of bugs replayed

Bug	LOC	# br	Replay	Split	# constr	# groups	# vars	Solve	Debug?
sqlite cast	2420548	83883	1225s	2s	86245	2140	11320	5hr	Y
sqlite join	1705078	149762	1011s	4s	71596	2891	11127	4hr	Y
sqlite collate	2546778	132084	820s	2s	67890	1272	9750	3hr	Y
memcached decr	8708	1744	286s	0.67s	811	158	1342	13s	Y
memcached CAS	24837	2249	1955s	3.74s	1705	265	2483	158s	Y
tcpdump BGP	2947998	247161	31s	0.08s	13476	147	362	1s	Y
tcpdump ISIS	61769562	4634390	370s	0.65s	235081	86	228	5s	Y
tcpdump RSVP	1108294	83341	93.8s	0.03s	11541	233	233	1s	Y
betaftpd char	122036	9869	7s	0.01s	950	106	358	1s	N
thttpd defang	514879	55521	542s	1.06s	21003	2153	6105	2s	Y
ghttpd CGI	352710	31297	40s	0.01s	8553	728	784	2s	Y
ghttpd log	344662	45054	40s	0.01s	8570	698	753	1s	Y

Figure 10: Replay experiment results

In general, for all of the bugs that bbr was able to replay, even though the solved instances of the database contents / network packets / http requests / etc are not necessarily the exact same one as in the original input, they still exhibit the errors. This illustrates the power of our approach – recording branches and a few extra pieces of data is enough for developers to debug the problem in hand. Also, observe that without using bbr’s partial replay ability, in order to replay these bugs it would be necessary to record all non-deterministic data from the beginning, or take periodic snapshots of the heap, which would be very costly as demonstrated. In both cases, there is a huge overhead in both time and data log sizes. Furthermore, as our experiment shows, the longer the program runs, the worse the problem becomes.

bbr failed to replay the betaftpd non-Latin character bug. This is because the bug was caused by the lack of checking the characters that are read from the incoming socket before passing them to `open`. The constraints generated from symbolic execution simply bound the length of the file requested (since the code checks for ‘\0’), but not the contents that were read (this is precisely the reason why the bug exists). As a result, the solver returned a random filename that does not trigger the bug when re-executed. While the developer can use the bbr debugger to request another plausible value for the filename, it is unlikely that she will be able to hit upon one that causes the problem within a reasonable amount of time given the unconstrained nature of the bug. This illustrates a limitation of bbr – the ability to recon-

struct a trace that leads to the problem relies on the fact that the buggy execution is sufficiently bound by the bbr logs. Otherwise, the solver might return a trace that does not trigger the error, even though it is branch-deterministic.

Finally, we note that for a few of these bugs (e.g., those from ghttpd), it is possible to replay them by just recording the incoming requests after the last checkpoint, since they do not involve the internal state of the program. However, it would be very difficult for a developer to know a priori whether the bugs that her program exhibits will involve the internal state of the application or not, unless the application keeps no internal state whatsoever, which is highly unlikely. For instance, the sqlite and memcached bugs are related to the internal state of the program (in particular, the cached values in memory). Thus, it would not be possible to replay such bugs simply by recording incoming requests.

4.3 Constraint Splitting

Next, we look at the effectiveness of splitting constraints into independent groups, using ghttpd and betaftpd as examples. We replayed different number of requests coming into the two servers and measured the solving time. We chose these two applications since they have relatively simple internal states, and each incoming request should be independent from each other in terms of the constraints that are generated. We compared the time taken to solve the generated constraints all at once from a single file (Single), and the total time needed to solve the split constraints, using 1000 threads for the parallel solver on a 4 core machine (Split).

# reqs	# const.	# vars	# groups	single	split
ghhttpd web server					
5	8547	750	698	8s	1s
10	20383	1923	823	20s	5s
50	60384	5330	2314	32m	6m
betaftpd ftp server					
5	960	358	106	2s	1s
10	1534	533	210	40s	10s
50	13223	1903	530	40m	13m

Figure 11: Constraint splitting experiment results

The results of the experiment are shown in Fig. 11.

The results show a huge difference between parallel and single-threaded solving. We believe that one possible reason for the substantial slowdown in the single-threaded case is because of cache misses when the input problem size becomes large. We have also noticed that the solver frequently suffers from memory thrashing in the single-threaded case (on a machine with 24GB of physical RAM), which further contributes to the slowdown. While one might argue that this is a relatively obvious way to speed up solving, we have compared our results with different solver implementations (Z3, yices, stp), and none of them seem to have this feature in place, possibly because constraint solvers are traditionally targeted to solving instances where most variables are related, but not instances with a large number of independent components. Given that the time needed to do constraint splitting is relatively small as compared to the replay time (as shown in Fig. 10), it appears worthwhile to use parallel solving for replay.

4.4 Memory Model Implementation

Lastly, we looked at the performance differences between different memory models in our replay engine. As discussed in Section 3.2, bbr models memory as a conceptual mapping from program variables to constraint expressions. Internally, this is implemented using two maps, one that maps program variables to memory locations, and one that maps memory locations to contents in memory. There are different ways to implement the second map. We experimented with several different implementations for a 32-bit memory (i.e., addresses are 32-bit each) and compared the time it took to symbolically execute three different small programs:

1. **[wc]**: word count on an empty file (37k LLVM LOC).
2. **[db connect]**: a simple application that connects to a sqlite database (66k LLVM LOC).
3. **[db query]**: application that connects and issues a select query to a sqlite database (341k LLVM LOC).

Debug printouts were turned on so they took longer to run. The different memory model implementations include:

1. A single Z3 bit vector array [Z3 bit vector(8) \rightarrow Z3 bit vector(8)]. This is the most general representation based on the theory of arrays.
2. For an allocation site of n bytes, create a Z3 bit vector of length $8n$, with the entire memory modeled as a `std::map [uint \rightarrow Z3 bit vector($8n$)]`.
3. A single `std::map [uint \rightarrow Z3 bit vector(8)]`.
4. A Z3 bit vector array for each allocation site, with the entire memory modeled as a single `std::map [uint \rightarrow Z3 bit vector array]`.
5. For an allocation site of n bytes, create a `std::vector` of size n , with each entry being an 8-bit Z3 bit vector to model a byte, and the entire memory is modeled as a single `std::map [uint \rightarrow std::vector of Z3 bit vector(8)]`.

Impl. #	wc	db connect	db query
1	13m	52m	>1hr
2	>1hr	crashed	crashed
3	8m	15m	>1hr
4	15m	35m	>1hr
5	47s	1.9m	17m
6	10s	42s	2m

Figure 12: Memory modeling experiment results

6. We built our own data structure to represent a bit vector and implemented a number of peephole optimizations. A single `std::map [custom bit vector(8) \rightarrow custom bit vector(8)]` is then used to model memory. This is the implementation we used for the operations described in Section 3.2.2.

Fig. 12 reports different memory model implementations and their replay times. “crashed” means the experiment exhausted the address space on a 32-bit machine and crashed. As the experiment shows, using the Z3 theory of arrays incurs a substantial replay overhead in real programs, since each read / write to the array potentially creates a new array data structure. Using Z3 bit vectors still incurs some overhead in the solver, possibly due to housekeeping routines such as structural hashing. Note that the programs above were replayed from the entry point and no memory aliases were involved. We anticipate that the performance of the first 5 implementations to perform much worse in partial replay with memory aliases, and thus built our own representation of expressions at the end.

5. RELATED WORK

A number of bug replay tools based on symbolic execution have been proposed recently. However, to our knowledge, no tools support partial symbolic replay for long running programs, which fundamentally distinguishes our approach. In [19], the authors proposed to record input traces during execution, and perform symbolic execution to generate bug cases for the developer. However, they used a trace logger with high overhead [17], and it is not clear if their approach is scalable to long-running programs. ESD [40] uses a path searching approach similar to model checking to reproduce bugs. It has no runtime overhead, but requires a coredump, makes no guarantee regarding how long it would take to regenerate the bug, and is tricky to use for replaying non-crash or hang bugs. ODR [15] targets multicore bugs and collects data traces during normal execution. They report a substantial runtime overhead because of data collection and also a long solving time. PRES [31] is similar to ODR but reproduces bugs iteratively with significant slowdowns.

Replaying executions deterministically from data logs has been a topic in systems research for many years, with the emphasis on using different mechanisms to reduce the runtime overhead and log sizes [30, 33, 22, 17, 25] and recording at the operating systems level rather than application level [27]. Recent tools such as R2 [24] and iTarget [37] allow users to decide what data to be logged in order to reduce recording overhead, but at the expense of not being able to reproduce all executions. These tools generally have higher runtime overheads for data-intensive programs than bbr.

At the other end are techniques that automatically find bugs without any inputs. These range from static analysis tools [16, 21, 29] to dynamic tools such as model checkers [36, 38]. Advances in SMT solvers and symbolic execution have enabled new test generation tools such as concolic testing [32, 34] and other test generators based on symbolic ex-

ecution [23, 18]. These tools focus on finding all sorts of bugs, rather than reproducing a specific bug in hand.

Our memory model is inspired by the work done in test generation for languages that include pointer operations. In [35], the authors proposed a memory model that maintains heaps for different types of objects and introduces constraints to enforce the disjointness among the different object heaps. bbr imposes similar assumptions on object types and their alias properties. In [20], the authors proposed a region-based approach, with each address represented by a memory location object similar to that in bbr, and each location can point to different concrete addresses depending on alias resolution. In [39], a hybrid memory model was proposed with a concrete store and a symbolic store implemented using theory of arrays, which led to our initial memory model implementation. There is similar work in languages without explicit pointer constructs such as [26].

6. CONCLUSIONS

We presented bbr, a *branch-deterministic* replayer that performs low-overhead replay by logging only a small amount of data at runtime. We also proposed the notion of *partial replay*, which allows users to replay long running data-intensive programs with bounded log sizes. To make our approach scale, we introduced several new techniques, including a new memory model for handling aliased addresses, and a partitioning approach for parallel solving. We showed that these techniques allow bbr to efficiently replay a number of data-intensive programs, and can reproduce real bugs in long-running programs, including sqlite and several web servers.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. We also thank Lidong Zhou, Vijay Ganesh, and Nikolaj Bjørner for their helpful discussions.

References

- [1] <http://code.google.com/p/memcached/issues/detail?id=21>.
- [2] <http://www.uclibc.org>.
- [3] <http://research.microsoft.com/en-us/um/redmond/projects/z3/>.
- [4] <http://www.tpcc.org>.
- [5] <http://www.sqlite.org/src/info/eb5548a849>.
- [6] <http://www.sqlite.org/src/info/b73fb0bd64>.
- [7] <http://www.sqlite.org/src/info/360c6073e197>.
- [8] <http://code.google.com/p/memcached/issues/detail?id=15>.
- [9] CVE-2005-1279.
- [10] CVE-2005-1278.
- [11] CVE-2005-1280.
- [12] CVE-2003-0899.
- [13] CVE-2001-0820.
- [14] CVE-2002-1904.
- [15] G. Altekar and I. Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP*, pages 193–206, 2009.
- [16] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.
- [17] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 154–163, 2006.
- [18] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
- [19] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS*, pages 319–328, 2008.
- [20] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, pages 129–140, 2009.
- [21] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.
- [22] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conference*, pages 289–300, 2006.
- [23] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
- [24] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI*, pages 193–208, 2008.
- [25] J. Huang, P. Liu, and C. Zhang. Leap: lightweight deterministic multi-processor replay of concurrent java programs. In *SIGSOFT FSE*, pages 207–216, 2010.
- [26] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.
- [27] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, pages 1–15, 2005.
- [28] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, pages 191–206, 2002.
- [29] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: explaining program failures via postmortem static analysis. In *SIGSOFT FSE*, pages 63–72, 2004.
- [30] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI*, pages 313–325, 1994.
- [31] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, pages 177–192, 2009.
- [32] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.
- [33] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, pages 29–44, 2004.
- [34] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *TAP*, pages 134–153, 2008.
- [35] D. Vanoverberghe, N. Tillmann, and F. Piessens. Test input generation for programs with pointers. In *TACAS*, pages 277–291, 2009.
- [36] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA*, pages 97–107, 2004.
- [37] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *FSE-18*, 2010.
- [38] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *NSDI*, pages 213–228, 2009.
- [39] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, pages 243–257, 2006.
- [40] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys*, pages 321–334, 2010.