Parallel Symbolic Execution for Automated Real-World Software Testing

Stefan Bucur Vlad Ureche Cristian Zamfir George Candea

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{stefan.bucur,vlad.ureche,cristian.zamfir,george.candea}@epfl.ch

Abstract

This paper introduces Cloud9, a platform for automated testing of real-world software. Our main contribution is the scalable parallelization of symbolic execution on clusters of commodity hardware, to help cope with path explosion. Cloud9 provides a systematic interface for writing "symbolic tests" that concisely specify entire families of inputs and behaviors to be tested, thus improving testing productivity. Cloud9 can handle not only single-threaded programs but also multi-threaded and distributed systems. It includes a new symbolic environment model that is the first to support all major aspects of the POSIX interface, such as processes, threads, synchronization, networking, IPC, and file I/O. We show that Cloud9 can automatically test real systems, like memcached, Apache httpd, lighttpd, the Python interpreter, rsync, and curl. We show how Cloud9 can use existing test suites to generate new test cases that capture untested corner cases (e.g., network stream fragmentation). Cloud9 can also diagnose incomplete bug fixes by analyzing the difference between buggy paths before and after a patch.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Symbolic Execution

General Terms Reliability, Verification

1. Introduction

Software testing is resource-hungry, time-consuming, labor-intensive, and prone to human omission and error. Despite massive investments in quality assurance, serious code defects are routinely discovered after software has been released [RedHat], and fixing them at so late a stage carries substantial cost [McConnell 2004]. It is therefore imperative to overcome the human-related limitations of software testing by developing automated software testing techniques.

Existing automated techniques, like model checking and symbolic execution, are highly effective [Cadar 2008, Holzmann 2008], but their adoption in industrial general-purpose software testing has been limited. We blame this gap between research and practice on three challenges faced by automated testing: scalability, applicability, and usability.

First, path explosion—the fact that the number of paths through a program is roughly exponential in program size—severely limits the extent to which large software can be thoroughly tested. One must be content either with low coverage for large programs, or apply automated tools only to small programs. For example, we do not know of any symbolic execution engine that can thoroughly test systems with more than a few thousand lines of code (KLOC).

Second, real-world systems interact heavily with the environment (e.g., through system calls, library calls) and may communicate with other parties (e.g., over sockets, IPC, shared memory). For an automated testing tool to be used in practice, it must be capable of handling these interactions. Third, an important hurdle to adoption is that, in order to productively use most current tools, a user must become as versed in the underlying technology as the tool's developers.

Our goal in building Cloud9 is to address these challenges: we envision Cloud9 as a testing platform that bridges the gap between symbolic execution and the requirements of automated testing in the real world. As will be seen later, doing so requires solving a number of research problems.

Cloud9 helps cope with path explosion by parallelizing symbolic execution in a way that scales well on large clusters of cheap commodity hardware. Cloud9 scales linearly with the number of nodes in the system, thus enabling users to "throw hardware at the problem." Doing so without Cloud9 is hard, because single computers with enough CPU and memory to symbolically execute large systems either do not exist today or are prohibitively expensive.

We built into Cloud9 features we consider necessary for a practical testing platform: Besides single-threaded single-node systems, Cloud9 handles also multi-threaded and dis-

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

tributed software. It offers fine grain control over the behavior being tested, including the injection of faults and the scheduling of threads. Cloud9 embeds a new symbolic model of a program's environment that supports all major aspects of the POSIX interface, including processes, threads, synchronization, networking, IPC, and file I/O. Cloud9 provides an easy-to-use API for writing "symbolic tests"—developers can specify concisely families of inputs and environment behaviors for which to test the target software, without having to understand how symbolic execution works, which program inputs need to be marked symbolic, or how long the symbolic inputs should be. By encompassing entire families of behaviors, symbolic tests cover substantially more cases than "concrete" regular tests.

This paper makes three contributions: (1) the first cluster-based *parallel symbolic execution* engine that scales linearly with the number of nodes; (2) a *testing platform* for writing symbolic tests; and (3) a quasi-complete *symbolic POSIX model* that makes it possible to use symbolic execution on real-world systems. We built a Cloud9 prototype that runs on Amazon EC2, private clusters, and multicore machines.

In this paper we present Cloud9 and report on our experience using it for testing several parallel and distributed systems, describe some of the bugs found, and explain how we debugged flawed bug fixes. We describe the problem (§2), Cloud9's design (§3-§5), our prototype (§6), we evaluate the prototype (§7), survey related work (§8), and conclude (§9).

2. Problem Overview

Testing a program consists of exercising many different paths through it and checking whether they "do the right thing." In other words, testing is a way to produce partial evidence of correctness, and thus increase confidence in the tested software. Yet, due to the typically poor coverage one can get today, testing often turns into a mere hunt for bugs.

In practice, most software test harnesses consist of manually written tests that are run periodically; regression test suites provide an automated way of checking whether new bugs have entered the code [McConnell 2004]. Such suites tend to be tedious to write and maintain but, once in place, they can be reused and extended. In practice, the state of the art consists mostly of fuzzing, i.e., trying various inputs in the hope of finding bugs and improving test coverage.

In research, the state of the art consists of model checkers and automated test generators based on symbolic execution [Cadar 2008, Godefroid 2005]. Instead of running a program with regular concrete inputs (e.g., $x = 5$), symbolic execution consists of running a program with "symbolic" inputs that can take on all values allowed by the type (e.g., $x = \lambda$, where $\lambda \in \mathbb{N}$). Whenever a conditional branch is encountered that involves a predicate π that depends (directly or indirectly) on x, state and execution are forked into two alternatives: one following the then-branch (π) and another following the else-branch ($\neg\pi$). The two executions can now be pursued independently. When a bug is found, test generators can compute concrete values for program inputs that take the program to the bug location. This approach is efficient because it analyzes code for entire classes of inputs at a time, thus avoiding the redundancy inherent in fuzzing.

The *first challenge* for such tools is path explosion, as mentioned earlier. One way to cope is to memoize the symbolic execution of sub-paths into test summaries that can be subsequently reused when the same sub-path is encountered again, as done in compositional test generation [Godefroid 2007]. Alternatively, it is possible to use various heuristics to prioritize the most interesting paths first, as done in KLEE [Cadar 2008]. Another approach is to execute symbolically only paths that are of interest to the test, as done in selective symbolic execution [Chipounov 2009].

We pursue a complementary approach—*parallel symbolic execution*—in which we symbolically execute a program in parallel on a cluster, thus harnessing the machines into a "distributed computer" whose aggregate CPU and memory surpass that of an individual machine. An alternative to a cluster-based approach would be to run a classic single-node symbolic execution engine on a Blue Gene-like supercomputer with vast shared memory and CPUs communicating over MPI. Supercomputers, however, are expensive, so we favor instead clusters of cheap commodity hardware.

One way to parallelize symbolic execution is by statically dividing up the task among nodes and having them run independently. However, when running on large programs, this approach leads to high workload imbalance among nodes, making the entire cluster proceed at the pace of the slowest node [Staats 2010]. If this node gets stuck, for instance, while symbolically executing a loop, the testing process may never terminate. Parallelizing symbolic execution on shared-nothing clusters in a way that scales well is difficult.

The *second challenge* is mediating between a program and its environment, i.e., symbolically executing a program that calls into libraries and the OS, or communicates with other systems, neither of which execute symbolically. One possible approach is to simply allow the call to go through into the "concrete" environment (e.g., to write a file) [Cadar 2006, Godefroid 2005]; unfortunately, this causes the environment to be altered for *all* forked executions being explored in parallel, thus introducing inconsistency. Another approach is to replace the real environment with a symbolic model, i.e., a piece of code linked with the target program that provides the illusion of interacting with a symbolically executing environment. For instance, KLEE uses a symbolic model of the file system [Cadar 2008]. Of course, real-world programs typically interact in richer ways than just file I/O: they fork processes, synchronize threads, etc.

We originally viewed the building of a complete environment model as an engineering task, but our "mere engineering" attempt failed: for any functionality that, in a normal execution, requires hardware support (such as enforcing iso-

lation between address spaces), the core symbolic execution engine had to be modified. The research challenge therefore is to find the minimal set of engine primitives required to support a rich model of a program's environment.

The *third challenge* is using an automated test generator in the context of a development organization's quality assurance processes. To take full advantage of the automated exploration of paths, a testing tool must provide ways to control all aspects of the environment. For example, there needs to be a clean API for injecting failures at the boundary between programs and their environment, there must be a way to control thread schedules, and so on. There should be a way to programmatically orchestrate all environment-related events, but doing so should not require deep expertise in the technology behind the testing tools themselves.

The work presented here aims to address these three challenges. Cluster-based parallel symbolic execution (§3) provides the illusion of running a classic symbolic execution engine on top of a large, powerful computer. Without changing the exponential nature of the problem, parallel symbolic execution harnesses cluster resources to make it feasible to run automated testing on larger systems than what was possible until now. Our work complements and benefits all tools and approaches based on symbolic execution. We describe a way to accurately model the environment (§4) with sufficient completeness to test complex, real software, like the Apache web server and the Python interpreter. We present the APIs and primitives that we found necessary in developing a true testing platform (§5). We show how using these APIs enables, for instance, finding errors in bug patches by reproducing environment conditions which otherwise would have been hard or impossible to set up with regular test cases.

3. Scalable Parallel Symbolic Execution

In this section we present the design of the Cloud9 engine, focusing on the algorithmic aspects: after a conceptual overview (§3.1), we describe how Cloud9 operates at the worker level (§3.2) and then at the cluster level (§3.3).

3.1 Conceptual Overview

Classic Symbolic Execution Cloud9 employs symbolic execution, an automated testing technique that has recently shown a lot of promise [Cadar 2008, Godefroid 2005].

A symbolic execution engine (SEE) executes a program with unconstrained symbolic inputs. When a branch involves symbolic values, execution forks into two parallel executions (see §2), each with a corresponding clone of the program state. Symbolic values in the clones are *constrained* to make the branch condition evaluate to false (e.g., $\lambda \geq$MAX) respectively true (e.g., $\lambda <$MAX). Execution recursively splits into sub-executions at each subsequent branch, turning an otherwise linear execution into an *execution tree* (Fig. 1).

In this way, all execution paths in the program are explored. To ensure that only feasible paths are explored,

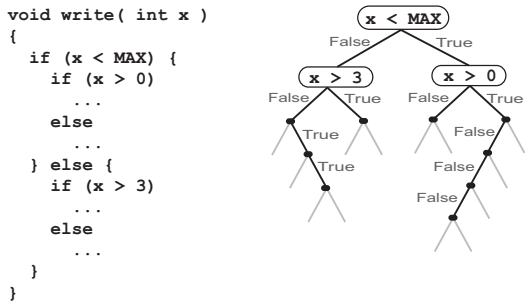


Figure 1: Symbolic execution produces an execution tree.

the SEE uses a constraint solver to check the satisfiability of each branch's predicate, and it only follows satisfiable branches. If a bug is encountered (e.g., a crash or a hang) along one of the paths, the solution to the constraints accumulated along that path yields the inputs that take the tested program to the bug—these inputs constitute a *test case*.

Parallel Symbolic Execution Since the size of the execution tree is exponential in the number of branches, and the complexity of constraints increases as the tree deepens, state-of-the-art SEEs can quickly bottleneck on CPU and memory even for programs with just a couple KLOC. We therefore build a parallel SEE that runs on a commodity cluster and enables "throwing hardware at the problem."

The key design goal is to enable individual cluster nodes to explore the execution tree independently of each other. One way of doing this is to statically split the execution tree and farm off subtrees to worker nodes. Alas, the contents and shape of the execution tree are not known until the tree is actually explored, and finding a balanced partition (i.e., one that will keep all workers busy) of an unexpanded execution tree is undecidable. Besides subtree size, the amount of memory and CPU required to explore a subtree is also undecidable, yet must be taken into account when partitioning the tree. Since the methods used so far in parallel model checkers [Barnat 2007, Holzmann 2008] rely on static partitioning of a finite state space, they cannot be directly applied to the present problem. Instead, Cloud9 partitions the execution tree *dynamically*, as the tree is being explored.

Dynamic Distributed Exploration Cloud9 consists of worker nodes and a load balancer (LB). Workers run independent SEEs, based on KLEE [Cadar 2008]. They explore portions of the execution tree and send statistics on their progress to the LB, which in turn instructs, whenever necessary, pairs of workers to balance each other's work load. Encoding and transfer of work is handled directly between workers, thus taking the load balancer off the critical path.

The goal is to dynamically partition the execution tree such that the parts are *disjoint* (to avoid redundant work) and together they *cover* the global execution tree (for exploration to be complete). We aim to minimize the number of work transfers and associated communication overhead. A fortuitous side effect of dynamic partitioning is the transparent

handling of fluctuations in resource quality, availability, and cost, which are inherent to large clusters in cloud settings.

Cloud9 operates roughly as follows: The first component to come up is the load balancer. When the first worker node W_1 joins the Cloud9 cluster, it connects to the LB and receives a "seed" job to explore the entire execution tree. When the second worker W_2 joins and contacts the LB, it is instructed to balance W_1's load, which causes W_1 to break off some of its unexplored subtrees and send them to W_2 in the form of *jobs*. As new workers join, the LB has them balance the load of existing workers. The workers regularly send to the LB status updates on their load in terms of exploration jobs, along with current progress in terms of code coverage, encoded as a bit vector. Based on workers' load, the LB can issue job transfer requests to pairs of workers in the form $\langle$ source worker, destination worker, # of jobs $\rangle$. The source node decides which particular jobs to transfer.

3.2 Worker-level Operation

A worker's visibility is limited to the subtree it is exploring locally. As W_i explores and reveals the content of its local subtree, it has no knowledge of what W_j's ($i \neq j$) subtree looks like. No element in the system—not even the load balancer—maintains a global execution tree. Disjointness and completeness of the exploration (see Fig. 2) are ensured by the load balancing algorithm.

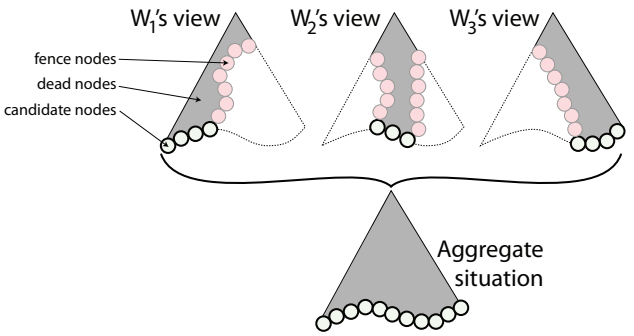


Figure 2: Dynamic partitioning of exploration in Cloud9.

As will be explained later, each worker has the root of the global execution tree. The tree portion explored thus far on a worker consists of three kinds of nodes: (1) internal nodes that have already been explored and are thus no longer of interest—we call them *dead* nodes; (2) *fence* nodes that demarcate the portion being explored, separating the domains of different workers; and (3) *candidate* nodes, which are nodes ready to be explored. A worker exclusively explores candidate nodes; it never expands fence or dead nodes.

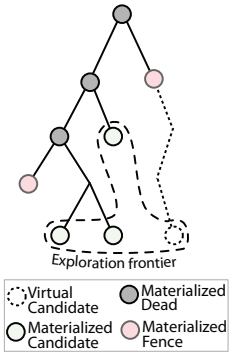
Candidate nodes are leaves of the local tree, and they form the *exploration frontier*. The work transfer algorithm ensures that frontiers are disjoint between workers, thus ensuring that no worker duplicates the exploration done by another worker. At the same time, the union of all frontiers in the system corresponds to the frontier of the global execution tree. The goal of a worker W_i at every step is to choose the next candidate node to explore and, when a bug is encountered, to compute the inputs, thread schedule, and system call returns that would take the program to that bug.

The implementation of this conceptual model lends itself to many optimizations, some of which we cover in §6. Broadly speaking, judicious use of copy-on-write and a novel state-encoding technique ensure that actual program state is only maintained for candidate and fence nodes.

Worker-to-Worker Job Transfer When the global exploration frontier becomes poorly balanced across workers, the load balancer chooses a loaded worker W_s and a less loaded worker W_d and instructs them to balance load by sending n jobs from W_s to W_d. In the extreme, W_d is a new worker or one that is done exploring its subtree and has zero jobs left.

W_s chooses n of its candidate nodes and packages them up for transfer to W_d. Since a candidate node sent to another worker is now on the boundary between the work done by W_s and the work done by W_d, it becomes a fence node at the sender. This conversion prevents redundant work.

A job can be sent in at least two ways: (1) serialize the content of the chosen node and send it to W_d, or (2) send to W_d the path from the tree root to the node, and rely on W_d to "replay" that path and obtain the contents of the node. Choosing one vs. the other is a trade-off between time to encode/decode and network bandwidth: option (1) requires little work to decode, but consumes bandwidth (the state of a real program is typically at least several megabytes), while encoding a job as a path requires replay on W_d. We assume that large commodity clusters have abundant CPU but meager bisection bandwidth, so in Cloud9 we chose to encode jobs as the path from the root to the candidate node. As an optimization, we exploit common path prefixes: jobs are not encoded separately, but rather the corresponding paths are aggregated into a job tree and sent as such.



When the job tree arrives at W_d, it is imported into W_d's own subtree, and the leaves of the job tree become part of W_d's frontier (at the time of arrival, these nodes may lie "ahead" of W_d's frontier). W_d keeps the nodes in the incoming jobs as *virtual* nodes, as opposed to *materialized* nodes that reside in the local subtree, and replays paths only lazily. A materialized node is one that contains the corresponding program state, whereas a virtual node is an "empty shell" without corresponding program state. In the common case, the frontier of a worker's local subtree contains a mix of materialized and virtual nodes, as shown in the diagram above.

As mentioned earlier, a worker must choose at each step which candidate node to explore next—this choice is guided by a *strategy*. Since the set of candidate nodes now contains

both materialized and virtual nodes, it is possible for the strategy to choose a virtual node as the next one to explore. When this happens, the corresponding path in the job tree is replayed (i.e., the symbolic execution engine executes that path); at the end of this replay, all nodes along the path are dead, except the leaf node, which has converted from virtual to materialized and is now ready to be explored. Note that, while exploring the chosen job path, each branch produces child program states; any such state that is not part of the path is marked as a fence node, because it represents a node that is being explored elsewhere, so W_d should not pursue it.

Summary A node N in W_i's subtree has two attributes, $N^{\text{status}} \in \{\text{materialized, virtual}\}$ and $N^{\text{life}} \in \{\text{candidate, fence, dead}\}$. A worker's frontier F_i is the set of all candidate nodes on worker W_i. The worker can only explore nodes in F_i, i.e., dead nodes are off-limits and so are fence nodes, except if a fence node needs to be explored during the replay of a job path. The union $\cup F_i$ equals the frontier of the global execution tree, ensuring that the aggregation of worker-level explorations is complete. The intersection $\cap F_i = \emptyset$, thus avoiding redundancy by ensuring that workers explore disjoint subtrees. Fig. 3 summarizes the life cycle of a node.

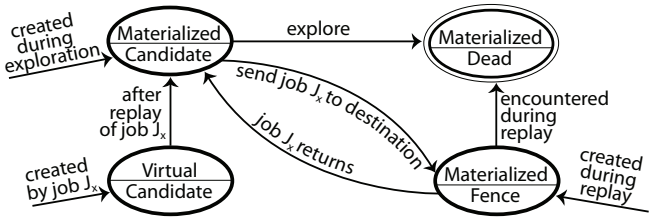


Figure 3: Transition diagram for nodes in a worker's subtree.

As suggested in Fig. 3, once a tree node is dead, it has reached a terminal state; therefore, a dead node's state can be safely discarded from memory. This enables workers to maintain program states only for candidate and fence nodes.

3.3 Cluster-level Operation

Load Balancing When jobs arrive at W_d, they are placed conceptually in a queue; the *length* of this queue is sent to the load balancer periodically. The LB ensures that the worker queue lengths stay within the same order of magnitude. The balancing algorithm takes as input the lengths l_i of each worker W_i's queue Q_i. It computes the average $\bar{l}$ and standard deviation σ of the l_i values and then classifies each W_i as underloaded ($l_i < max\{\bar{l} - \delta \cdot \sigma, 0\}$), overloaded ($l_i > \bar{l} + \delta \cdot \sigma$), or OK otherwise; δ is a constant factor. The W_i are then sorted according to their queue length l_i and placed in a list. LB then matches underloaded workers from the beginning of the list with overloaded workers from the end of the list. For each pair $\langle W_i, W_j \rangle$, with $l_i < l_j$, the load balancer sends a job transfer request to the workers to move $(l_j - l_i)/2$ candidate nodes from W_j to W_i.

Coordinating Worker-level Explorations Classic symbolic execution relies on heuristics to choose which state on the frontier to explore first, so as to efficiently reach the chosen test goal (code coverage, finding a particular type of bug, etc.). In a distributed setting, local heuristics must be coordinated across workers to achieve the global goal, while keeping communication overhead at a minimum. What we have described so far ensures that eventually all paths in the execution tree are explored, but it provides no aid in focusing on the paths desired by the global strategy. In this sense, what we described above is a *mechanism*, while the exploration strategies represent the *policies*.

Global strategies are implemented in Cloud9 using its interface for building *overlays* on the execution tree structure. We used this interface to implement distributed versions of all strategies that come with KLEE [Cadar 2008]; the interface is also available to Cloud9 users. Due to space limitations, we do not describe the strategy interface further, but provide below an example of how a global strategy is built.

A coverage-optimized strategy drives exploration so as to maximize coverage [Cadar 2008]. In Cloud9, coverage is represented as a bit vector, with one bit for every line of code; a set bit indicates that a line is covered. Every time a worker explores a program state, it sets the corresponding bits locally. The current version of the bit vector is piggybacked on the status updates sent to the load balancer. The LB maintains the current global coverage vector and, when it receives an updated coverage bit vector, ORs it into the current global coverage. The result is then sent back to the worker, which in turn ORs this global bit vector into its own, in order to enable its local exploration strategy to make choices consistent with the global goal. The coverage bit vector is an example of a Cloud9 overlay data structure.

4. The POSIX Environment Model

Symbolically executing real-world software is challenging not only because of path explosion but also because real-world systems interact with their environment in varied and complex ways. This section describes our experience building Cloud9's symbolic model of a POSIX environment, which supports most essential interfaces: threads, process management, sockets, pipes, polling, etc. We believe the described techniques are general enough to model other OSes and environments as well.

4.1 Environment Model Design

The goal of a symbolic model is to simulate the behavior of a real execution environment, while maintaining the necessary symbolic state behind the environment interface. The symbolic execution engine (SEE) can then seamlessly transition back and forth between the program and the environment.

While writing and maintaining a model can be laborious and prone to error [Chipounov 2011], there exist cases in which models provide distinct advantages. First, sym-

bolic execution with a model can be substantially faster than without. For instance, in the Linux kernel, transferring a packet between two hosts exercises the entire TCP/IP networking stack and the associated driver code, amounting to over 30 KLOC. In contrast, Cloud9's POSIX model achieves the same functionality in about 1.5 KLOC. Requirements that complicate a real environment/OS implementation, such as performance and extensibility, can be ignored in a symbolic model. Second, when an interface is as stable as POSIX, investing the time to model it becomes worthwhile.

We designed a minimal yet general "symbolic system call" interface to the Cloud9 SEE, which provides the essential building blocks for thread context switching, address space isolation, memory sharing, and sleep operations. These are hard to provide solely through an external model. We give more details about symbolic system calls in §4.2.

In some cases, it is practical to have the host OS handle parts of the environment via *external calls*. These are implemented by concretizing the symbolic parameters of a system call before invoking it from symbolically executing code. Unlike [Cadar 2008; 2006, Godefroid 2005], Cloud9 allows external calls *only* for stateless or read-only system calls, such as reading a system configuration file from the `/etc` directory. This restriction ensures that external concrete calls do not clobber other symbolically executing paths.

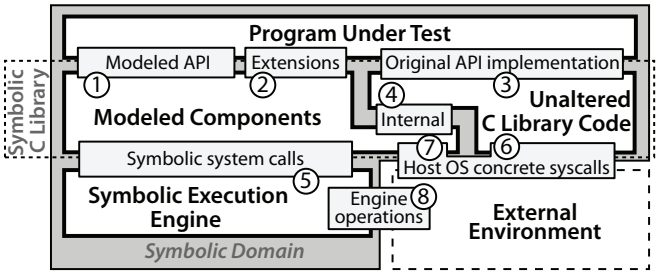


Figure 4: Architecture of the Cloud9 POSIX model.

Cloud9 builds upon the KLEE symbolic execution engine, and so it inherits from KLEE the mechanism for replacing parts of the C Library with model code; it also inherits the external calls mechanism. Cloud9 adds the symbolic system call interface and replaces parts of the C Library with the POSIX model. The resulting architecture is shown in Fig. 4.

Before symbolic execution starts, the Cloud9 system links the program under test with a special symbolic C Library. We built this library by replacing parts of the existing uClibc library in KLEE with the POSIX model code. Developers do not need to modify the code of to-be-tested programs in any way to make it run on Cloud9.

In the C Library, we replaced operations related to threads, processes, file descriptors, and network operations with their corresponding model ①, and augmented the API with Cloud9-specific extensions ②. A large portion of the C Library is reused, since it works out of the box ③ (e.g. memory and string operations). Finally, parts of the original C Library itself use the modeled code ④ (e.g., Standard I/O `stdio` relies on the modeled POSIX file descriptors).

Primitive Name	Description
cloud9_make_shared	Share object across a CoW domain
cloud9_thread_create cloud9_thread_terminate	Create and destroy threads
cloud9_process_fork cloud9_process_terminate	Fork and terminate the current process
cloud9_get_context	Get the current context (pid and tid)
cloud9_thread_preempt	Preempt a thread
cloud9_thread_sleep	Thread sleep on waiting queue
cloud9_thread_notify	Wake threads from waiting queue
cloud9_get_wlist	Create a new waiting queue

Table 1: Cloud9 primitives used to build the POSIX model.

The modeled POSIX components interface with the SEE through symbolic system calls ⑤, listed in Table 1. Occasionally, the unmodified part of the C Library invokes external system calls ⑥, and the model code itself needs support from the host OS ⑦—in order to make sure the external calls do not interfere with the symbolic engine's own operations ⑧, such access is limited to read-only and/or stateless operations. This avoids problems like, for instance, allowing an external `close()` system call to close a network connection or log file that is actually used by the SEE itself.

4.2 Symbolic Engine Modifications

In order to support the POSIX interface, we augmented KLEE with two major features: multiple address spaces per state and support for scheduling threads and processes. This functionality is accessed by model code through the symbolic system call interface (Table 1). Additional models of non-POSIX environments can be built using this interface.

Address Spaces KLEE uses copy-on-write (CoW) to enable memory sharing between symbolic states. We extend this functionality in two ways. First, we enable multiple address spaces within a single execution state, corresponding to multiple processes encompassed in that state. Address spaces can thus be duplicated both across states (as in classic KLEE) and within a state, when `cloud9_process_fork` is invoked, e.g., as used by the POSIX model's `fork()`.

Second, we organize the address spaces in an execution state as *CoW domains* that permit memory sharing between processes. A memory object can be marked as shared by calling `cloud9_make_shared`; it is then automatically mapped in the address spaces of the other processes within the CoW domain. Whenever a shared object is modified in one address space, the new version is automatically propagated to the other members of the CoW domain. The shared memory objects can then be used by the model as global memory for inter-process communication.

Multithreading and Scheduling Threads are created in the currently executing process by calling `cloud9_thread_`

`create`. Cloud9's POSIX threads (pthreads) model makes use of this primitive in its own `pthread_create()` routine.

Cloud9 implements a cooperative scheduler: An enabled thread runs uninterrupted (atomically), until either (a) the thread goes to sleep; (b) the thread is explicitly preempted by a `cloud9_thread_preempt` call; or (c) the thread is terminated via symbolic system calls for process/thread termination. Preemption occurs at explicit points in the model code, but it is straightforward to extend Cloud9 to automatically insert preemptions calls at instruction level (as would be necessary, for instance, when testing for race conditions).

When `cloud9_thread_sleep` is called, the SEE places the current thread on a specified waiting queue, and an enabled thread is selected for execution. Another thread may call `cloud9_thread_notify` on the waiting queue and wake up one or all of the queued threads.

Cloud9 can be configured to schedule the next thread deterministically, or to fork the execution state for each possible next thread. The latter case is useful when looking for concurrency bugs, but it can be a significant source of path explosion, so it should be disabled when not needed.

If no thread can be scheduled when the current thread goes to sleep, then a hang is detected, the execution state is terminated, and a corresponding test case is generated.

Note that parallelizing symbolic execution is orthogonal to providing the multithreading support described above. In the former case, the execution engine is instantiated on multiple machines and each instance expands a portion of the symbolic execution tree. In the latter case, multiple symbolic threads are multiplexed along the same execution path in the tree; execution is serial along each path.

4.3 POSIX Model Implementation

In this section, we describe the key design decisions involved in building the Cloud9 POSIX model, and we illustrate the use of the symbolic system call interface. This is of particular interest to readers who wish to build additional models on top of the Cloud9 symbolic system call interface.

The POSIX model uses shared memory structures to keep track of all system objects (processes, threads, sockets, etc.). The two most important data structures are stream buffers and block buffers, analogous to character and block device types in UNIX. Stream buffers model half-duplex communication channels: they are generic producer-consumer queues of bytes, with support for event notification to multiple listeners. Event notifications are used, for instance, by the polling component in the POSIX model. Block buffers are random-access, fixed-size buffers, whose operations do not block; they are used to implement symbolic files.

The symbolic execution engine maintains only basic information on running processes and threads: identifiers, running status, and parent–child information. However, the POSIX standard mandates additional information, such as open file descriptors and permission flags. This information is stored by the model in auxiliary data structures associated with the currently running threads and processes. The implementations of `fork()` and `pthread_create()` are in charge of initializing these auxiliary data structures and making the appropriate symbolic system calls.

Modeling synchronization routines is simplified by the cooperative scheduling policy: no locks are necessary, and all synchronization can be done using the sleep/notify symbolic system calls, together with reference counters. Fig. 5 illustrates the simplicity this engenders in the implementation of pthread mutex lock and unlock.

```
typedef struct {
  wlist_id_t wlist;
  char taken;
  unsigned int owner;
  unsigned int queued;
} mutex_data_t;

int pthread_mutex_lock(pthread_mutex_t *mutex) {
  mutex_data_t *mdata = ((mutex_data_t**)mutex);
  if (mdata->queued > 0 || mdata->taken) {
    mdata->queued++;
    cloud9_thread_sleep(mdata->wlist);
    mdata->queued--;
  }
  mdata->taken = 1;
  mdata->owner = pthread_self();
  return 0;
}

int pthread_mutex_unlock(pthread_mutex_t *mutex) {
  mutex_data_t *mdata = ((mutex_data_t**)mutex);
  if (!mdata->taken ||
      mdata->owner != pthread_self()) {
    errno = EPERM;
    return -1;
  }
  mdata->taken = 0;
  if (mdata->queued > 0)
    cloud9_thread_notify(mdata->wlist);
  return 0;
}
```

Figure 5: Example implementation of pthread mutex operations in Cloud9's POSIX environment model.

Cloud9 inherits most of the semantics of the file model from KLEE. In particular, one can either open a symbolic file (its contents comes from a symbolic block buffer), or a concrete file, in which case a concrete file descriptor is associated with the symbolic one, and all operations on the file are forwarded as external calls on the concrete descriptor.

In addition to file objects, the Cloud9 POSIX model adds support for networking and pipes. Currently, the TCP and UDP protocols are supported over IP and UNIX network types. Since no actual hardware is involved in the packet transmission, we can collapse the entire networking stack into a simple scheme based on two stream buffers (Fig. 6). The network is modeled as a single-IP network with multiple available ports—this configuration is sufficient to connect multiple processes to each other, in order to simulate and test

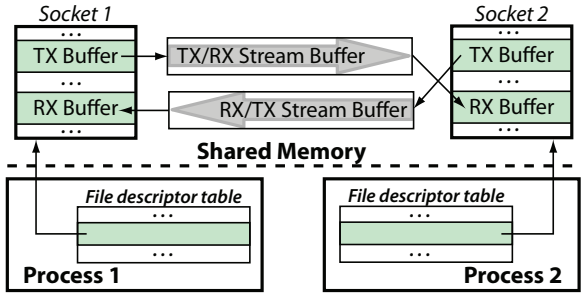


Figure 6: A TCP network connection is modeled in Cloud9 using TX and RX buffers implemented as stream buffers.

distributed systems. The model also supports pipes through the use of a single stream buffer, similar to sockets.

The Cloud9 POSIX model supports polling through the `select()` interface. All the software we tested can be configured to use `select()`, so it was not necessary to implement other polling mechanisms. The `select()` model relies on the event notification support offered by the stream buffers that are used in the implementation of blocking I/O objects (currently sockets and pipes).

The constraint solver used in Cloud9 operates on bit vectors; as a result, symbolic formulas refer to contiguous areas of memory. In order to reduce the constraint solving overhead, we aim to reduce the amount of intermixing of concrete and symbolic data in the same memory region. Thus, Cloud9's POSIX model segregates concrete from symbolic data by using static arrays for concrete data and linked lists (or other specialized structures) for symbolic data. We allocate into separate buffers potentially-symbolic data passed by the tested program through the POSIX interface.

In order to enable testing the systems presented in the evaluation section (§7), we had to add support for various other components: IPC routines, `mmap()` calls, time-related functions, etc. Even though laborious, this was mostly an engineering exercise, so we do not discuss it further.

5. Symbolic Test Suites

Software products and systems typically have large "handmade" test suites; writing and maintaining these suites requires substantial human effort. Cloud9 aims to reduce this burden while improving the quality of testing, by offering an easy way to write "symbolic test suites." First, a symbolic test case encompasses many similar concrete test cases into a single symbolic one—each symbolic test a developer writes is equivalent to many concrete ones. Second, a symbolic test case explores conditions that are hard to produce reliably in a concrete test case, such as the occurrence of faults, concurrency side effects, or network packet reordering, dropping and delay. Furthermore, symbolic test suites can easily cover unknown corner cases, as well as new, untested functionality. In this section, we present the API for symbolic tests and illustrate it with a use case.

Function Name	Description
cloud9_make_symbolic	Mark memory regions as symbolic
cloud9_fi_enable cloud9_fi_disable	Enable/disable the injection of faults
cloud9_set_max_heap	Set heap size for symbolic `malloc`
cloud9_set_scheduler	Set scheduler policy (e.g., round-robin)

Table 2: Cloud9 API for setting global behavior parameters.

Extended Ioctl Code	Description
SIO_SYMBOLIC	Turns this file or socket into a source of symbolic input
SIO_PKT_FRAGMENT	Enables packet fragmentation on this socket (must be a stream socket)
SIO_FAULT_INJ	Enables fault injection for operations on this descriptor

Table 3: Cloud9 extended `ioctl` codes to control environmental events on a per-file-descriptor basis.

5.1 Testing Platform API

The Cloud9 symbolic testing API (Tables 2 and 3) allows tests to programmatically control events in the environment of the program under test. A test suite needs to simply include a `cloud9.h` header file and make the requisite calls.

Symbolic Data and Streams The generality of a test case can be expanded by introducing bytes of symbolic data. This is done by calling `cloud9_make_symbolic`, a wrapper around `klee_make_symbolic`, with an argument that points to a memory region. `klee_make_symbolic` is a primitive provided by KLEE to mark data symbolic. In addition to wrapping this call, we added several new primitives to the testing API (Table 2). In Cloud9, symbolic data can be written/read to/from files, can be sent/received over the network, and can be passed via pipes. Furthermore, the `SIO_SYMBOLIC` `ioctl` code (Table 3) turns on/off the reception of symbolic bytes from individual files or sockets.

Network Conditions Delay, reordering, or dropping of packets causes a network data stream to be fragmented. Fragmentation can be turned on or off at the socket level using one of the Cloud9 `ioctl` extensions. §7 presents a case where symbolic fragmentation enabled Cloud9 to prove that a bug fix for the lighttpd web server was incomplete.

Fault Injection Calls in a POSIX system can return an error code when they fail. Most programs can tolerate such failed calls, but even high-quality production software misses some [Marinescu 2009]. Such error return codes are simulated by Cloud9 whenever fault injection is turned on.

Symbolic Scheduler Cloud9 provides multiple scheduling policies that can be controlled for purposes of testing on a per-code-region basis. Currently, Cloud9 supports a round-robin scheduler and two schedulers specialized for bug finding: a variant of the iterative context bounding scheduling

algorithm [Musuvathi 2008] and an exhaustive exploration of all possible scheduling decisions.

5.2 Use Case

Consider a scenario in which we want to test the support for a new `X-NewExtension` HTTP header, just added to a web server. We show how to write tests for this new feature.

A symbolic test suite typically starts off as an augmentation of an existing test suite; in our scenario, we reuse the existing boilerplate setup code and write a symbolic test case that marks the extension header symbolic. Whenever the code that processes the header data is executed, Cloud9 forks at all the branches that depend on the header content. Similarly, the request payload can be marked symbolic to test the payload-processing part of the system:

```
char hData[10];
cloud9_make_symbolic(hData);
strcat(req, "X-NewExtension: ");
strcat(req, hData);
```

The web server may receive HTTP requests fragmented in a number of chunks, returned by individual invocations of the `read()` system call—the web server should run correctly regardless of the fragmentation pattern. To test different fragmentation patterns with Cloud9, one simply enables symbolic packet fragmentation on the client socket:

```
ioctl(ssock, SIO_PKT_FRAGMENT, RD);
```

To test how the web server handles failures in the environment, we can ask Cloud9 to selectively inject faults when the server reads or sends data on a socket by placing in the symbolic test suite calls of the form:

```
ioctl(ssock, SIO_FAULT_INJ, RD | WR);
```

Cloud9 can also enable/disable fault injection globally for all file descriptors within a certain region of the code using calls to `cloud9_fi_enable` and `cloud9_fi_disable`. For simulating low-memory conditions, Cloud9 provides a `cloud9_set_max_heap` primitive, which can be used to test the web server with different maximum heap sizes.

6. Cloud9 Prototype

We developed a Cloud9 prototype that runs on private clusters as well as cloud infrastructures like Amazon EC2 [Amazon] and Eucalyptus [Eucalyptus]. The prototype has 10 KLOC; the POSIX model accounts for half of the total code size. Cloud9 workers embed KLEE [Cadar 2008], a state-of-the-art single-node symbolic execution engine; the Cloud9 fabric converts a cluster of individual engines into one big parallel symbolic execution engine. This section presents selected implementation decisions underlying the prototype. More details are available at http://cloud9.epfl.ch.

Broken Replays As discussed in §3.2, when a job is transferred from one worker to another, the replay done during materialization must successfully reconstruct the transferred state. Along the reconstruction path, the destination must execute the same instructions, obtain the same symbolic memory content, and get the same results during constraint solving as on the source worker. Failing to do so causes the replayed path to be *broken*: it either diverges, or terminates prematurely. In both cases, this means the state cannot be reconstructed, and this could affect exploration completeness.

The main challenge is that the underlying KLEE symbolic execution engine relies on a global memory allocator to service the tested program's `malloc()` calls. The allocator returns actual host memory addresses, and this is necessary for executing external system calls that access program state. Unfortunately, this also means that buffers are allocated at addresses whose values for a given state depend on the history of previous allocations in *other* states. Such cross-state interference leads to frequent broken replays.

We therefore replaced the KLEE allocator with a per-state deterministic memory allocator, which uses a per-state address counter that increases with every memory allocation. To preserve the correctness of external calls (that require real addresses), this allocator gives addresses in a range that is also mapped in the SEE address space using `mmap()`. Thus, before external calls are invoked, the memory content of the state is copied into the `mmap`-ed region.

Constraint Caches KLEE implements a cache mechanism for constraint-solving results; this cache can significantly improve solver performance. In Cloud9, states are transferred between workers without the source worker's cache. While one might expect this to hurt performance significantly, in practice we found that the necessary portion of the cache is mostly reconstructed as a side effect of path replay, as the path constraints are re-sent to the local solver.

Custom Data Structures We developed two custom data structures for handling symbolic execution trees: *Node pins* are a kind of smart pointer customized for trees. Standard smart pointers (e.g., the ones provided by Boost libraries) can introduce significant performance disruptions when used for linked data structures: chained destructors can introduce noticeable deallocation latency and may even overflow the stack and crash the system. The node pin allows trees to be treated akin to a "rubber band" data structure: as nodes get allocated, the rubber band is stretched, and some nodes act as pins to anchor the rubber band. When such a pin is removed, the nodes with no incoming references are freed up to the point where the rubber band reaches the pin next closest to the root. Tree nodes between two pins are freed all at once, avoiding the use of the stack for recursive destructor calls.

Another custom data structure is the *tree layer*. At first, Cloud9 used a single tree to represent the entire symbolic execution. As Cloud9 evolved, tree nodes acquired an increasing number of objects: program states, imported jobs, breakpoints, etc. This made tree searches inefficient, complicated synchronization, and generally impeded our development effort. We therefore adopted a layer-based structure

similar to that used in CAD tools, where the actual tree is a superposition of simpler layers. When exploring the tree, one chooses the layer of interest; switching between layers can be done dynamically at virtually zero cost. Cloud9 currently uses separate layers for symbolic states, imported jobs, and several other sets of internal information.

7. Evaluation

There are several questions one must ask of a parallel symbolic execution platform, and we aim to answer them in this section: Can it be used on real-world software that interacts richly with its environment (§7.1)? Does it scale on commodity shared-nothing clusters (§7.2)? Is it an effective testing platform, and does it help developers gain confidence in their code (§7.3)? How do its different components contribute to overall efficiency (§7.4)?

For all our experiments, we used a heterogeneous cluster environment, with worker CPU frequencies between 2.3–2.6 GHz and with 4–6 GB of RAM available per core.

On each worker, the underlying KLEE engine used the best searchers from [Cadar 2008], namely an interleaving of random-path and coverage-optimized strategies. At each step, the engine alternately selects one of these heuristics to pick the next state to explore. Random-path traverses the execution tree starting from the root and randomly picks the next descendant node, until a candidate state is reached. The coverage-optimized strategy weighs the states according to an estimated distance to an uncovered line of code, and then randomly selects the next state according to these weights.

To quantify coverage, we report both line coverage and path coverage numbers. Line coverage measures the fraction of program statements executed during a test run, while path coverage reports how many execution paths were explored during a test. Path coverage is the more relevant metric when comparing thoroughness of testing tools. Nevertheless, we also evaluate properties related to line coverage, since this is still a de facto standard in software testing practice. Intuitively, and confirmed experimentally, path coverage in Cloud9 scales linearly with the number of workers.

7.1 Handling Real-World Software

Table 4 shows a selection of the systems we tested with Cloud9, covering several types of software. We confirmed that each system can be tested properly under our POSIX model. In the rest of this section, we focus our in-depth evaluation on several networked servers and tools, as they are frequently used in settings where reliability matters.

Due to its comprehensive POSIX model, Cloud9 can test many kinds of servers. One example is lighttpd, a web server used by numerous high-profile web sites, such as YouTube, Wikimedia, Meebo, and SourceForge. For lighttpd, Cloud9 proved that a certain bug fix was incorrect, and the bug could still manifest even after applying the patch (§7.3.4). Cloud9 also found a bug in curl, an Internet transfer application that is part of most Linux distributions and other operating systems (§7.3.2). Cloud9 also found a hang bug in the UDP handling code of memcached, a distributed memory object cache system used by many Internet services, such as Flickr, Craigslist, Twitter, and Livejournal (§7.3.3).

System	Size (KLOC)	Type of Software
Apache httpd 2.2.16	226.4	Web servers
Lighttpd 1.4.28	39.5	
Ghttpd 1.4.4	0.6	
Memcached 1.4.5	8.3	Distributed object cache
Python 2.6.5	388.3	Language interpreter
Curl 7.21.1	65.9	Network utilities
Rsync 3.0.7	35.6	
Pbzip 2.1.1	3.6	Compression utility
Libevent 1.4.14	10.2	Event notification library
Coreutils 6.10	72.1	Suite of system utilities
Bandicoot 1.0	6.4	Lightweight DBMS

Table 4: Representative selection of testing targets that run on Cloud9. Size was measured using the `sloccount` utility.

In addition to the testing targets mentioned above, we also tested a benchmark consisting of a multi-threaded and multi-process producer-consumer simulation. The benchmark exercises the entire functionality of the POSIX model: threads, synchronization, processes, and networking.

We conclude that Cloud9 is practical and capable of testing a wide range of real-world software systems.

7.2 Scaling on Commodity Shared-Nothing Clusters

We evaluate Cloud9 using two metrics:

1. The time to reach a certain goal (e.g., an exhaustive path exploration, or a fixed coverage level)—we consider this an *external* metric, which measures the performance of the testing platform in terms of its end results.
2. The useful work performed during exploration, measured as the number of useful (non-replay) instructions executed symbolically. This is an *internal* metric that measures the efficiency of Cloud9's internal operation.

A cluster-based symbolic execution engine *scales* with the number of workers if these two metrics improve proportionally with the number of workers in the cluster.

Time Scalability We show that Cloud9 scales linearly by achieving the same testing goal proportionally faster as the number of workers increases. We consider two scenarios.

First, we measure how fast Cloud9 can exhaustively explore a fixed number of paths in the symbolic execution tree. For this, we use a symbolic test case that generates all the possible paths involved in receiving and processing two symbolic messages in the memcached server (§7.3 gives more details about the setup). Fig. 7 shows the time required to finish the test case with a variable number of workers: every doubling in the number of workers roughly halves the

time to completion. With 48 workers, the time to complete is about 10 minutes; for 1 worker, exploration time exceeds our 10-hour limit on the experiment.

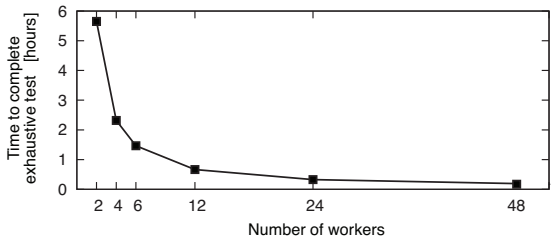


Figure 7: Cloud9 scalability in terms of the time it takes to exhaustively complete a symbolic test case for memcached.

Second, we measure the time it takes Cloud9 to reach a fixed coverage level for the `printf` UNIX utility. `printf` performs a lot of parsing of its input (format specifiers), which produces complex constraints when executed symbolically. Fig. 8 shows that the time to achieve a coverage target decreases proportionally with the number of added workers. The low 50% coverage level can be easily achieved even with a sequential SEE (1-worker Cloud9). However, higher coverage levels require more workers, if they are to be achieved in a reasonable amount of time; e.g., only a 48-worker Cloud9 is able to achieve 90% coverage. The anomaly at 4 workers for 50% coverage is due to high variance; when the number of workers is low, the average (5 ± 4.7 minutes over 10 experiments) can be erratic due to the random choices in the random-path search strategy.

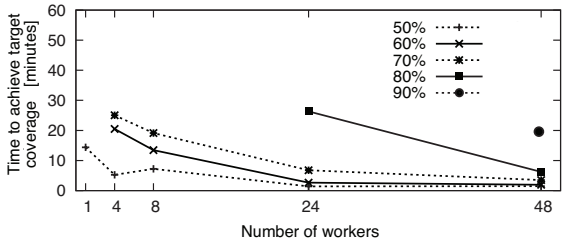


Figure 8: Cloud9 scalability in terms of the time it takes to obtain a target coverage level when testing `printf`.

Work Scalability We now consider the same scalability experiments from the perspective of useful work done by Cloud9: we measure both the total number of instructions (from the target program) executed during the exploration process, as well as normalize this value per worker. This measurement indicates whether the overheads associated with parallel symbolic execution impact the efficiency of exploration, or are negligible. Fig. 9 shows the results for memcached, confirming that Cloud9 scales linearly in terms of useful work done (top graph). The average useful work done by a worker (bottom graph) is relatively independent

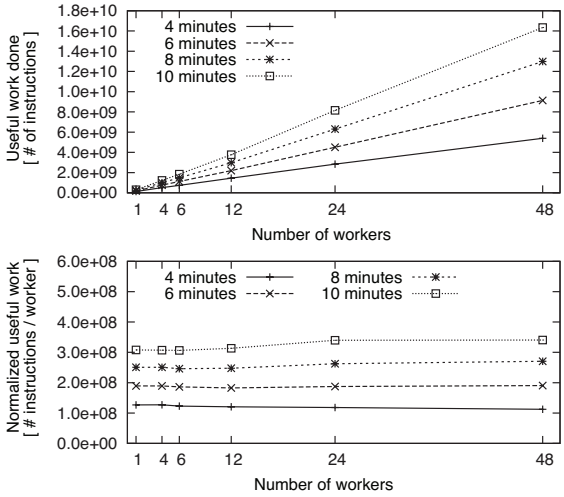


Figure 9: Cloud9 scalability in terms of useful work done for four different running times when testing memcached.

of the total number of workers in the cluster, so adding more workers improves proportionally Cloud9's results.

In Fig. 10 we show the results for `printf` and `test`, UNIX utilities that are an order of magnitude smaller than memcached. We find that the useful work done scales in a similar way to memcached, even though the three programs are quite different from each other (e.g., `printf` does mostly parsing and formatting, while memcached does mostly data structure manipulations and network I/O).

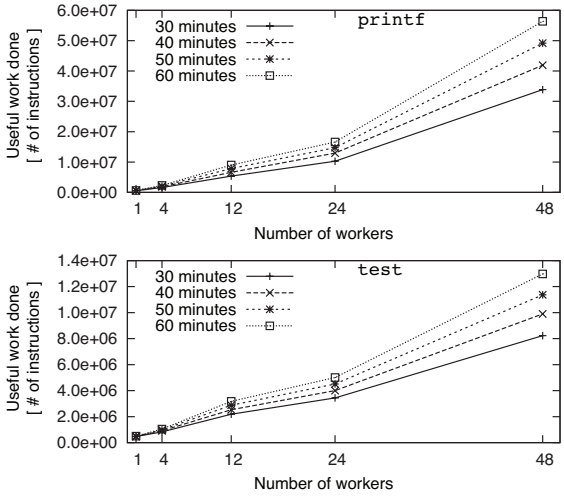


Figure 10: Cloud9's useful work on `printf` (top) and `test` (bottom) increases roughly linearly in the size of the cluster.

In conclusion, Cloud9 scales linearly with the number of workers, both in terms of the time to complete a symbolic testing task and in terms of reaching a target coverage level.

7.3 Effectiveness as a Testing Platform

In this section we present several case studies that illustrate how Cloud9 can explore and find new bugs, confirm/disprove that existing bugs have been correctly fixed, and regression-test a program after it has been modified. In the common case, Cloud9 users start with a concrete test case (e.g., from an existing test suite) and generalize it by making data symbolic and by controlling the environment.

7.3.1 Case Study #1: UNIX Utilities

KLEE is an excellent tool for testing command-line programs, in particular UNIX utilities. It does not tackle more complex systems, like the ones in Table 4, mainly due to path explosion (since KLEE is a single-node engine) and insufficient environment support. We cannot compare Cloud9 to KLEE on parallel and distributed systems, but we can compare on the Coreutils suite of UNIX utilities [Coreutils].

We run KLEE on each of the 96 utilities for 10 minutes, and then run a 12-worker Cloud9 on each utility for 10 minutes. Fig. 11 reports the average coverage increase obtained with Cloud9 over 7 trials, using KLEE's 7-trial average results as a baseline; the experiment totals $2 \times 7 \times 96 \times 10 = 13,440$ minutes > 9 days. The increase in coverage is measured as *additional* lines of code covered, expressed as a percentage of program size (i.e., we do not report it as a percentage of the baseline, which would be a higher number).

Cloud9 covers up to an additional 40% of the target programs, with an average of 13% additional code covered across all Coreutils. In general, improving coverage becomes exponentially harder as the base coverage increases, and this effect is visible in the results: a 12× increase in hardware resources does not bring about a 12× increase in coverage. Our results show that Cloud9 allows "throwing hardware" at the automated testing problem, picking up where KLEE left off. In three cases, Cloud9 achieved 100% coverage in 10 minutes on real-world code. This experiment does not aim to show that Cloud9 is a "better" symbolic execution engine than KLEE—after all, Cloud9 is based on KLEE—but rather that Cloud9-style parallelization can make existing symbolic execution engines more powerful.

The way we compute coverage is different from [Cadar 2008]—whereas KLEE was conceived as an automated *test generator*, Cloud9 is meant to *directly test* software. Thus, we measure the number of lines of code tested by Cloud9, whereas [Cadar 2008] reports numbers obtained by running the concrete test cases generated by KLEE. Our method yields more-conservative numbers because a test generated by KLEE at the end of an incomplete path (e.g., that terminated due to an environment failure) may execute further than the termination point when run concretely.

7.3.2 Case Study #2: Curl

Curl is a popular data transfer tool for multiple network protocols, including HTTP and FTP. When testing it, Cloud9 found a new bug which causes Curl to crash when given a URL regular expression of the form "`http://site.{one,two,three}.com{`". Cloud9 exposed a general problem in Curl's handling of the case when braces used for regular expression globbing are not matched properly. The bug was confirmed and fixed within 24 hours by the developers.

This problem had not been noticed before because the globbing functionality in Curl was shadowed by the same functionality in command-line interpreters (e.g., Bash). This case study illustrates a situation that occurs often in practice: when a piece of software is used in a way that has not been tried before, it is likely to fail due to latent bugs.

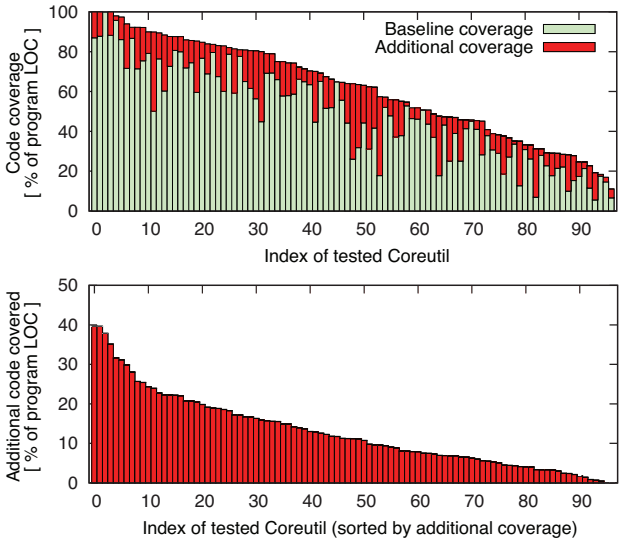


Figure 11: Cloud9 coverage improvements on the 96 Coreutils (1-worker Cloud9 vs. 12-worker Cloud9).

7.3.3 Case Study #3: Memcached

Memcached is a distributed memory object cache system, mainly used to speed up web application access to persistent data, typically residing in a database.

Memcached comes with an extensive test suite comprised of C and Perl code. Running it completely takes about 1 minute; it runs 6,472 different test cases and explores 83.66% of the code. While this is considered thorough by today's standards, two easy Cloud9 test cases further increased code coverage. Table 5 contains a summary of our results, presented in more details in the following paragraphs.

Symbolic Packets The memcached server accepts commands over the network. Based on memcached's C test suite, we wrote a test case that sends memcached a generic, symbolic binary command (i.e., command content is fully symbolic), followed by a second symbolic command. This test captures all operations that entail a pair of commands.

A 24-worker Cloud9 explored in less than 1 hour all 74,503 paths associated with this sequence of two symbolic packets, covering an additional 1.13% of the code relative to

Testing Method	Paths Covered	Isolated Coverage*	Cumulated Coverage**
Entire test suite	6,472	83.67%	—
Binary protocol test suite	27	46.79%	84.33% (+0.67%)
Symbolic packets	74,503	35.99%	84.79% (+1.13%)
Test suite + fault injection	312,465	47.82%	84.94% (+1.28%)

Table 5: Path and code coverage increase obtained by each symbolic testing technique on memcached. We show total coverage obtained with each testing method (*), as well as total coverage obtained by augmenting the original test suite with the indicated method (**); in parentheses, we show the increase over the entire test suite's coverage.

the original test suite. What we found most encouraging in this result is that such exhaustive tests constitute first steps toward using symbolic tests to *prove* properties of real-world programs, not just to look for bugs. Symbolic tests may provide an alternative to complex proof mechanisms that is more intuitive for developers and thus more practical.

Symbolic Fault Injection We also tested memcached with fault injection enabled, whereby we injected all feasible failures in memcached's calls to the C Standard Library. After 10 minutes of testing, a 24-worker Cloud9 explored 312,465 paths, adding 1.28% over the base test suite. The fact that *line* coverage increased by so little, despite having covered almost 50× more paths, illustrates the weakness of line coverage as a metric for test quality—high line coverage should offer no high confidence in the tested code's quality.

For the fault injection experiment, we used a special strategy that sorts the execution states according to the number of faults recorded along their paths, and favors the states with fewer fault injection points. This led to a uniform injection of faults: we first injected one fault in every possible fault injection point along the original C test suite path, then injected pairs of faults, and so on. We believe this is a practical approach to using fault injection as part of regular testing.

Hang Detection We tested memcached with symbolic UDP packets, and Cloud9 discovered a hang condition in the packet parsing code: when a sequence of packet fragments of a certain size arrive at the server, memcached enters an infinite loop, which prevents it from serving any further UDP connections. This bug can seriously hurt the availability of infrastructures using memcached.

We discovered the bug by limiting the maximum number of instructions executed per path to 5×10^6. The paths without the bug terminated after executing $\sim 3 \times 10^5$ instructions; the other paths that hit the maximum pointed us to the bug.

7.3.4 Case Study #4: Lighttpd

The lighttpd web server is specifically engineered for high request throughput, and it is quite sensitive to the rate at which new data is read from a socket. Alas, the POSIX specification offers no guarantee on the number of bytes that can be read from a file descriptor at a time. lighttpd 1.4.12 has a bug in the command-processing code that causes the server to crash (and connected clients to hang indefinitely) depending on how the incoming stream of requests is fragmented.

Fragmentation pattern (data sizes in bytes)	ver. 1.4.12 (pre-patch)	ver. 1.4.13 (post-patch)
1×28	OK	OK
$1 \times 26 + 1 \times 2$	crash + hang	OK
$2+5+1+5+2 \times 1+3 \times 2+5+2 \times 1$	crash + hang	crash + hang

Table 6: The behavior of different versions of lighttpd to three ways of fragmenting the HTTP request "GET /index.html HTTP/1.0CRLFCRLF" (string length 28).

We wrote a symbolic test case to exercise different *stream fragmentation* patterns and see how different lighttpd versions behave. We constructed a simple HTTP request, which was then sent over the network to lighttpd. We activated network packet fragmentation via the symbolic `ioctl()` API explained in §5. We confirmed that certain fragmentation patterns cause lighttpd to crash (prior to the bug fix). However, we also tested the server right after the fix and discovered that the bug fix was incomplete, as some fragmentation patterns still cause a crash and hang the client (Table 6).

This case study shows that Cloud9 can find bugs caused by specific interactions with the environment which are hard to test with a concrete test suite. It also shows how Cloud9 can be used to write effective regression test suites—had a stream-fragmentation symbolic test been run after the fix, the lighttpd developers would have promptly discovered the incompleteness of their fix.

7.3.5 Case Study #5: Bandicoot DBMS

Bandicoot is a lightweight DBMS that can be accessed over an HTTP interface. We exhaustively explored all paths handling the GET commands and found a bug in which Bandicoot reads from outside its allocated memory. The particular test we ran fortuitously did not result in a crash, as Bandicoot ended up reading from the libc memory allocator's metadata preceding the allocated block of memory. However, besides the read data being wrong, this bug could cause a crash depending on where the memory block was allocated.

To discover and diagnose this bug without Cloud9 is difficult. First, a concrete test case has little chance of triggering the bug. Second, searching for the bug with a sequential symbolic execution tool seems impractical: the exhaustive exploration took 9 hours with a 4-worker Cloud9 (and less than 1 hour with a 24-worker cluster).

7.3.6 Discussion

Cloud9 inherits KLEE's capabilities, being able to recognize memory errors and failed assertions. We did not add much

in terms of bug detection, only two mechanisms for detecting hangs: check if all symbolic threads are sleeping (deadlock) and set a threshold for the maximum number of instructions executed per path (infinite loop or livelock). Even so, Cloud9 can find bugs beyond KLEE's abilities because the POSIX model allows Cloud9 to reach more paths and explore deeper portions of the tested program's code—this exposes additional potentially buggy situations. Cloud9 also has more total memory and CPU available, due to its distributed nature, so it can afford to explore more paths than KLEE. As we have shown above, it is feasible to offer proofs for certain program properties: despite the exponential nature of exhaustively exploring paths, one can build small but useful symbolic test cases that can be exhaustively executed.

7.4 Utility of Load Balancing

In this section we explore the utility of dynamic load balancing. Consider the example of exhaustively exploring paths with two symbolic packets in memcached, using 48 workers, but this time from a load balancing perspective. Fig. 12 shows that load balancing events occur frequently, with 3–6% of all states in the system being transferred between workers in almost every 10-second time interval.

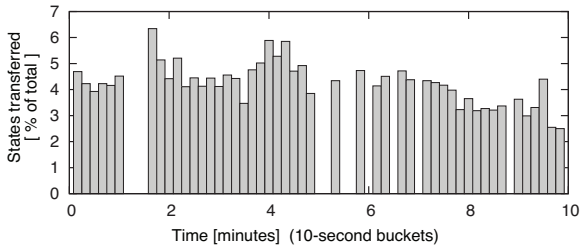


Figure 12: The fraction of total states (candidate nodes) transferred between workers during symbolic execution.

To illustrate the benefits of load balancing, we disable it at various moments in time and then analyze the evolution of total useful work done. Fig. 13 shows that the elimination of load balancing at any moment during the execution significantly affects the subsequent performance of exploration due to the ensuing imbalance. This demonstrates the necessity of taking a dynamic approach to parallel symbolic execution, instead of doing mere static partitioning of the execution tree.

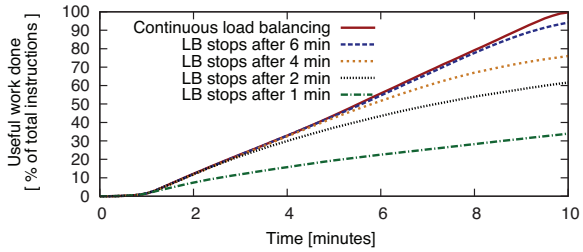


Figure 13: Instruction throughput of Cloud9 with load balancing disabled at various points during the exhaustive test.

8. Related Work

To our knowledge, parallel symbolic execution was first described in [Ciortea 2009]. Cloud9 builds upon those ideas.

Recently, [Staats 2010] described an extension to Java Pathfinder (JPF) that parallelizes symbolic execution by using parallel random searches on a static partition of the execution tree. JPF pre-computes a set of disjoint constraints that, when used as preconditions on a worker's exploration of the execution tree, steer each worker to explore a subset of paths disjoint from all other workers. In this approach, using constraints as preconditions imposes, at *every* branch in the program, a solving overhead relative to exploration without preconditions. The complexity of these preconditions increases with the number of workers, as the preconditions need to be more selective. Thus, per-worker solving overhead increases as more workers are added to the cluster. This limits scalability: the largest evaluated program had 447 lines of code and did not interact with its environment. Due to the *static* partitioning of the execution tree, total running time is determined by the worker with the largest subtree (as explained in §2). As a result, increasing the number of workers can even increase total test time instead of reducing it [Staats 2010]. Cloud9 mitigates these drawbacks.

Several sequential symbolic execution engines [Cadar 2008, Godefroid 2005; 2008, Majumdar 2007] have had great success in automated testing. These state-of-the-art tools exhaust available memory and CPU fairly quickly (as explained in §2). Cloud9 can help such tools scale beyond their current limits, making symbolic execution a viable testing methodology for a wider spectrum of software systems.

To our knowledge, we are the first to scalably parallelize symbolic execution to shared-nothing clusters. There has been work, however, on parallel model checking [Barnat 2007, Grumberg 2006, Kumar 2004, Lerda 1999, Stern 1997]. The SPIN model checker has been parallelized two-way for dual-core machines [Holzmann 2007]. Nevertheless, there are currently no model checkers that can scale to many loosely connected computers, mainly due to the overhead of coordinating the search across multiple machines and transferring explicit states. Cloud9 uses an encoding of states that is compact and enables better scaling.

Swarm verification [Holzmann 2008] generates series of parallel verification runs with user-defined bounds on time and memory. Parallelism is used to execute different search strategies independently. The degree of parallelism is limited to the number of distinct search strategies. Cloud9 is not limited in this way: due to the use of dynamic load balancing, Cloud9 affords arbitrary degrees of parallelism.

Korat [Misailovic 2007] is a parallel testing system that demonstrated ~ 500× speedup when using 1024 nodes. Korat is designed for cases when all program inputs—within a certain bound—can be generated offline. In contrast, Cloud9 handles arbitrarily complex inputs, even if enumeration is infeasible, and can handle system calls and thread schedules.

VeriSoft [Godefroid 1997] introduced stateless search for model checkers, in which a list of state transitions is used to reconstruct an execution state whenever needed. We extend this idea in the Cloud9 job model, which is in essence a form of stateless search. However, jobs are replayed from nodes on the frontier of the execution tree, instead of being replayed from the root. This is an important optimization: in practice, most programs have either a long linear path in the beginning (i.e., initialization code that does not depend on the symbolic input), or long linear executions between two consecutive state forks. Replaying from the root would represent a significant waste of resources for large programs.

Other techniques can be used to improve the scalability of symbolic execution. For example, compositional test generation [Boonstoppel 2008, Godefroid 2007] automatically computes summaries that condense the result of exploration inside commonly used functions. S^2E [Chipounov 2011] improves the scalability of symbolic execution by selectively executing symbolically only those parts of a system that are of interest to the tests. Cloud9 is complementary to these techniques and could be used to scale them further.

9. Conclusion

This paper presented Cloud9, an automated testing platform that employs parallelization to scale symbolic execution by harnessing the resources of commodity clusters. Cloud9 can automatically test real systems, such as memcached, that interact in complex ways with their environment. It includes a new symbolic environment that supports all major aspects of the POSIX interface, and provides a systematic interface to writing symbolic tests. Further information on Cloud9 can be found at http://cloud9.epfl.ch.

Acknowledgments

We are indebted to Ayrat Khalimov, Cristian Cadar, Daniel Dunbar, and our EPFL colleagues for their help on this project. We thank Michael Hohmuth, our shepherd, and the anonymous EuroSys reviewers for their guidance on improving our paper. We are grateful to Google for supporting our work through a Focused Research Award.

References

[Amazon] Amazon. Amazon EC2. http://aws.amazon.com/ec2.

[Barnat 2007] Jiri Barnat, Lubos Brim, and Petr Rockai. Scalable multi-core LTL model-checking. In *Intl. SPIN Workshop*, 2007.

[Boonstoppel 2008] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[Cadar 2008] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[Cadar 2006] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[Chipounov 2009] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[Chipounov 2011] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[Ciortea 2009] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. In *Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[Coreutils] Coreutils. Coreutils. http://www.gnu.org/software/coreutils/.

[Eucalyptus] Eucalyptus. Eucalyptus software. http://open.eucalyptus.com/, 2010.

[Godefroid 2005] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conf. on Programming Language Design and Implementation*, 2005.

[Godefroid 1997] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Symp. on Principles of Programming Languages*, 1997.

[Godefroid 2007] Patrice Godefroid. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages*, 2007. Extended abstract.

[Godefroid 2008] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[Grumberg 2006] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2), 2006.

[Holzmann 2007] G. J. Holzmann and D. Bosnacki. Multi-core model checking with SPIN. In *Intl. Parallel and Distributed Processing Symp.*, 2007.

[Holzmann 2008] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the Swarm tool. In *Intl. SPIN Workshop*, 2008.

[Kumar 2004] Rahul Kumar and Eric G. Mercer. Load balancing parallel explicit state model checking. In *Intl. Workshop on Parallel and Distributed Methods in Verification*, 2004.

[Lerda 1999] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Intl. SPIN Workshop*, 1999.

[Majumdar 2007] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Intl. Conf. on Software Engineering*, 2007.

[Marinescu 2009] Paul D. Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[McConnell 2004] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[Misailovic 2007] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. Parallel test generation and execution with Korat. In *Symp. on the Foundations of Software Eng.*, 2007.

[Musuvathi 2008] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[RedHat] RedHat. RedHat security. http://www.redhat.com/security/updates/classification, 2005.

[Staats 2010] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Intl. Symp. on Software Testing and Analysis*, 2010.

[Stern 1997] Ulrich Stern and David L. Dill. Parallelizing the Murϕ verifier. In *Intl. Conf. on Computer Aided Verification*, 1997.