Cloud9: A Software Testing Service

Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Cloud9 aims to reduce the resource-intensive and labor-intensive nature of high-quality software testing. First, Cloud9 parallelizes symbolic execution (an effective, but still poorly scalable test automation technique) to large shared-nothing clusters. To our knowledge, Cloud9 is the first symbolic execution engine that scales to large clusters of machines, thus enabling thorough automated testing of real software in conveniently short amounts of time. Preliminary results indicate one to two orders of magnitude speedup over a state-of-the-art symbolic execution engine. Second, Cloud9 is an on-demand software testing service: it runs on compute clouds, like Amazon EC2, and scales its use of resources over a wide dynamic range, proportionally with the testing task at hand.

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 [17], and fixing them at so late a stage carries substantial cost [16].

In this paper, we introduce Cloud9, a cloud-based testing service that promises to make high-quality testing fast, cheap, and practical. Cloud9 runs on compute utilities like Amazon EC2 [1], and we envision the following three use cases: First, developers can upload their software to Cloud9 and test it swiftly, as part of their development cycle. Second, end users can upload recently downloaded programs or patches and test them before installing, with no upfront cost. Third, Cloud9 can function as a quality certification service, akin to Underwriters Labs [20], by publishing official coverage results for tested applications. In an ideal future, software companies would be required to subject their software to quality validation on such a service, akin to mandatory crash testing of vehicles. In the absence of such certification, software companies could be held liable for damages resulting from bugs.

For a software testing service to be viable, it must aim for maximal levels of automation. This means the service must explore as many of the software's execution paths as possible without requiring a human to explicitly write test scripts. But such automation can suffer from the tension between soundness and completeness—e.g., static analysis can be complete on large code bases, but typically has a large number of false positives (i.e., is unsound), while model checking is sound, but takes too long to achieve practical completeness on real, large code bases. Of course, some level of assistance is inherently necessary to specify the difference between correct and wrong behavior, but driving the program down execution paths should not require human effort.

Cloud9 achieves high levels of automation by employing symbolic execution. Introduced in the 1970s [12], this technique can explore all feasible execution paths in a program, thus being an ideal candidate for test automation. Unfortunately, it faces serious challenges, namely *high memory consumption* and *CPU-intensive constraint solving*, both exponential in program size. On a present-day computer, it is only feasible to test programs with a few thousand lines of code; for larger programs, typically only the shorter paths can be explored. Thus, symbolic execution is virtually unheard of in the general software industry, because real software often has millions of lines of code, rendering symbolic execution infeasible.

Cloud9 is the first *parallel* symbolic execution engine to run on *large shared-nothing clusters* of computers, thus harnessing their aggregate memory and CPU resources. While parallelizing symbolic execution is a natural way to improve the technique's scalability, doing so in a cluster presents significant research challenges: First, balancing execution workload among nodes becomes a complex multi-dimensional optimization problem with several unknown inputs. Second, global coordination can only be done infrequently, so new search strategies must be devised for exploring a program's paths in parallel.

This paper presents preliminary steps toward solving these problems. After presenting the concept of a software testing service in more detail (§2), we describe our techniques for efficiently parallelizing symbolic execution (§3). An initial prototype suggests that our approach is practical and useful: compared to the state of the art, Cloud9 reduced automated testing time of 32 real UNIX utilities on average by a factor of 47, with a maximum of 250-fold speedup (§4). The paper closes with related work (§5) and conclusions (§6).

2 Software Testing as a Service

Unlike classic testing frameworks, Cloud9 runs as a Web service. It is primarily meant to operate on public cloud infrastructures like Amazon EC2 [1], but can also be used privately on clusters running cloud software like Eucalyptus [7]. Testing-as-a-service has several benefits.

First, Cloud9 offers a cost-effective, flexible way to run massive test jobs with no upfront cost. Unlike owning a private cluster, Cloud9 allows necessary machines to be commissioned only when needed for testing, in a number suitable to the complexity of the testing task. If a team required, e.g., 1,000 nodes for one hour every fortnight, the corresponding yearly budget could be as low as $2,500 on EC2. This is orders of magnitude less than the cost of acquiring and operating a private cluster of the same size.

Second, an automated-test service reduces the learning curve associated with test frameworks. A standard Web service API can hide the complexity of operating an automated test infrastructure, thus encouraging developers to use it more frequently and, especially for new hires, to adopt thorough testing practices early on.

Third, running test infrastructure as a service offers high flexibility in resource allocation. Whereas one would normally have to reserve a test cluster ahead of time, a cloud-based service can provision resources on-demand, corresponding to the complexity of the testing task at hand (e.g., depending on program size). It can also elastically recruit more resources during compute-intensive phases of the tests and release them during the other phases.

The service interface requires a user to upload the program to test, a testing goal, and a resource policy. The *program* can be in binary form, source code, or an intermediate representation like LLVM [14]. The *test goal* tells the symbolic engine how to determine whether it reached a termination condition—e.g., the test goal might be to find inputs that crash the program or exercise a security vulnerability. The *resource policy* indicates a cost budget along with guidance on how to resolve tradeoffs between the test goal and the budget. E.g., one may want as high a coverage as possible, but all within a $1,000 budget.

Cloud9 returns to the user a set of automatically discovered input tuples that trigger conditions specified in the test goal, together with statistical information. For example, if the test goal was 90% line coverage, Cloud9 would produce a test suite (i.e., a set of program input tuples, or test cases) that, in the aggregate, exercise 90% of the uploaded program. Alternatively, the goal may be to test for crashes, in which case Cloud9 produces a set of pathological input tuples that can be used to crash the program, prioritized based on severity. Each such input tuple can be accompanied by a corresponding coredump and stack trace, to speed up debugging. Cloud9 does not require any special software on the user's side—the input tuples serve as the most eloquent evidence of the discovered bugs.

Upon receiving the results, users may be charged for the service based on previously agreed terms. It is essential that the pricing model capture the true value offered by Cloud9. While compute clouds today adopt a rental model (e.g., EC2 nodes cost $0.10/hour/node), a Cloud9 user does not derive value proportional to this cost. We favor a model in which users are charged according to their test goal specification. For example, if the goal is a certain level of coverage, then the user is charged a telescoping amount $\$x$ for each percentage point of coverage. If the goal is to find crashes, then a charge of $\$x$ for each crash-inducing defect is reasonable. In both cases, x can be proportional to program size. A good pricing model encourages frequent use of the service, thus increasing the aggregate quality of the software on the market.

Finally, a viable testing service must address issues related to confidentiality of both uploaded material and corresponding test results, as well as multi-tenancy. There are also opportunities for amortizing costs across customers, e.g., by reusing test results for frequently used libraries and frameworks, like libc, STL, log4j, etc.

In the rest of this paper, we focus on our main research contribution: parallelizing symbolic execution.

3 Parallel Symbolic Execution

Symbolic execution [12] offers great promise as a technique for automated testing [10, 15, 4], as it can find bugs without human assistance. Instead of running the program with regular inputs, a symbolic execution engine executes a program with "symbolic" inputs that are unconstrained, e.g., an integer input x is given as value a symbol α that can take on any integer value. When the program encounters a branch that depends on x, program state is forked to produce two parallel executions, one following the then-branch and another following the else-branch. The symbolic values are constrained in the two clones so as to make the branch condition evaluate to true (e.g., $\alpha<0$), respectively false (e.g., $\alpha\geq 0$). Execution recursively splits into sub-executions at each relevant branch, turning an otherwise linear execution into an execution tree (Fig. 1).

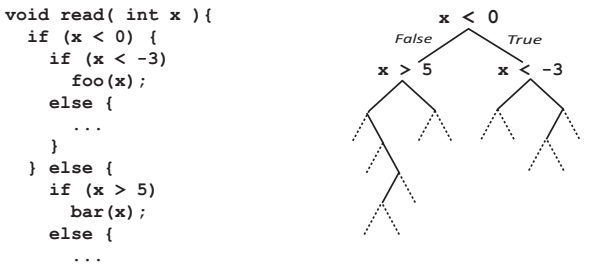

Figure 1: Example of an execution tree.

Symbolic execution, then, consists of the systematic exploration of this execution tree. Each inner node is a branching decision, and each leaf is a program state that contains its own address space, program counter, and set of constraints on program variables. When an execution

encounters a testing goal (e.g., a bug), the constraints collected from the root to the goal leaf can be solved to produce concrete program inputs that exercise the path to the bug. Thus, symbolic execution is substantially more efficient than exhaustive input-based testing (it analyzes the behavior of code for entire classes of inputs at a time, without having to try each one out), and equally complete.

3.1 Challenges

Classic symbolic execution faces three major challenges: path explosion, constraint solving overhead, and memory usage. *Path explosion* is due to the number of execution paths in a program growing exponentially in the number of branch instructions that depend (directly or indirectly) on inputs; as a result, larger programs cause testing to take exponentially longer. Compounding this effect is the fact that the CPU-intensive *constraint solver* must be invoked every time a branch instruction that depends on symbolic inputs is executed; we have found constraint solving to consume on the order of half of the total execution time. Finally, high *memory usage* resulting from path explosion causes symbolic execution engines to run out of memory before having explored a substantial fraction of a program's paths. As a result, state-of-the-art symbolic execution engines [10, 4] can test only small programs with a few thousands of lines of code, while real software systems are orders of magnitude bigger. By parallelizing symbolic execution on clusters, we aim for the equivalent of a classic symbolic execution engine on a "machine" with endlessly expandable memory and CPU power.

Parallel symbolic execution brings three new challenges: the need to do "blindfolded" work partitioning, distributing the search strategy without coordination, and avoiding work and memory redundancy.

First, the path exploration work must be distributed among worker nodes without knowing how much work each portion of the execution tree entails. The size of subtrees cannot be known a priori: determining the propagation of symbolic inputs to other program variables requires executing the program first. It is precisely this propagation that determines which branch instructions will create new execution states, i.e., nodes in the execution tree. As it turns out, execution trees are highly unbalanced, and statically finding a balanced partitioning of an unexpanded execution tree reduces to the halting problem. In addition to subtree size, another unknown is how much memory and CPU will be required for a given state—the amount of work for a subtree is the sum of all nodes' work. Thus, work distribution requires (as we will see later) a dynamic load balancing technique.

Second, distributed exploration of an execution tree requires coordinating the strategies of a large number of workers, which is expensive. Classic symbolic execution relies on heuristics to choose which state from the execution tree to explore first, so as to efficiently reach the test goal. In the parallel case, local heuristics must be coordinated across workers, to achieve the global goal while minimizing redundant work; but global coordination implies high communication overhead. Test goals, like maximizing test coverage, require more complex search strategies than, e.g., iterative deepening depth-first search, often used in model checkers.

Third, in a shared-nothing cluster, the risks of redundancy in path exploration and the opportunities for memory duplication are many more than on a single node. For instance, symbolic execution engines [5, 4] use copy-on write to maximize sharing of memory between execution states, substantially decreasing memory consumption. To achieve this in a cluster, load balancing must take into account the memory sharing opportunities and group similar states on the same worker. Deduplication techniques, such as bitstate hashing used in SPIN [11], are not readily usable for nodes in a symbolic execution tree, since different nodes in the tree can turn out to have identical state (e.g., due to commutative path segments) and a distributed hashing data structure would need to be implemented, which requires special effort and also incurs some performance penalties.

In general, the methods used so far in parallel model checkers [19, 3, 13, 2, 11] do not scale to shared-nothing clusters. They also rely often on a priori partitioning a finite state space.

In a cloud setting, running parallel symbolic execution further requires coping with frequent fluctuation in resource quality, availability, and cost. Machines have variable performance characteristics, their network proximity to each other is unpredictable, and failures are frequent. A system like Cloud9 must therefore cope with these problems in addition to the fundamental challenges of parallel symbolic execution.

3.2 Overview of Our Solution

Cloud9 consists of multiple workers and a load balancer (see Fig. 2). Each worker independently explores a subtree of the program's execution tree, by running a symbolic execution engine—consisting of a runtime and a searcher—and a constraint solver. Upon encountering a conditional branch in the program, the *runtime* initializes a child node in the execution tree corresponding to each branch outcome. The *searcher* is asked which node in the tree to go to next (e.g., a DFS strategy would dictate always choosing the leftmost unexplored child). Once the searcher returns a choice, the runtime calls upon the *constraint solver* to determine the feasibility of the chosen node, i.e., whether there exist values that would satisfy the conjunction of the branch predicates along the path to the chosen node. If the path is feasible, the runtime follows it and adds the corresponding branch predicate to the path's constraints; otherwise, a new choice is requested.

A smart exploration strategy helps find sooner the paths leading to the requested goal. This is particularly relevant for symbolic execution trees of infinite size. The searcher

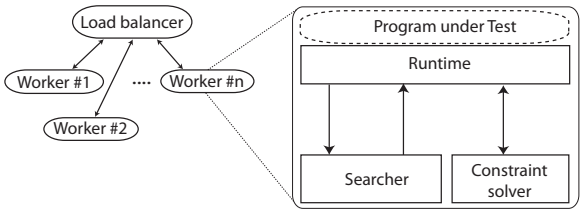

Figure 2: Cloud9 architecture

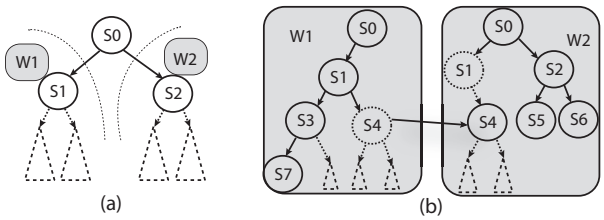

Figure 3: (a) Initial partition of the search space; (b) Repartitioning: W_1 delegates to W_2 the subtree rooted at state S_4. Dotted circles indicate states that are remote; dotted edges similarly indicate locally-unexplored paths.

can choose any node on the unexplored horizon of the execution tree, not just the immediate descendants of the current node.

The overall exploration is global, while Cloud9 searchers have visibility only into the execution trees assigned to their particular workers. Thus, worker-level strategies must be coordinated—a tightly coordinated strategy could achieve as efficient an exploration (i.e., with as little redundant work) as a single-node symbolic execution engine. It is also possible to run multiple instances of the runtime and searcher on the same physical machine, in which case the strategies of the co-located searchers can see all subtrees on that machine. But tight coupling limits the ability of workers to function independently of each other, and would thus hurt scalability.

In order to steer path exploration toward the global goal, Cloud9 employs several techniques: Two-phased load balancing (§3.3) starts with an initial static split of the execution tree and adjusts it dynamically, as exploration progresses. Replacing a single strategy with a portfolio of strategies (§3.4) not only compensates for the limited local visibility, but can also better exploit Cloud9's parallelism. Finally, we employ techniques for reducing redundancy, handling worker failures, and coping with heterogeneity (§3.5).

3.3 Load Balancing

A measure of a Cloud9 worker's *instantaneous load* is the amount of work it is guaranteed to have in the near future. When a worker forks new states, it adds these states to a work queue, and the length of this queue provides a lower bound on the amount of work left. In order to account for heterogeneity of both the states and the underlying hardware, we are exploring other second-order metrics as well, such as states' average queue wait time, amount of memory they consume, number of queries passed to the constraint solver, etc. Workers periodically report their load to the central load balancer, which computes the global load distribution.

When Cloud9 starts, there is no load information, so it statically partitions the search space (execution tree) among workers. Figure 3(a) illustrates an initial choice for a two-worker Cloud9: one branch of the first branch instruction in the program is explored by worker W_1 and the other branch by worker W_2.

The search space must be repartitioned on the fly, because the execution tree can become highly imbalanced. Consider the execution tree in Figure 1: calling *foo(α)* could execute substantially more branch instructions that depend on α than *bar(α)*, thereby causing more new states to be created in the corresponding subtree. Figure 3b illustrates one of the simplest cases of work imbalance: worker W_2 finishes exploring its subtree before W_1, so W_1 delegates to W_2 the expansion of subtree S_4.

More generally, the load balancer declares a *load imbalance* when the most loaded worker W has at least x times the load of the least loaded worker w. We obtained good results by using $x = 10$, which helps preventing high overheads associated with frequent load balancing. At this point, the load balancer instructs W and w to negotiate a way of equalizing their load. The two workers agree on the set of states $\{S_i\}$ to delegate from W to w, based not only on the number of states, but also on other locally-computed metrics, such as which states (subtrees) appear to have highest constraint solving time or highest memory footprint. The main load balancing primitive is a worker-to-worker operation *delegate(S,W,w)*, which shifts the responsibility for exploring the subtree rooted at state S from overloaded worker W to the lighter-loaded w. The actual computation and delegation occur asynchronously, allowing the workers to continue exploring while load is being balanced.

There are two ways to delegate subtrees: state copying and state reconstruction. In *state copying*, W marshals the states (roots of the subtrees to be explored) and sends them over the network to w, which unmarshals them and adds them to its work queue. In *state reconstruction*, the identity of a state S_i is encoded as a bitvector representing the then/else decisions needed to reach from the program's start state to S_i. For example, referring to Figure 3b, node S_1 is encoded as *0*, S_4 as *01*, S_7 as *000*, etc. When delegating responsibility for S_i, W merely sends the bitvector encoding of S_i to w, after which w reconstructs S_i by rerunning the program symbolically along the path indicated by the bitvector. Since the execution tree is uniquely determined by the program under test, each node can be deterministically reconstructed this way. Our bitvector encoding resembles encodings used in

stateless search, first introduced by Verisoft [9] for model checking concurrent programs.

Choosing the best candidate for delegation is governed by the CPU vs. network tradeoff: sending bitvectors is network-efficient, but consumes CPU for reconstruction on the target worker, while transferring states is CPU-efficient, but consumes network bandwidth. State reconstruction is cheaper for subtrees whose roots are shallow in the execution tree. In addition, to optimize reconstruction time, the target worker reconstructs from the deepest common ancestor between already-explored nodes and the newly received subtree. Since Cloud9 uses copy-on-write to share common memory objects between states, the longer the common prefix of two nodes in the execution tree, the higher the memory sharing benefit will be. Finally, during reconstruction, Cloud9 need not invoke the constraint solver, since the bitvector-encoded path is guaranteed to be feasible.

3.4 Exploration Strategy Portfolio

Load balancing provides the means of connecting local strategies to the global goal. For instance, if the goal is to obtain high coverage tests, Cloud9 searchers will assign a local score for each state *S* indicating the expected coverage one might obtain by exploring *S*. High-score states are moved to the head of each worker's queue and prioritized for delegation to less loaded workers, to be executed as soon as possible. Thus, each load balancing decision moves Cloud9 closer to the global goal.

In contrast to sequential symbolic execution, which is constrained to using one search strategy at a time, Cloud9 can employ a portfolio of concurrent strategies. We think of this portfolio in analogy to financial investment portfolios: if a strategy is a stock, workers represent cash, and the portfolio's return is measured in results per unit of time, then we face the problem of allocating cash to stocks so as to maximize overall return. By casting the exploration problem as an investment portfolio optimization problem, we expect to reuse portfolio theory results, such as diversification and speculation, as well as quantitative techniques for improving returns, such as efficient frontier and alpha/beta coefficients. For example, we can speculatively devote a small number of workers to a strategy that works exceptionally well, but only for a small fraction of programs. Running this exploration on a copy of the execution tree in parallel with a classic strategy that bears less risk may improve the expected time of reaching the overall goal.

3.5 Minimizing Redundant Work

An important aspect of achieving scalability is the avoidance of redundant work. One source of redundant work is state duplication, which occurs by expanding identical states on different machines, which occurs frequently when testing multi-threaded programs. We intend to handle this problem by exploring commutative thread interleavings on the same machine, and perform state deduplication locally, using dynamic partial order reduction [8]. Commutative (but different) path segments can also lead to different nodes being identical in terms of contents.

We found constraint solving to account for half or more of total symbolic execution time. Some of this time goes into re-solving constraints previously solved. Thus, we are building a distributed cache of constraint solutions, which allows workers to reuse the computation performed by other workers.

Failing workers—a frequent occurrence in large clusters—also lead to redundancy, as other workers have to redo the work of failed workers. In Cloud9 we employ checkpointing at multiple levels to enable restarting a failed search on a peer worker. Since symbolic execution is memory and CPU-intensive, asynchronous checkpoints to stable storage are cheap. Worker failure can be thought of as an extreme case of worker performance heterogeneity, which is normally handled by monitoring queue lengths and constraint solving times, as mentioned in §3.3.

4 Initial Prototype

We built a preliminary Cloud9 prototype that runs on Amazon EC2 [1] and uses the single-node Klee symbolic execution engine [4]. Preliminary measurements indicate that Cloud9 can achieve substantial speedups over Klee. For our measurements, we used single-core EC2 nodes and instructed both Klee and Cloud9 to automatically generate tests to exercise various UNIX utilities, with the aim of maximizing test coverage, as in [4].

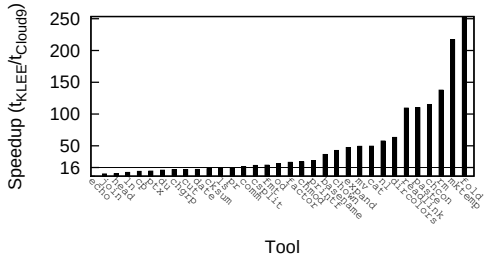

Figure 4: Time to match Klee's coverage with Cloud9. The 16× line represents linear speedup.

We measured how much faster a 16-node Cloud9 can achieve the same level of coverage that Klee achieves in one hour. We tested a random subset of 32 UNIX utilities, with a uniform distribution of binary sizes between the smallest utility (`echo` at 40 KB) and the largest one (`ls` at 170 KB). Figure 4 shows the results: speedup ranges from 2× to 250×, with an average speedup of 47×. The speedup exceeds the 16-fold increase in computation resources, because Cloud9 not only partitions the search

across 16 nodes, but also increases the probability that a given worker will find states with high coverage potential.

We also compared the amount of coverage obtained for a given level of CPU usage. We ran Klee for 16 hours on one node and Cloud9 for 1 hour on 16 nodes (Figure 5), thus giving each tool 16 CPU-hours. Cloud9 outperformed Klee in 28 out of 32 cases, reconfirming the multiplicative benefit of parallel symbolic execution.

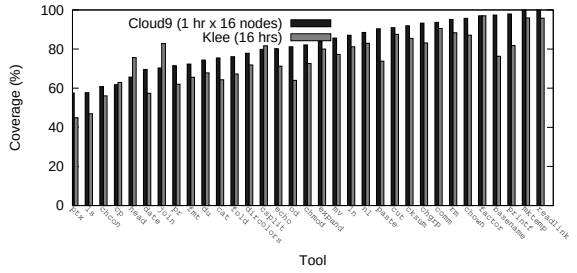

Figure 5: Coverage obtained by Cloud9 and Klee, using identical number of CPU-hours.

5 Related Work

To our knowledge, we are the first to parallelize symbolic execution to clusters of computers. There has been work, however, on parallel model checking [19, 3, 2]. 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. SPIN is a mature model checker that parallelizes and diversifies its search strategy on a shared-memory multi-core system [3, 11]; we cannot directly apply those techniques to shared-nothing clusters. Moreover, for the programs tested in [3, 11], the search space could be statically partitioned a priori, which is not feasible for Cloud9.

There have been previous efforts to scale symbolic execution that do not involve parallelization. For example, concolic testing [18] runs a program concretely, while at the same time collecting path constraints along the explored paths; the constraints are then used to find alternate inputs that would take the program along different paths. Another example is S^2E [6], which improves scalability by automatically executing symbolically only those parts of a system that are of interest. Our techniques are complementary, and in our future work we intend to combine S^2E with Cloud9. In general, Cloud9 benefits from almost all single-node improvements of symbolic execution.

6 Conclusion

This paper proposes Cloud9, a cloud-based parallel symbolic execution service. Our work is motivated by the severe limitations of symbolic execution—memory and CPU usage—that prevent its wide use. Cloud9 is designed to scale gracefully to large shared-nothing clusters; by harnessing the aggregate resources of such clusters, we aim to make automated testing based on symbolic execution feasible for large, real software systems.

Cloud9 is designed to run as a Web service, thus opening up the possibility of doing automated testing in a pay-as-you-go manner. We believe that a cloud-based testing service can become an essential component of software development infrastructure: it provides affordable and effective software testing that can be provisioned on-demand and be accessible to all software developers.

References

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

[2] J. Barnat, L. Brim, and P. Rockai. Scalable multi-core LTL model-checking. In *Intl. SPIN Workshop*, 2007.

[3] J. Barnat, L. Brim, and J. Stribna. Distributed LTL model-checking in SPIN. In *Intl. SPIN Workshop*, 2001.

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

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[6] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[7] Eucalyptus software. http://open.eucalyptus.com/.

[8] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 2005.

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

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

[11] G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification. In *Intl. Conf. on Automated Software Engineering*, 2008.

[12] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

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

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[15] R. Majumdar and K. Sen. Hybrid concolic testing. In *Intl. Conf. on Software Engineering*, 2007.

[16] S. McConnell. *Code Complete*, chapter 3. Microsoft Press, 2004.

[17] Redhat security. `http://www.redhat.com/security/updates/classification/`, 2005.

[18] K. Sen. Concolic testing. In *Intl. Conf. on Automated Software Engineering*, 2007.

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

[20] Underwriters Labs. `http://www.ul.com`.