Automated Vulnerability Discovery in Distributed Systems

Radu Banabic, George Candea and Rachid Guerraoui
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

In this paper we present a technique for automatically assessing the amount of damage a small number of participant nodes can inflict on the overall performance of a large distributed system. We propose a feedback-driven tool that synthesizes malicious nodes in distributed systems, aiming to maximize the performance impact on the overall behavior of the distributed system. Our approach focuses on the interface of interaction between correct and faulty nodes, clearly differentiating the two categories. We build and evaluate a prototype of our approach and show that it is able to discover vulnerabilities in real systems, such as PBFT, a Byzantine Fault Tolerant system. We describe a scenario generated by our tool, where even a single malicious client can bring a BFT system of over 250 nodes down to zero throughput.

1. Introduction

The future of computing is communication. Already today, most modern devices are interconnected: computers, phones, cars, TVs, PDAs, navigation devices, MP3 players, even watches. However, the benefits of such interconnectedness are clouded by the many ways in which malicious entities can exploit unsuspecting users. Our goal is to help developers build high-assurance distributed systems.

Distributed systems are complex, thus developers have a hard time reasoning about how these systems will behave once deployed. There has been extensive work on using techniques such as model checking to verify distributed protocols ([8], [10], [14]). Complementing this valuable prior work, we note that distributed system behaviors emerge not only from the design of the system and the algorithms, but also from how they are implemented and how third party software (that they interact with) behaves.

Alarmingly, large distributed systems are vulnerable to even a few compromised nodes that are either failed, or controlled by a malicious attacker. For example, a vulnerability [2] was recently discovered in BitTorrent, a popular peer-to-peer file sharing system: a malicious entity can craft a distributed hash table that co-opts correct nodes into unwittingly performing a distributed Denial of Service (DoS) attack on a target of the entity's choosing. A malicious user, controlling a single machine, can redirect tens of thousands of correct nodes in the file sharing system towards any target, even outside the BitTorrent pool. PBFT [4], a Byzantine Fault Tolerant system, is also surprisingly vulnerable; we show in our experiments that even a single faulty (or malicious) client can completely disrupt a PBFT deployment of 250 nodes.

State-of-the-art testing of large distributed systems often relies in practice on fuzzing, i.e., on trying out a variety of inputs and checking how the system behaves [18]. Unfortunately, this approach suffers from the fact that the space of possible inputs is extremely large. Even more advanced approaches, such as symbolic execution, are overkill, since they consider the distributed system as a whole. This "over-zealous" testing leads to a very large number of possible test scenarios and, thus, to an unfeasible total test time. It also incurs significant post-test human effort to distinguish between realistic and unrealistic scenarios. Furthermore, many such testing techniques focus on security or dependability of the entire system, without distinguishing between correct and compromised nodes and without considering system performance; this may lead to both false negatives, missing various performance issues, and to false positives, e.g., by finding scenarios that lead to the "attacker" itself crashing.

2. AVD: Synthesizing Malicious Entities

Our goal is to develop techniques for automatically assessing the amount of damage that a few faulty (or compromised) nodes could inflict on the overall behavior of the implementation of a distributed system. Developers can then use these techniques to efficiently and cost-effectively identify vulnerabilities before releasing their software. The express purpose is to test implementations, because often the vulnerabilities lie not in protocols and models, but in the code that implements them. In this way, our work complements existing work in model checking of distributed systems.

A key insight is that failures (or attacks) generally occur only in a small fraction of the nodes in a distributed

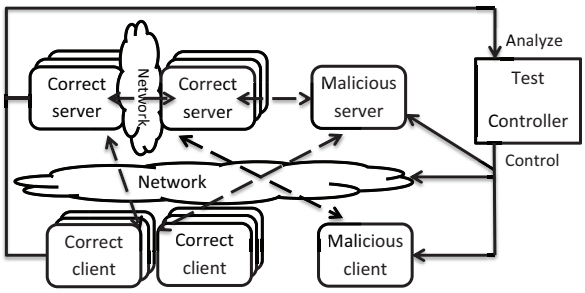

Figure 1. Architecture of AVD.

system. Furthermore, the probability of such faults occurring goes up as distributed systems grow larger and are becoming heterogeneous. Distributed systems should be able to contain errors to the failing nodes and, through replication, trim those nodes out of the system without data loss. However, it is often the case that errors spread throughout the system before detection, making gracious recovery impossible. Thus, our approach only targets a small subset of the total nodes in a distributed system and focuses on the interface of interaction between correct and faulty nodes, searching for faults that propagate through the interface and impact the entire distributed system.

We embody our approach in AVD, an automated vulnerability discovery platform. The key idea is for AVD to generate malicious entities in the target distributed system, instead of generating low-level inputs. AVD then assesses the impact the malicious entities have on the overall system's behavior with regard to the correct, unmodified nodes. One can think of this approach as "fuzzing" at the level of system nodes — akin to input fuzzing, but at a higher level of abstraction. Such an approach can be used to find potential bugs or bottlenecks in a distributed system, but also to evaluate an Application Programming Interface before deployment (i.e., discover if the API enables certain attacks from clients, by being too permissive).

Figure 1 shows AVD's architecture. The system under test is composed of several servers and several clients, connected by networks. The nodes of the system are partitioned into correct nodes, which are considered to be properly configured, and malicious nodes, which are controlled by AVD. The networks may also be under the control of AVD, since attackers can be assumed to exercise some sort of control over the network (ranging from DoS attacks to taking control of routers). The core of AVD is the Test Controller, which generates the parameters for each testing tool in use and coordinates them to enhance the performance impact.

3. Coordinating the Tests

Modern distributed systems are prone to complex failures; for instance Amazon EC2 was recently subjected to severe outage [**?**] due to a combination of factors, including a human configuration error and a race condition. Malicious attacks on distributed systems are also complex, entailing coordination among several attacker nodes and using combinations of attack vectors (e.g., use a DoS attack to bring down a server, then spoof its identity to take control over the entire distributed system). Unlike tools that focus on exploiting individual faults, AVD will automatically explore the extent to which a combination of failures or attacks can damage the target system.

In order to synthesize complex failures, the AVD controller needs to coordinate the various tools available, generating parameters for each testing tool, for the target system and for the network. This can be seen as exploring a hyperspace, where each point represents the configuration of an individual test scenario. Each dimension in the hyperspace represents the set of values that can be assigned to a particular parameter in the test. For example, a library-level fault injector can inject a variety of faults in the system under test — the function where to inject, the error code and the call number are the three dimensions describing the hyperspace of library fault injection parameters. Each testing tool, as well as the network configuration and system under test configuration have their own hyperspaces. The controller has to iterate through the composition of each individual hyperspace.

Exploring the hyperspace of test parameters can be seen as playing a game of battleships, where the hyperspace corresponds to the grid, the vulnerabilities of the system under test correspond to the battleships and running a test corresponds to firing a shot. In the board game, players begin by firing random shots in order to gain knowledge on the structure of the opponent's board (the placement of the ships). Then, as the player gains more and more knowledge on the structure, the shots become more informed, more focused and more efficient. The Test Controller, just like a battleships player, explores the space iteratively, by initially running random tests and then leveraging the information gained in order to improve the efficiency of subsequent tests.

The metric used by AVD to assess the impact of a test (of a fault) is the impact on the correct, unmodified nodes of the target system. Each explored point in the hyperspace of test parameters will have an associated impact measurement, obtained after running the test. Individual tests are independent; the distributed system is reinitialized before running a new test.

The exploration algorithm in the Controller is a metaheuristic similar to hill-climbing. The Controller keeps a set containing the tests that have previously had the highest impact on the system under test (as measured with the

previously defined metric) and performs slight mutations on those tests, aiming to constantly improve the quality of the results. The interaction between the Test Controller and the individual testing tools is done through specialized plugins. The Controller has a high-level view on the testing process, leaving the details of each particular tool to the plugins. The meta-heuristic algorithm in the Controller is based on the intuition that there is inherent structure in the explored hyperspace, and this structure can be exploited to improve the exploration. Engler et al. also discovered that bugs in software are correlated [7], while Inkumsah and Xie showed the benefit of using Genetic Algorithms (another meta-heuristic exploration algorithm) to improve the quality of method sequence generation for concolic execution [**?**].

Data: Π: current set of top-impact results,
Ψ: queue of test scenarios pending execution,
Ω: history of previously executed tests,
μ: maximum observed impact so far

1 $parent := sample(\Pi)$
2 $plugin := sample(parent.plugins)$
3 $mutateDistance := 1 - parent.impact/\mu$
4 $newScenario :=$
$plugin.mutate(parent, mutateDistance)$
5 **if** $newScenario \notin \Omega$ *and* $newScenario \notin \Pi$ **then**
6 $\quad \Psi := \Psi \cup newScenario$
7 **end**

Algorithm 1: Test generation algorithm

A sketch of the algorithm in the Test Controller is shown in Algorithm 1. The algorithm begins by choosing a previously executed test scenario, at line 1. The parent test scenario is sampled from the set Π based on the impact; thus, test scenarios (attacks) that have had a large impact on system performance will be chosen more often than those with little impact. Once a parent test has been chosen, the algorithm needs to decide which parameter of the test scenario to change. It delegates this task to a plugin, since the plugin has deeper knowledge on the meaning of the parameters. The algorithm samples a plugin based on the historical benefit of choosing each plugin (line 2); if a plugin yields an increase in impact over the parent whenever it is selected, then it will be selected more often (this approach is similar to the fitness-gain computation used in Fitnex [16]). The new scenario is obtained by asking the selected plugin to mutate the parent scenario (lines 3-4). The *mutateDistance* represents how "strong" the mutation should be. A "strong" mutation would lead to a child scenario significantly different from its parent, while a "weak" mutation would lead to only small variations. The *mutateDistance* is computed by comparing the fitness of the parent with the maximum fitness seen so far in the experiment. If the fitness of the parent is significantly lower, the parent is not very promising, therefore the child should differ significantly from the parent. If, on the other hand, the parent is promising, the algorithm only "fine tunes" the parent test parameters. Finally, if the resulting test scenario has not been previously explored, it is added to the Ψ queue. A worker thread dequeues scenarios from Ψ, instantiates the test configuration (using the plugins), executes the test and computes the impact (e.g., by calling performance analysis tools).

4. Power of an Attacker

AVD also enables the tester to estimate the knowledge necessary to an attacker to find weaknesses in a distributed system. The various testing tools coordinated by AVD use various levels of access to the inputs/binaries/code (e.g., fuzzers use nothing, fault injectors use documentation, symbolic execution uses code etc.). There are two dimensions in the "power" of the tools at hand.

First, there are various levels of access to the distributed system nodes. Without access to any source, binary, or documentation, an attacker (and AVD) can only resort to random bit flips, random fuzzing, or to random packet drops/reordering. Given access to documentation, it is possible to infer the grammar of the protocol and be smarter about the fuzzing or bit flips. With access to binaries, the attacker can perform static analysis in order to gain knowledge about the various execution paths and find corner cases in the protocol implementation. Finally, with access to the source code, the attacker can gain complete knowledge of the protocol and discover any vulnerabilities in the system. The attacker can use symbolic execution, for instance, to find all execution paths through the distributed system and exploit those paths that lead to failures.

Second, there are various degrees of control the attacker can have on the distributed system. It can gain control of client nodes, of part of the network infrastructure, or even of server nodes.

Similarly to a real attacker, AVD finds vulnerabilities faster as it has more power over the target distributed system. Thus, the number of tests necessary for AVD to find a vulnerability is an indication of how difficult it would be for a real attacker to find similar vulnerabilities, given the same amount of power. Although not completely accurate, as real attackers have prior experience that cannot be directly input in AVD, this approach can be used as a rule of thumb when prioritizing bug fixes.

5. Tools

In this section, we give a brief overview of various classes of test tools that can be used in AVD and describe how their associated plugins can implement the

mutateDistance parameter described in (§3) in order to "focus" the testing towards relevant vulnerabilities.

Symbolic execution is a testing technique used to exercise various code paths through a target system. In a nutshell, symbolic execution works as follows: instead of running the target system with concrete input values, a symbolic execution engine replaces the inputs with symbolic variables, that are initially allowed to be anything, then runs the target system. Whenever the system execution branches based on a symbolic value (that depends on symbolic inputs), the symbolic execution engine forks, following each branch and adding constraints on the symbolic variable in the branch node. Thus, each execution path through the target system will have associated a set of constraints on the symbolic inputs that need to be satisfied in order for the execution path to be feasible. The set of constraints can be "solved", generating a set of concrete inputs that would exercise the respective path.

Symbolic execution of a node in distributed system finds all the messages that the node may produce, enabling AVD to evaluate the response of the correct nodes in the system to every combination of messages. In order to synthesize malicious nodes, the consistency models in the symbolic execution (the strictness with which constraints are obeyed) can be relaxed, thus generating sequences of messages that would not normally be allowed by the code; for instance, in the case of PBFT, a malicious replica could send a "View Change" message without actually suspecting the primary of being malicious.

For symbolic execution tools, a large *mutateDistance* parameter means obtaining a mutated execution path that has very little in common with the its *parent*. The plugin can keep track, for each branch node, of the average "disparity" between the two sub-trees the branch produced so far (e.g., in terms of basic block coverage). When the plugin receives a mutate command with a small *mutateDistance*, it will flip branch nodes that are known to produce little "disparity" and vice-versa.

AVD could use concolic execution tools, such as Cute [13] and Fitnex [16], or selective symbolic execution tools, such as S2E [5]. Concolic execution tools have the advantage of producing one complete execution path at a time, a model that is very close to the iterative exploration used by AVD. Selective symbolic execution, on the other hand, allows relaxing the consistency model of the symbolic execution, giving AVD more control over the target nodes, enabling more "arbitrary behavior" of malicious nodes.

Message reordering is a technique to re-arrange packets in a network. Many distributed systems use asynchronous communication, where the order of incoming messages is not guaranteed. Therefore, vulnerabilities may hide in the order in which messages are received.

In the case of message reordering, the *mutateDistance* can be reflected in the edit distance (Levenshtein distance) between two streams of messages. A strong mutation (large *mutateDistance*) would lead to a high edit distance between the original stream of messages and its mutation, while a weak mutation would lead to a low edit distance.

MODIST [17] is a tool that verifies a distributed system against various events related to message timing. Guerraoui et al. [8] also propose an approach that separates the message ordering from the local state of each node during model checking.

Fault injection is a form of testing that entails introducing faults in a system, with the goal of exercising error-handling code paths. Fault injection tools generally simulate a fault in the environment of the system under test (in the Operating System, libraries, hardware, etc.) in order to assess the response of the system under test.

We also note that fault injection can be used together with symbolic execution, for a more efficient exploration of recovery code paths. KLEE [3] has an operation mode that allows the symbolic execution engine to also explore recovery code paths, essentially performing fault injection (however, these faults are restricted to the level of the application under test).

The *mutateDistance* can be reflected in the call number at which a fault is injected. A small *mutateDistance* means injecting in a neighboring call, while a large distance entails injecting further away than the parent.

There are several fault injection tools available, targeting various aspects of the system environment. LFI [11] makes it easy for testers to write custom fault injection scenarios. Gunawi et al. [9] describe a framework for systematically injecting sequences of faults in a system under test. WebErr [1] can also broaden the scope of AVD towards testing Web Applications.

6. Preliminary Experiments

In this section we present some preliminary results that show AVD's ability to find vulnerabilities. We chose to evaluate PBFT [4], a well-known Byzantine Fault Tolerant system. AVD was able to reproduce a previously known vulnerability of PBFT, as well as discover a new attack that brings PBFT throughput down to 0. In this preliminary experiment we focused on a single test tool, MAC corruption, in order to validate our approach on previously discovered bugs. We show that: 1) AVD finds bugs; 2) there is structure in the hyperspace of test scenarios; 3) AVD exploits that structure in order to improve the efficiency of the exploration. Although our preliminary experiment is limited, we expect similar results for other testing tools as well; Engler et al. [7] and Inkumsah

and Xie [**?**] show that similar structure exists in various classes of software testing.

Inspired by Aardvark [6], we set out to check if AVD can find the Big MAC attack described by Clement et al. [6]. We deployed PBFT on Emulab [15] and set up a fault injector that corrupts the Message Authentication Code (MAC) in the client nodes. AVD controls three test parameters: which MACs to corrupt, how many correct clients to connect to PBFT and how many malicious clients to connect to PBFT. Each parameter corresponds to a dimension in the hyperspace explored by PBFT. The dimension corresponding to which MAC to corrupt has $4,096$ values (see below for explanation), the number of correct clients varies between 10 and 250, with an increment of 10 (25 values), while the number of malicious clients is 1 or 2. Thus, AVD explores a hyperspace with $4,096 * 25 * 2 = 204,800$ possible scenarios. The performance impact of an attack is obtained by measuring the average throughput observed by the correct clients.

The parameter describing which MAC to corrupt is a 12-bit-wide bit mask, where bit n decides whether to corrupt or not the (n mod 12)-th call to the generateMAC function in the malicious client. In order to implement the *mutateDistance* parameter, the 12-bit number is encoded in Gray code. Thus, a small *mutateDistance* entails choosing a neighboring value (in Gray code, consecutive numbers always differ in only one binary position).

Our experiment confirms the Big MAC attack. AVD shows that by corrupting the MAC in all messages sent by a malicious client, PBFT will perform a view change and crash. Figure 2 shows the evolution of the performance impact in scenarios generated by the fitness-guided exploration of AVD, versus random exploration. It can be seen that AVD finds more efficient attacks by exploiting feedback. AVD finds an instance of the Big MAC attack in a few tens of iterations (on the order of minutes).

AVD has also helped us discover a previously undocumented bug in PBFT. We noticed that despite a malicious client altering the MACs for the replicas, the protocol did not always perform a View Change (specifically, if every retransmission from the malicious client was correct). The PBFT protocol specifies a timer associated to each request received by replicas directly from clients. If a replica receives such a message, it forwards it to the primary and starts a view change timer. If the respective message is not executed before the timer expires, the replica will initiate a view change (this ensures that malicious primaries cannot ignore clients forever). However, in the implementation of PBFT there is a single such timer, rather than one per request. If a message is received by a replica directly from a client, the timer is set. If *any* such message is executed before the timer expires, the timer is reset. Therefore, a malicious primary only has to execute one client request per timer period

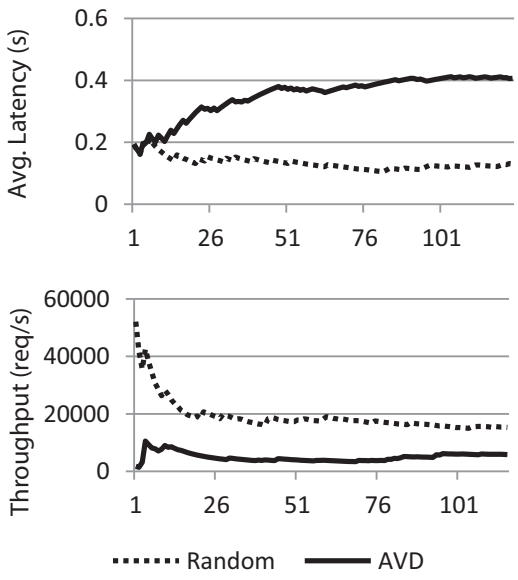

Figure 2. Evolution of average latency of requests from correct clients and of average throughput of PBFT system, as induced by attacks generated by the fitness-guided exploration of AVD, versus random exploration, over 125 executed tests

(5 seconds by default), diminishing PBFT throughput to 0.2 requests / second. If the respective client is also malicious, cooperating with the primary, the primary can ignore all messages from correct clients decreasing the useful throughput of PBFT to 0. This is a particular instance of "slow primaries", which Clement et al. [6] also broadly described; Aardvark avoids this bug by enforcing minimum throughput thresholds for each primary.

Figure 3 shows a subset of the hyperspace of possible test scenarios for the PBFT MAC attack experiment, allowing a visual inspection of the inherent structure exploited by AVD. We used an exhaustive approach to explore the entire subspace, in order to visualize its structure. Dark points represent test scenarios where the performance of PBFT is severely impacted by the faulty nodes (throughput is smaller than 500 requests/second). It can be seen that the subspace has both horizontal and vertical structure: there are several clearly defined vertical lines and they are clustered together on the horizontal axis. This structure makes the space suitable for exploration with hill-climbing.

7. Related Work

To the best of our knowledge, there are no other tools that take a similar approach to vulnerability detection. Existing security tools either focus on specific security aspects or test the system as a whole.

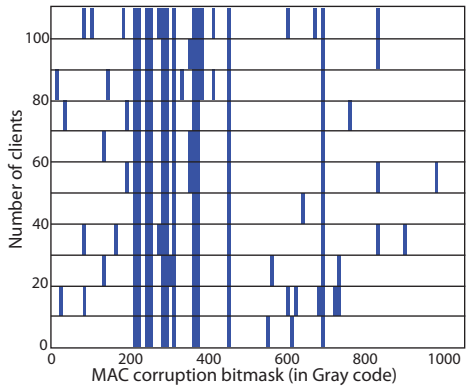

Figure 3. A subset of the hyperspace of possible test scenarios for PBFT MAC fault injection, exhaustively explored. Dark points represent scenarios where the throughput of PBFT drops below 500 requests/sec.

The Mace Performance Checker [10] and BFTSim [14] both add execution time to distributed system models. MacePC uses this information to find performance issues in the recovery code of various distributed systems, while BFTSim simulates byzantine fault tolerant systems under various network conditions. Unlike our work, both tools rely on an existing model of the target distributed system; the model can be buggy, incomplete or out of date.

McMinn surveyed various approaches towards the use of meta-heuristic search techniques in software testing [12], covering various heuristics and various test techniques. However, the survey focuses on exploring different execution paths through a single-threaded system, rather than distributed system vulnerability.

8. Conclusion

In this paper we described a technique for automatically discovering vulnerabilities in distributed systems, focusing on the impact that a small number of compromised nodes can have on a large distributed system. Our approach is to synthesize malicious nodes in a distributed system and assess the performance hit they incur on the correct, unmodified nodes of the system. We also presented AVD, a prototype implementation of our technique and used it to find two vulnerabilities in PBFT.

References

[1] S. Andrica and G. Candea. WebErr: High fidelity web application recording and replaying. In *Intl. Conf. on Dependable Systems and Networks*, 2011.

[2] Bittorrent dos vulnerability. http://events.ccc.de/congress/2010/Fahrplan/events/4210.en.html.

[3] 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.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

[5] V. Chipounov, V. Kuznetsov, and G. 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.

[6] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, 2009.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. Lake Louise, Canada, Oct 2001.

[8] R. Guerraoui and M. Yabandeh. Model checking a networked system without the network. In *Symp. on Networked Systems Design and Implementation*, 2011.

[9] H. S. Gunawi, T. Do, P. Joshi, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and K. Sen. Towards automatically checking thousands of failures with micro-specifications. Technical Report UCB/EECS-2010-98, UC Berkeley, 2010.

[10] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *Symp. on the Foundations of Software Eng.*, 2010.

[11] P. D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Annual Technical Conf.*, 2010.

[12] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[13] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Symp. on the Foundations of Software Eng.*, 2005.

[14] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Symp. on Networked Systems Design and Implementation*, 2008.

[15] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symp. on Operating Systems Design and Implementation*, 2002.

[16] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[17] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Symp. on Networked Systems Design and Implementation*, 2009.

[18] Michael Zalewski's security homepage. http://lcamtuf.coredump.cx/.