Toward Online Testing of Federated and Heterogeneous Distributed Systems

Marco Canini, Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić
School of Computer and Communication Sciences, EPFL, Switzerland
email: firstname.lastname@epfl.ch

Abstract

Making distributed systems reliable is notoriously difficult. It is even more difficult to achieve high reliability for federated and heterogeneous systems, *i.e.*, those that are operated by multiple administrative entities and have numerous inter-operable implementations. A prime example of such a system is the Internet's inter-domain routing, today based on BGP.

We argue that system reliability should be improved by proactively identifying potential faults using an online testing functionality. We propose DiCE, an approach that continuously and automatically explores the system behavior, to check whether the system deviates from its desired behavior. DiCE orchestrates the exploration of relevant system behaviors by subjecting system nodes to many possible inputs that exercise node actions. DiCE starts exploring from current, live system state, and operates in isolation from the deployed system. We describe our experience in integrating DiCE with an open-source BGP router. We evaluate the prototype's ability to quickly detect origin misconfiguration, a recurring operator mistake that causes Internet-wide outages. We also quantify DiCE's overhead and find it to have marginal impact on system performance.

1 Introduction

Internet's inter-domain routing, based on the standard Border Gateway Protocol (BGP), is a prime example of a distributed system that is fundamentally *federated* and *heterogeneous*. Multiple administrative domains autonomously control their own BGP routers and policies, while ensuring universal connectivity. Further, the open standards upon which the Internet is built allow for and promote numerous, mutually inter-operating implementations of BGP. Examples of other systems of such nature include DNS, electronic mail, peer-to-peer content distribution [10], content and resource peering [5].

Recent events have shown that Internet's routing falls short of ensuring reliable operation at all times. For example, Pakistan Telecom mistakenly managed to hijack the vast majority of traffic directed toward YouTube, making the popular website unreachable to many users for almost two hours [2][1].

In general, system behavior is the aggregate result of interleaved actions of system nodes, each of which is generally driven by code as well as configuration. Understandably, it is hard to reason *a priori* about every corner case and anticipate all possible combinations of system configurations. As a consequence, insidious bugs can survive until the system is deployed or configuration mistakes become a problem under certain unanticipated conditions — all these with dire consequences for the system's reliable operation.

We argued [9] that making heterogeneous and federated distributed systems reliable is challenging because (i) the source code of every node may not be readily available for testing and (ii) competitive concerns are likely to induce individual providers to keep private much of their current state and configuration.

Our overarching vision is to harness the continuous increases in available computational power and bandwidth to improve the reliability of distributed systems. In particular, we argue for an online testing functionality that strives to detect what node actions lead to potential faults (*i.e.*, deviations of system components from their expected behavior).

We have to address several difficult challenges (of which a more thorough account is in [9]). First, the federated nature of the systems we target give rise to a number of issues because of the different administrative domains desire to keep private their node states and configurations. Most importantly, no single node can have unrestricted access to remote node state and configuration.

[1] This problem persists to this day. China Telecom managed to hijack 10% of the Internet prefixes as recently as April 2010. Google's services were mistakenly hijacked in July and August of 2010 [1].

This affects how we can drive the exploration of system behavior and how we can check system-wide state to detect faults. This also hinders the possibility of simply applying existing approaches that drive exploration from the initial state (*e.g.*, [22]). In addition, we need to carefully consider what information crosses domain boundaries and how to preserve its confidentiality. Second, system heterogeneity makes it difficult if not impossible to have local access at one node to the source or binary code of other nodes. This difficulty and other constraints we mentioned above mean that we cannot use existing techniques for live model checking (*e.g.*, [21]) that operate locally. Last but not least, systematic exploration of system behavior or even single node behavior runs into the problem of exponential explosion of the number of possible node actions.

In this paper, we introduce DiCE, an approach that continuously and automatically explores the system behavior, to check whether the system deviates from its desired behavior. DiCE orchestrates the exploration of relevant system behaviors by subjecting system nodes to many possible inputs that systematically exercise node actions. To quickly reach relevant states and overcome problems with an exponential number of actions, DiCE starts exploring from current system state, while it operates in isolation from the deployed system.

We describe our general vision and outline the problem we want to address in DiCE (§2.1). We provide an initial design (§2.3) and discuss our experience in integrating DiCE with a BGP router (§3). We evaluate (§4) the prototype's ability to quickly detect origin misconfiguration, a recurring operator mistake that causes Internet-wide outages. We also quantify DiCE's overhead and find it to have marginal impact on system performance.

2 DiCE

We start by providing an overview of the problem that we want to address.

2.1 Problem overview

Our goal is to systematically explore system behavior so as to detect potential faults. At the same time, the approach that orchestrates the exploration of system behavior needs to accommodate the constraints of federated and heterogeneous environments.

A central question for reaching this goal is in understanding how to drive system behavior. We observe that distributed system behavior is the aggregate result of interleaved node actions. In turn, these actions are determined by the paths taken through the code running at the nodes that is driven by the configuration and the inputs. Therefore, to explore node actions, we want to subject the node's code to inputs that systematically exercise the node's possible actions. In other words, we need a mechanism that systematically exercises the node's code paths.

In practice, achieving extensive path coverage is greatly limited by the exponential explosion in the number of possible code paths. Given our desire to quickly detect potential faults, we would ideally just focus on covering relevant states. However, these are usually deep in the execution path. Recall that we target systems that are likely to run for a long time over which a large history of inputs accumulates. Thus, we need to avoid the need to replay a long history of inputs from initial state to reach a desired point in the code, as doing so can be prohibitively time-consuming.

An intriguing question is whether local testing of a single node is sufficient to detect such actions. While local testing is certainly a necessary step, we argue that it alone is not sufficient. In fact, local testing does not allow to observe far reaching consequences of single node actions. These consequences need to be observed from a system-wide perspective.

Therefore, we still need to be able to judge the system-wide consequences of node actions. In the general case, this cannot be done locally because a node does not know, and we assume cannot obtain the state and configuration, of other nodes. Also, a remote node could be running a different implementation. Effectively, we face the problem of how to let the local node communicate with remote nodes during exploration of behavior. However, we must not affect the deployed system and, at the same time, we need to support checking node states while preserving confidentiality of private information.

In summary, we are trying to address these questions: (i) how can we automatically exercise code paths? (ii) how can we enable code path exploration to guide state space exploration? (iii) how can we extend the horizon of local state space exploration to reach across the network? and (iv) how can we check for faults while preserving privacy between parties?

In this paper, we focus on the first two of these questions and provide a discussion around the other two. Before presenting our initial design, we proceed to briefly review certain software testing techniques that offer the basic mechanics necessary for systematically exploring a node's code paths.

2.2 Background

Symbolic execution (*e.g.*, see [8, 13]) is an automated testing technique that executes a program by treating the inputs to the program as symbolic. Upon encountering a branch that involves symbolic values, the symbolic execution engine creates the constraints that correspond to

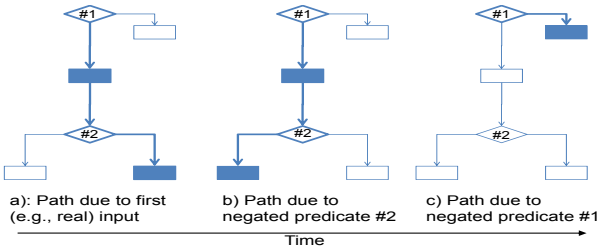

Figure 1: A concolic execution engine negates the predicates to try to systematically explore code paths (blocks that are executed in each run are shown as shaded).

both sides of the branch, and schedules execution to take place down both paths. It then queries a constraint solver to determine which paths are feasible, so that they can be explored. While symbolic execution is in theory capable of exploring all possible paths in the program, in practice it is severely limited as the number of paths to explore in an application grows exponentially with the size of the input and the number of branches in the code. A typical symbolic execution engine starts exploring paths from the beginning of the program and progressively explores all paths for which it can find suitable input values.

Concolic execution (*e.g.*, see [7, 11]) is a variant of symbolic execution that, instead of strictly operating on symbolic inputs, executes the code with concrete inputs while still collecting constraints along code paths. To drive execution down a particular path, the concolic execution engine picks a constraint (*e.g.*, branch predicate) and queries the constraint solver to find a concrete input value that negates the constraint. Figure 1 illustrates this process. The main benefit of concolic execution is the ease in interacting with the environment (due to the use of concrete values), and less overhead during execution than the "classic" symbolic execution (*e.g.*, only one call to the constraint solver for every branch).

2.3 Initial design

DiCE employs a concolic execution engine to solve the mechanical problem of exercising all possible code paths. Unlike standard concolic execution, DiCE starts exploring from the current, live state because of the desire to (i) quickly detect potential faults, and (ii) avoid the overhead of replaying execution from initial state to reach a desired point in the code (as we expect a large history of inputs).

First, DiCE takes a node checkpoint. Then, DiCE clones this checkpoint and feeds it with a previously observed input (*i.e.*, a message) to record the constraints that are encountered on the code path executed by invoking a message handler with that input. We rely on the programmer to identify message handlers and we only use those to process inputs for path exploration. This design decision lets us quickly zoom in on the relevant code, at the expense of requiring some developer involvement[2]. After completing the recording of these initial constraints recording, the concolic execution engine starts negating constraints one at a time, resulting in a set of inputs. To explore a particular input, DiCE makes a clone of the checkpoint, and then resumes execution with that input from the checkpointed state. The constraints encountered on the code path during execution with that particular input are once again recorded and used to update the aggregate set of constraints so far encountered. Updating the aggregate set is important for achieving full coverage, since the previous runs might not have reached all branches that exist in the code.

Note that we want the exploratory execution over a node checkpoint to work alongside the running system. Therefore, DiCE intercepts the messages generated during exploration.

2.4 Discussion

We consider the ability to explore node actions that we described as the initial building block for providing a full online testing functionality.

In fact, once we can locally exercise all possible node actions, we can then turn to how to observe their consequences on the system-wide state. We anticipate that we would let these actions result into messaging with other nodes in a way that doesn't affect the live system. For instance, we could intercept all messages and let them go through isolated communication channels. In addition, we would enable remote nodes to checkpoint their state and process these messages in isolation over their checkpointed states. Effectively, this would extend the scope of the concolic execution engine to reach across the network and exercise system behavior.

Ultimately, we want to check system-wide states for faults. We envision that, by having a notion of desired system behavior, we could check whether the system deviates from its desired behavior during the exploration with particular inputs. However, it is challenging to check system-wide states without compromising the confidentiality of private information. As we noted earlier, there cannot be unrestricted access to node states and configurations. In essence, we would want to control the information shared across domains and ensure that nodes only communicate state information through a narrow interface yet capable to allow us to detect faults.

[2]Given the great importance of the deployed federated systems, this effort is well-justified.

3 Experiences with the BGP use case

This section describes our DiCE prototype and details our experiences for integrating with the BGP implementation of BIRD [3] 1.1.7 open-source routing daemon. Because of space limit, we omit a review of BGP and point the reader to [14] for a succinct overview and to the RFC [20] for full details. We first introduce Oasis [11], the concolic execution engine we use.

3.1 Oasis

We use the Oasis concolic execution engine [11] as the basis for code path exploration. Oasis is a result of substantial modification of the Crest [7] concolic execution engine. Oasis instruments C programs using CIL [18] to be able to track at run-time the statements executed and record the constraints on symbolic inputs. Oasis handles the entire C language and supports interaction with the network and filesystem. Oasis has multiple search strategies, and it can execute multiple explorations in parallel. The default exploration strategy, which we use, attempts to cover all execution paths reachable by the set of controlled symbolic inputs.

3.2 Prototype implementation

Our DiCE prototype consists of a modified version of Oasis and a part written in C and integrated in BIRD.

We modify Oasis in three ways. First, we introduce support in Oasis to explore by resuming execution from a checkpoint instead of starting a new execution for each set of inputs. Second, we change the Oasis filesystem/network model to control the interactions of the program under test with the environment and ensure isolation from the running system. Third, we modify Oasis to allow both the original and the instrumented code to be automatically compiled and linked in a single executable, where they co-exist and operate on the same data in a similar fashion to the work by Anagnostakis *et al.* [6]. This enables the running systems to run with virtually no overhead, and only during exploration, which takes place off the critical path, switch to the instrumented code.

For integrating with BIRD, we first change its BGP implementation to mark certain inputs as symbolic. We choose to treat UPDATE messages as the basis to derive new inputs during exploration. In BGP, UPDATE messages are the main drivers for state change while the other state changing messages are only responsible for establishing or tearing down peerings and we leave them for future work.

A simple approach would be to mark an entire UPDATE message as symbolic. However, this has the effect of causing Oasis to produce a large variety of invalid messages that simply exercise the message parsing code[3]. This is undesirable for us because we want to explore node actions, and so we need to go deeper in the message processing code. As the format of BGP messages is well-defined in the RFC [20], we selectively define as symbolic small-sized inputs that directly derive from the message. For instance, the Network Layer Reachability Info (NLRI) region of the message contains the announced routes with their respective netmask lengths. We mark these as symbolic. An UPDATE message also typically carries multiple path attributes each of which is encoded as a type, length, and value fields that can also be marked symbolic. However, one needs to be careful that the symbolic length matches the actual length of the value field and that its semantic is consistent with the attribute type; otherwise the message is invalid and of no utility. Note that this approach is very effective in reducing the space of exploration because the produced messages are always syntactically valid.

In practice, this choice allows DiCE to construct inputs that exercise BGP behavior in two dimensions: the first due to BIRD's code implementing BGP, the second as the result of the particular configuration currently in use. This is because the source code instrumentation encompasses the BIRD's configuration interpreter and so allows Oasis to record constraints for the interpreted configuration. Therefore, the explored execution paths are comprehensive of both code and configuration. Finally, to enable Oasis to perform path exploration of BIRD's code, we handle the well-known cases that are difficult or impossible to handle in symbolic execution. For example, we avoid recording constraints that result from applying hash functions, as they cannot be reversed.

We make a second change to BIRD for taking a checkpoint. We implement this procedure by simply using the `fork` system call. This way of checkpointing allows us to create a large number of checkpoints with a small memory footprint. We are careful to isolate the forked process from its parent by closing the open sockets.

4 Evaluation

We use a 2.6 GHz 48-core machine with 64 GB of RAM, running Linux 2.6.30. Using virtual interfaces and multiple BIRD router instances, we install the 3-router topology shown in Figure 2. DiCE runs in the Provider's router. The DiCE-enabled router loads 319,355 prefixes from the "rest of the Internet" where we replay a BGP trace obtained from RouteViews (a full dump plus 15-min updates trace from route-views.eqix at April 1, 2010, 17:28 UTC). To be able to detect route leaks, we con-

[3] Which ought to be correct and could anyway be tested with a symbolic execution engine that targets single-machine code.

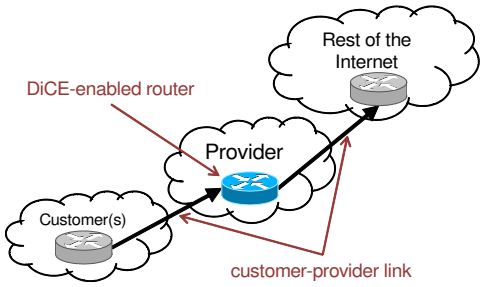

Figure 2: The experimental topology.

figure a partially correct route filtering. Customer route filtering happens in the provider and is a best common practice currently adopted by several large ISPs to defend against BGP prefix hijacking.

4.1 Performance impact

Here we provide a summary of the micro-benchmarks we run to understand how much DiCE impacts memory and CPU usage during exploration.

Memory overhead. We perform measurements that quantify the memory overhead on a BIRD router that has a full routing table loaded. We then run the exploration while the routers processing a 15 minute trace replay of BGP messages. The checkpoint process has 3.45% unique memory pages. The processes forked for exploring from the checkpoint process consume on average 36.93% pages more (maximum of 39%).

CPU/performance. We use the number of BGP update messages the DiCE-enabled router handles per second as a measure of how much the performance is affected while running exploration. The BIRD processes are configured to run on separate CPU cores, with the explorer having to share the single CPU core with its checkpoints that are used during exploration.

Under full load (running the exploration while loading the routing table), the BIRD process manages 13.9 updates per second. Without exploration, in the same interval of time during the trace replay, it is handling 15.1 updates per second. Thus, the performance impact even in this most stressful case is still small, namely 8%.

In a different, more realistic scenario, we run the exploration a few minutes inside the replay of a real-time trace of 15 min (after the full routing table was loaded). In this case the difference is negligible, with the BIRD process managing 0.272 queries per second during exploration and 0.287 when free to use the full CPU core.

4.2 Detecting route leaks

In a highly publicized router misconfiguration incident, Pakistan Telecom (an ISP) managed to divert to itself and drop the vast majority of traffic directed toward YouTube, the popular video-on-demand website. Consequently, this important service was unavailable for almost two hours [2]. In this particular case of BGP misconfiguration called prefix hijacking, there were two compounded errors that caused the fault. First, Pakistan Telecom announced a route that it had only intended to blackhole (*i.e.*, make YouTube unavailable to Pakistani residents). Second, PCCW, the upstream provider for Pakistan Telecom, did not have filters installed to limit the spreading of this announcement.

To replicate the IP prefix hijacking problem in our testbed, we misconfigured customer route filtering at the Provider AS. That is, its policy either fails to filter customer routes or has erroneous filters. Then, DiCE locally exercises all possible execution paths, which also include the "if" statements in the configured filters. For each exploratory message, we check whether the announced route (as determined by Oasis's manipulation of the NLRI) is accepted, and in this case we detect a potential hijack if that route overrides the origin AS of a route already in the routing table prior to starting exploration[4]. Certain prefixes are "hijackable" by nature, *e.g.*, those used for IP anycast, and would appear as false positives. DiCE can simply filter these out once it is made aware of the IP anycast address space.

The benefit from running DiCE for a network operator is significant, as DiCE clearly states which prefix ranges can be leaked. In the case of YouTube hijacking, Pakistan's upstream provider would have been able to install a correct filter.

5 Related work

CrystalBall [21], and MODIST [22] represent the state-of-the-art in model checking distributed system implementations. CrystalBall [21] proactively predicts inconsistencies that can occur in a running distributed system due to unknown programming errors, and effectively prevents them. MODIST [22] is capable of model checking unmodified distributed systems.

Symbolic [8, 13] and concolic execution [11, 7] are techniques for achieving complete coverage of possible code paths and are effective in discovering bugs for single-machine code. In DiCE, we leverage concolic execution as the base mechanism for exploring distributed system states starting from covering possible node actions and ultimately judge their system-wide impact.

Collaboration among a program's or a system's stakeholders has been used in similar contexts. For example, Liblit *et al.* [15] proposed an approach that infers bugs by gathering information from the program's users. A

[4]This assumes that the existing routes are trustworthy.

sampling technique is used to maintain a low instrumentation overhead. Orso *et al.* [19] suggested an approach for continuously analyzing deployed software with minimal instrumentation with the goal of improving software quality. In the context of collaborative security, Locasto *et al.* [16] proposed that members of an application community share the burden of monitoring for software flaws and attacks, and notify the rest of the community when such are detected.

Nagaraja *et al.* [17] focused on operator mistakes in Internet services and argued for the creation of an online validation environment to be used to check operator actions before they are made visible. We consider their approach complementary to DiCE, in that our approach could be extended to explore system behavior under specific operator actions before they are introduced in the running system. In the case of a single ISP's routers, Alimi *et al.* [4] proposed to install an alternative configuration with which network operators can test proposed changes before committing them to the production network. Feamster *et al.* [12] demonstrated the effectiveness of static analysis to look for faults in the set of router configurations, but cannot check live node states.

6 Conclusions

In this paper we argued for leveraging the increases in computational power and bandwidth to make federated and heterogeneous distributed systems more reliable. We presented a preliminary design of DiCE, a system that systematically exercises node behavior with the goal of ultimately checking the system-wide impact of each node behavior. We described our experience in integrating a path exploration engine with an open-source BGP router written in C. We also outlined the challenges in extending this kind of online testing to reach across the network. Finally, we demonstrated our prototype's ability to detect BGP route leaks - an important class of configuration errors that plagues the Internet.

Acknowledgments.

We thank the anonymous reviewers and our shepherd, Angelos D. Keromytis, for their helpful comments and suggestions. We are grateful to Jennifer Rexford, Katerina Argyraki and Jon Crowcroft for their feedback on earlier drafts of this work. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110 and the Hasler foundation (grant 2103).

References

[1] Google's services redirected to Romania and Austria. `http://bgpmon.net/blog/?p=314`.

[2] Pakistan hijacks YouTube. `http://www.renesys.com/blog/2008/02/pakistan_hijacks_youtube_1.shtml`.

[3] The BIRD Internet Routing Daemon. `http://bird.network.cz`.

[4] R. Alimi, Y. Wang, and Y. R. Yang. Shadow Configuration as a Network Management Primitive. In *SIGCOMM*, 2008.

[5] L. Amini, A. Shaikh, and H. Schulzrinne. Effective Peering for Multi-provider Content Delivery Services. In *INFOCOM*, 2004.

[6] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *USENIX Security Symposium*, 2005.

[7] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.

[8] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[9] M. Canini, D. Novaković, V. Jovanović, and D. Kostić. Fault Prediction in Distributed Systems Gone Wild. In *LADIS*, 2010.

[10] B. Cohen. Incentives Build Robustness in BitTorrent. In *P2PECON*, 2003.

[11] O. Crameri, R. Bachwani, T. Brecht, R. Bianchini, D. Kostić, and W. Zwaenepoel. Oasis: Concolic Execution Driven by Test Suites and Code Modifications. Technical Report LABOS-REPORT-2009-002, EPFL, 2009.

[12] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*, 2005.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[14] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when Interdomain Routing Goes Wrong. In *NSDI*, 2009.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. In *PLDI*, 2003.

[16] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *SNDSS*, 2006.

[17] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI*, 2004.

[18] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, 2002.

[19] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma System: Continuous Evolution of Software after Deployment. In *ISSTA*, 2002.

[20] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4) IETF RFC 4271. 2006.

[21] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[22] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.