Checking for Insidious Faults in Deployed Federated and Heterogeneous Distributed Systems

EPFL Technical Report EPFL-REPORT-164475

Marco Canini, Vojin Jovanović, Daniele Venzano, Gautam Kumar, Dejan Novaković, and Dejan Kostić
School of Computer and Communication Sciences, EPFL, Switzerland
email: `firstname.lastname@epfl.ch`

Abstract

It is notoriously difficult to make distributed systems reliable. This becomes even harder in the case of the widely-deployed systems that are heterogeneous (multiple implementations) and federated (multiple administrative entities). The set of routers in charge of the Internet's inter-domain routing is a prime example of such a system. In this paper, we argue that a key step in making these systems reliable is the need to automatically explore the system behavior to check for potential faults. We present the design and implementation of DiCE, a system for online testing of heterogeneous and federated distributed systems. DiCE runs concurrently with the production system by leveraging distributed checkpoints and isolated communication channels. DiCE orchestrates the exploration of relevant system states by controlling the inputs that drive system actions. While respecting privacy among different administrative entities, DiCE detects faults by checking for violations of properties that capture the desired system behavior. We demonstrate the ease of integrating DiCE with a BGP router and a DNS server, the building blocks of two vital services in the Internet. Our evaluation in the testbed shows that DiCE quickly and successfully detects three important classes of faults, resulting from configuration mistakes, policy conflicts and programming errors.

1 Introduction

Many successful distributed systems are inherently heterogeneous and federated — heterogeneity arises from the creation of multiple, inter-operable implementations; federated refers to the existence of multiple service providers operating under different administrative domains. The Internet's inter-domain routing system governed by BGP is a prime example of a heterogeneous and federated system. Other such systems include DNS, electronic mail, peer-to-peer content distribution [16], content and resource peering [8], computing grids, and Web services. However, the resulting competing environment of mutually mistrusting providers fosters a tension between a provider's own goals versus the common overarching desire of keeping the federated system functioning properly.

In such an environment, making distributed systems reliable does not stop with the already difficult task of producing a robust design and implementation. Achieving high reliability also bears the difficulties in deploying and operating these systems whose aggregate behavior is the result of interleaved actions of multiple system nodes running in a heterogeneous and failure-prone environment. In fact, several factors such as subtle differences in the details of inter-operable implementations, or system-wide conflicts due to locally admissible (mis)configurations can cause harmful node interactions that lead to faults, i.e., *deviations of system components from their expected behavior.* These faults which span the state and configuration across multiple nodes are perhaps less frequent than single-machine bugs, e.g., memory-related issues. However, when these faults manifest themselves they have far-reaching and substantial negative impact, and require considerable resources to be diagnosed and eliminated.

For example, a BGP router can rightfully decide to reset its peering session in response to a syntactically valid, but semantically ambiguous message. However, when many of such routers are coupled with another large number of routers that propagate the ambiguous message (because of a different message parser implementation), the overall effect is a large fraction of routers that are continuously resetting and restoring sessions as it happened in several episodes [4]. The resulting high update processing rate causes a performance and reliability problem. Others have argued that a malformed packet could take down a significant fraction of the Internet [1]. Even with a 100% protocol-compliant message, such an incident inadvertently occurred in August of 2010 [3].

Our overarching vision is to harness the continuous increases in available computational power and bandwidth to improve the reliability of distributed systems. We argue that nodes in the system and their administrators should be proactively working towards finding which node actions could potentially lead to faults. This task cannot be done only locally by checking the single-node behavior, as the erroneous system state can span multiple nodes and remote node configurations are not available locally. Thus, detecting faults in the general case requires some collaboration among the nodes. The faults these actions lead to are evidence of possible future system failures which may be avoided by detecting these potential faults.

To detect faults, we propose to automatically explore the system behavior alongside the production system, but in complete isolation from it using a system snapshot captured from the current state. That is, we check system-wide consequences of actions nodes can undertake, and output actions that lead to failures. In practice, node actions are the result of subjecting the node's code in its current state to messages, configuration changes, failures, random choices, etc., collectively called *inputs* in the following. Ideally, we want to subject nodes to a large number of possible inputs that systematically exercise their behavior.

We have to address several difficult challenges [13] in our work. The federated nature of many deployed systems means that a node cannot gain unrestricted access to other nodes' state and configuration. Moreover, we have to carefully manage the information flowing between system participants to preserve their confidential nature. The heterogeneity of the system makes it difficult or impossible to have local access to the source or binary code of other participants. Systematically exploring node behavior even for a single node easily runs into the problem of exponential explosion in the number of code paths that need to be explored. Finally, the sheer size of the system can pose scalability problems.

Static analysis of configuration files [19] cannot be applied to this problem because it does not take into account the actual state and software of the system. Tools for predicting inconsistencies using live model checking (e.g., [44]) cannot be used because they require a node to (i) retrieve checkpoints (with private state and configuration) from other participants, and (ii) obtain access to the source code of other participants. Applying systematic source code exploration tools based on symbolic execution from initial state [11, 39] cannot explore code paths sufficiently deep due to exponential growth in the number of possible paths caused by having large inputs (configuration and messages received over a long time).

In this paper, we introduce DiCE, a system for online testing of heterogeneous, federated distributed systems. Accounting for the federated nature of the system, we let each node autonomously explore its local actions. We use a set of lightweight node checkpoints to allow the single node's actions reach out to other nodes as a way to drive and explore system-wide state in isolation from the production environment. To preserve privacy between different administrative domains, we define a narrow information sharing interface that enables a node to query remote nodes for relevant state checks. We detect faults by checking and flagging violations of given properties that, tying together state checks over multiple system nodes, capture the desired system behavior. These features enable the basic "what-if" exploration block that can be used in three different ways. First, it can be used to test for the outcome of a single input or configuration change. Second, to enable checking for insidious faults due to corner cases, remote node failures and different local choices, we enable the developer to encode these choices in a simple function that DiCE can then exercise. Finally, to systematically exercise possible node actions and check for the faults they might induce, we use a technique called *concolic execution* [10, 17, 23] to automatically produce the inputs that explore all possible code paths at one node. We overcome the problem of exponential explosion of code paths by starting exploring the node behaviors from current system state, and by subjecting the node's code to small-sized inputs that affect localized parts of state-changing code.

We demonstrate DiCE's ability to detect insidious faults in two systems that provide fundamental services: BGP and DNS. By doing so, we demonstrate the benefits of having a framework that enables system operators to specify the desired behavior in the form of safety properties, and learn about possible faults and their impact.

DiCE is a crucial step in being able to guard against important classes of faults. Advance warnings can be used to notify the system operator(s) about a particular misconfiguration, or to trigger automatic installation of a filter against the problem caused by the software reaction to an unanticipated message. A particular benefit of our approach is that the separate administrative entities can use DiCE by integrating only their source code with it, and without requiring access to the source code, executable, or configuration of other participants.

The contributions of this paper are as follows:

1. We describe the design and implementation of DiCE, a system for detecting possible faults in heterogeneous, federated environments that imposes light overhead and resource requirements.

2. We provide a technique for automatic and lightweight distributed snapshot creation that (i) respects trust boundaries, and (ii) allows system behavior to be explored in isolation. This technique effectively enables node actions to extend their reach across the network to

explore relevant system state. We believe that this primitive can be successfully applied to other "what-if" exploratory scenarios.

3. We demonstrate how a small amount of input-producing code can be used to drive execution across the relevant federated distributed system states. Doing so uncovers faults due to events that are difficult to explore, such as remote node failures. To the best of our knowledge, this is the first such approach.

4. We integrate DiCE with the BIRD [5] open-source router written in C. In our evaluation on the network testbed with Internet-like conditions, we demonstrate that DiCE quickly detects two important classes of faults that have affected the Internet: (i) Internet-wide BGP session resets, and (ii) policy conflicts among ISPs.

5. We integrate DiCE with the MaraDNS [2] open-source DNS server, and demonstrate its ability to detect cyclic zone dependencies – an insidious type of DNS misconfiguration that can render entire domain names unresolvable [37].

2 Design

To detect faulty states (those in which the system components deviate from their desired behavior), our goal is to continuously and systematically explore the behavior of a distributed system. In this section, we first offer an overview of how DiCE meets this goal. We then discuss each aspect of our design in detail, together with its rationale and principles.

Overview DiCE runs online, alongside a deployed system, off the critical execution path. Figure 1 gives a high-level illustration of how DiCE tests running distributed systems. First, one node in the system acts as an *explorer* (node marked with a double ellipse in step 1 of Figure 1). The explorer triggers the creation of a *shadow snapshot*, i.e., a consistent and distributed snapshot composed of nodes' local checkpoints based on the current state of the system (step 2). Then, DiCE exercises a variety of local behaviors at the explorer that result in exploring system-wide relevant states. As detailed later, the code, current state, and inputs fed to the code determine how a node behaves. Thus, DiCE uses a combination of techniques to carefully construct the inputs that systematically explore node behavior.

The execution of each node behavior occurs in isolation, over a clone of the shadow snapshot. The messaging is also confined to the cloned snapshot. Each cloned snapshot represents one instance of possible system behavior involving multiple nodes. DiCE detects faults by checking for violations of given safety properties in each cloned snapshot.

Using DiCE DiCE enables three modes of online testing. First, it allows for a single input or configuration change to be presented at the explorer (basic "what-if" building block). Second, it allows for systematic exploration of federated system states. DiCE user does this by writing and calling from the code a *choice* function that locally mimics remote node and network failures, random choices, etc., and writes the code that uses the return value to perform one of many possible actions. This type of exploration involves explicitly enumerating and exploring multiple individual inputs or configuration changes, and it uses the basic "what-if" block for each such case. Third, DiCE can systematically and automatically explore the code paths for relevant message handlers at the explorer. This feature is highly relevant in cases in which the code dominates node behavior, e.g., when a configuration file is interpreted on-the-fly or when message parsing code can create failures. This mode effectively allows the explorer node to judge its potential system-wide impact.

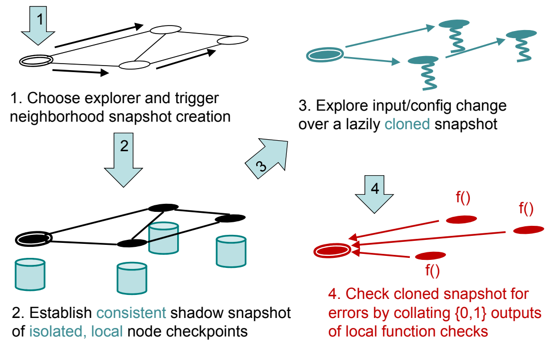

Figure 1: DiCE explores and checks system behavior over isolated snapshots. DiCE determines the result of each exploration by summing up 0 or 1 outputs of functions that run locally.

2.1 Exploring system behavior in isolation

In this section we describe the basic building block of DiCE: testing with the single input or configuration change, and gauging its impact across the neighborhood on a node in a manner that respects privacy. DiCE does this in isolation from the production system, starting from a consistent snapshot of its current state [33].

2.1.1 Checkpointing state across nodes

Despite all the best efforts in thorough local testing and configuration checking, there is no substitute for having the ability to inspect distributed system state for potential faults. This is challenging because the federated nature of the systems we target makes it impossible to simply retrieve state checkpoints from other nodes. Moreover, it may be impossible to have the exact copy of the software running at other nodes, as the system is fundamentally heterogeneous. Finally, the entities controlling different nodes might not be willing to reveal their configurations.

A snapshot that respects trust boundaries Presented with these constraints, we decide to let nodes keep their state, code and configuration in a local checkpoint. A checkpoint never leaves the node that creates it. However, the checkpoint has the ability to clone itself and resume execution from its saved state and to communicate with checkpoints belonging to other nodes. This way, a node that wishes to explore its behavior can do so by creating and executing inside a shadow snapshot, i.e., a consistent, distributed set of individual node checkpoints of the explorer's neighborhood.

The explorer establishes a snapshot by sending an annotated message to its immediate neighbors, which forward the message further on to their neighbors, etc. until a desired scope is reached.

Isolating execution in a snapshot To prevent the exploratory executions from changing system state, each node checkpoint is isolated from its environment. For example, all outgoing messages are intercepted and, instead of being transmitted over existing connections, are sent over *shadow connections* that the checkpointed node creates to the message destinations.

2.1.2 Detecting faults while respecting privacy

We detect faults by checking for violations of safety properties in the cloned snapshots. These properties are user-specified and we assume that they capture system-specific invariants or describe the desired system behavior. Some systems were designed with these types of properties in mind. When that is not the case, the properties can capture the best system practices and the post-mortem analysis of insidious faults that have previously occurred, as is the case with BGP [25]. In addition, approaches exist to automatically infer safety properties in distributed systems [43].

Checking properties across domains Let N be the set of nodes, and $\Theta_i, i \in N$ denote the set of node's i states executing in the cloned snapshot. A property, or global check, is expressed as a function $g(\Theta_1, \Theta_2, \cdots, \Theta_{\|N\|}) \in \{0, 1\}$. Note that a global check considers system-wide behavior and may potentially require accessing information at multiple nodes in different administrative domains.

To control the information shared across domains, we introduce a narrow interface. We consider a subfamily of global checks for which $g(\Theta_1, \Theta_2, \cdots, \Theta_{\|N\|}) = 1$ if $[\sum_{i \in N} f(\Theta_i)] > th, 0$ otherwise; where $f(\Theta_i) \in \mathbb{N}_0$ is a check that only accesses local state and th is a property-specific threshold (e.g., 0).

In this scheme, a centralized entity (e.g., the explorer) computes the global check as the sum of local check values[1]. To preserve privacy, the output of a local check should not contain any private information. For example, local checks can be written as: was there a certain change in the node's state? However, we anticipate there could be cases when individual domains are not willing to disclose local checks unless anonymity can be guaranteed, e.g., if a local check necessarily leaks private information. At the expense of increased computational complexity, we can control information sharing by securely summing local check values so that only the final outcome is known to participating nodes and single addends are not known. Appendix A discusses one such scheme for providing anonymous property checks.

2.2 Driving exploration of system behavior

The key step in detecting potential faults is to explore possible system behaviors. In practice, the aggregate behavior is the result of interleaved actions of multiple system nodes. To explore system behavior under different scenarios, we could take a position atop the system from where we would control all individual node actions and their interleaving[2]. Unfortunately, this principle would create the need for a third party responsible for orchestrating state exploration in the targeted federated system. Also, when considering a large-scale system, several scalability issues would arise.

Because we want to let the nodes (and administrative domains) maintain control of how they participate in the system state exploration, we propose a different principle — focus on local actions of one node and let the exploration of a single node's behavior reach out to other nodes as a way to explore system state. This kind of exploration can take place one node at a time, in parallel, or a combination thereof.

What drives node behavior In practice, the behavior of each node is determined by the path taken through its code. Keeping in mind that we resume execution from a local node checkpoint, we note that the code that will execute next is affected by (i) the current state and (ii) what we collectively term as inputs. As illustrated in Figure 2, the inputs encompass a variety of sources and events: e.g., messages, configuration changes, timers. Other less explicit inputs are events such as node failures and random choices. Next, we discuss the three modes of DiCE's operation that progressively cover an increasing number of possible inputs.

2.2.1 Testing a single input or configuration change

Testing a single input or configuration change is straightforward, and reuses the basic "what-if" exploration building block. This testing mode is useful for quickly checking whether a particular configuration change is

[1] A global check is decomposed into local checks; this might require an ad-hoc protocol as we show later for detecting policy conflicts.

[2] Commonly done for model checking distributed systems [30, 45].

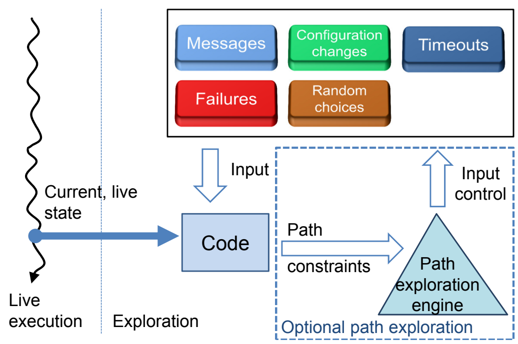

Figure 2: DiCE explores executions from live state, and provides a number of means of controlling inputs that it can use to explore system-wide behavior. Exploration occurs in parallel with the production system.

safe, or whether a change to a message will not have detrimental effect elsewhere in the system.

The node serving as the explorer, runs a *controller*, which starts by demanding the creation of a shadow snapshot (Section 3 describes the mechanism we adopt in our prototype). To explore a particular input, the controller instructs the explorer's shadow checkpoint to clone itself, and then resumes running from this cloned checkpoint. Once the exploration with a particular input completes[3], the cloned snapshot is checked for faults as described before.

2.2.2 Testing under network failures and different local choices

The second mode of testing is useful for examining system behavior under a variety of conditions and corner cases that might be difficult to test otherwise: remote failures, sequence of random choices, etc. The goal here is to test for faults either prior to contemplating a configuration change, or in steady-state.

As shown in Figure 2, a variety of means can be used to enable DiCE to drive the explorer's behavior to reach relevant federated system-wide states. However, how to identify all these inputs is an open question which might not have a general solution. In our experience, we find that leveraging domain knowledge is an effective approximation. For instance, anticipating some of the discussion from the next section, we identify that a key aspect of DNS name resolution is the *random choice* in querying one of many possible name servers for a given domain name. This driver of node behavior is easy to recognize and, when encoded in a *choice* function that DiCE can use to effect a fault, configuration change, or some other input in general. By doing so, DiCE can explicitly enumerate possible federated system states that are the result of nondeterminism including remote node and network failures, random choices, etc. that is similar to the approach taken by explicit state model checkers [46] for single-machine code. Interestingly, property definitions may give hints as to what inputs need to be exposed. For example, persistent oscillations in BGP can be caused by conflicting policies at different administrative domains. Policies are encoded in router configuration. Treating a policy *configuration change* as an input enables to exercise the BGP route selection process and find potential conflicts lurking in the configuration.

2.2.3 Systematically exploring node behavior

Because node behavior depends on the inputs, we might want to explore a node's behavior by subjecting the node to a variety of possible inputs in a way that systematically exercises its code paths. Given that this process is bound to take longer, this model of testing is useful in steady-state of the system, when we are interested in using the time to proactively find as many faults as possible.

The literature presents us with a technique for this purpose (automatically generating test cases from the code itself). In software testing, symbolic execution [11] is a technique that explores all possible code paths in a program — symbolic execution treats the input variables of the program as symbolic inputs, and during execution collects the constraints that describe which input values can lead to a particular point in the code. Albeit powerful, this technique comes with significant program execution overhead and, more problematically, it does not easily interact with the environment due to the abstraction of symbolic values.

As these two aspects are crucial for testing a real system that runs over the network, we look at a variant of symbolic execution called concolic execution [10, 17, 23] which easily interacts with the environment and has less overhead. This technique 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 one constraint (e.g., branch predicate) and queries the satisfiability solver to choose a concrete input that negates the constraint. As shown in Figure 2, this technique for automatically generating the inputs is optional and ties seamlessly together with the rest of DiCE.

To exhaustively explore node behavior, we would ideally explore all possible paths at the each node. While concolic execution is in theory capable of exploring all possible code paths, in practice it is severely limited as the number of paths to explore grows exponentially with the number of branches in the code and the size and number of the inputs (think of each invocation of an if statement that checks for end of input as an additional branch). We discuss later in this section our insights for

[3] Deciding on the termination of a distributed computation is a well-understood problem [32] (Section 3.8.5).

dealing with this problem.

Exercising node behavior to test system states The explorer uses a previously encountered real input (e.g., a message) to record the constraints encountered on the code path executed with that input (e.g., by invoking a message handler). This initial set of constraints are then passed on to the concolic engine. After completing the initial constraint recording, the concolic engine starts negating predicates one at a time, resulting in a set of inputs, each of which satisfies a particular constraint. To explore a particular input and check for its outcome, the explorer uses the basic "what-if" mechanism (the shadow snapshot is made only once and then used for cloning).

The constraints during this new execution path at the explorer node are once again recorded and fed to the concolic engine, which then updates the aggregate set of constraints and keeps producing new inputs. 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.

When dealing with symbolic messages we observe that concolic execution easily ends up creating many invalid inputs that simply exercise the message parsing code. Also, if the message format allows for variable length fields, we note that concolic execution has difficulties in producing messages where such fields are shrunk or grown. Therefore, we find it useful to use grammar-based whitebox-fuzzing [22] which leverages knowledge of the message format to produce a large number of inputs that quickly pass validation checks. We apply the fuzzing code before the message handlers, and rely on the domain knowledge to identify these handlers.

Countering exponential explosion of code paths In addition to a thoughtful choice of symbolic inputs, we have two key principles for dealing with the path explosion and large input problems: 1) start the exploration from current system state (the shadow snapshot). Doing so eliminates the need to replay from initial state a potentially large history of inputs to reach a desired point in the code, and 2) explore behaviors that are a result of small inputs, both size-wise and in number. The intuition is to try to reach faulty states that are small deviations from current state rather than being more exhaustive with the associated exponential increase in states.

2.2.4 Preventing information leakage

Ideally, the data that is crossing the trust boundary among the nodes should not reveal any confidential information. At a high level, we observe that there are two main kinds of information that can be leaked: potential node behavior and configuration data.

Leakage of node behavior is a direct consequence of systematic code path exploration. We argue however that in a long-running system the behavior has already been revealed for at least the most common set of code paths.

Configuration data can be leaked if the executed code paths produce messages containing a direct copy of the configuration data or an indirect manipulation thereof from which the configuration data can be reverse engineered. However, using concolic execution aids in information hiding. When the concolic engine wants to negate a constraint, it can pick any random value that negates the constraint to drive execution. Thus, the randomized nature of these inputs limits this kind of information leakage. In addition, we can annotate what data is confidential and avoid recording constraints from the code that handles the confidential data so that it cannot leak into the inputs the concolic engine produces.

Additional measures can be taken, including: (i) rate limiting the exploration or responses to property checks, or (ii) refusing certain explorer nodes altogether in the absence of any trust. In our future work we will do a thorough study of the security-related aspects of DiCE.

2.3 Discussion

A number of issues, such as the (possibly parallel) order in which the nodes act as explorers, the size of the shadow snapshot, and the amount of resources devoted to exploration at each node are application-specific and orthogonal to this paper; we discuss them in more detail in [12]. Here we only note that it is possible to limit resource consumption during exploration using existing primitives on many platforms.

Limitations The types of faults we can detect are a subset of faults that can be detected in a general distributed system [26]. We do not attempt to verify algorithms, protocols, or the operating system of the node. We do not incorporate Byzantine faults. To help it deal with this type of faults, DiCE could directly benefit from schemes that ensure accountability [25]. Further, we only check for known classes of faults that are captured in the supplied system-specific properties. We rely on the programmer, experienced system operator, or an automated tool [43] to identify these properties.

As with any fault detection solution, the potential DiCE issues are false positives and false negatives. DiCE can exhibit false negatives if the given properties are not capable of discerning the faulty state. False negatives also arise when there exists no code path that the concolic engine can exercise with small inputs to reach a faulty state. False positives are less of a problem, as the live execution over the cloned snapshot is evidence of behavior that is the result of processing a particular input. However, the properties themselves should be defined in a way that avoids false positives.

Note that the set of inputs that systematically covers message handling code on one node might not result in full path coverage of other participants (when they run using the inputs they receive in the shadow messages). We cannot easily accommodate system-wide coverage because we would need to share constraints with remote nodes and we deem this unfeasible because of privacy considerations. However, our evaluation with BGP and DNS demonstrates that important classes of faults can be detected without having system-wide path coverage.

Finally, DiCE is not a bug-finding tool that could be used to pinpoint the location of programming errors. DiCE is neither a traditional fault detection tool in the sense that the faults it can find are not detected in the live production system, but rather by reaching faulty states in the cloned snapshots.

3 DiCE prototype and applications

This section discusses our DiCE prototype and its application to two federated, heterogeneous distributed systems: BGP and DNS. For each case study, we present a brief overview of the target distributed system and describe how we integrate DiCE with it.

Our prototype consists of a concolic engine, a part written in C and integrated with the target systems, and a Python implementation of the DiCE controller. We use the Oasis [17] concolic engine as the basis for code path exploration. Oasis instruments C programs to record constraints on symbolic inputs during program execution. We discussed in [12] the modifications we made in Oasis. These include support for exploring from current state and the ability to use a single executable where both the original and instrumented code co-exist for avoiding performance overheads in the deployed system while recording constraints during exploration. In addition, in this work we change the Oasis filesystem/network model to manage shadow connections.

3.1 Integration with BGP

Here, we present the BGP case study, and use it to describe the details of our DiCE prototype. We start by providing an overview of BGP. We then discuss the integration of DiCE with the BIRD [5] open-source routing daemon. BIRD is written in C and supports multiple routing protocols. It is in production use serving as a route server in several Internet exchange points.

3.1.1 BGP overview

The Internet consists of tens of thousands of domains, so-called autonomous systems (ASes). ASes are typically administered by Internet Service Providers (ISPs). While the ASes have freedom in choosing their intra-domain routing protocol, Border Gateway Protocol (BGP) [38] is the inter-domain routing protocol that acts as the glue that ensures universal connectivity in the Internet and is spoken at each border router.

Each BGP speaker maintains a routing table, or Routing Information Base (RIB) that associates a route to a network prefix with the next hop router and the list of ASes (AS_PATH) that needs to be taken to reach a given IP in that prefix. The routing information is distilled into a Forwarding Information Base (FIB) that is used to make packet forwarding decisions. BGP speakers establish their routing tables by exchanging UPDATE messages which *announce* routes (each composed of a prefix and a bitmask length) along with their corresponding attributes (e.g., AS_PATH) and/or *withdraw* routes that are no longer available.

Recently, the protocol has been extended to allow for 4-byte AS numbers [42], and thus the messages can carry the optional AS4_PATH attribute. Legacy routers that do not understand the 4-byte AS numbers do not attempt to interpret the new attribute and simply pass it along with their updates.

3.1.2 Implementation

For integrating with BIRD, we made a small number of changes that fulfill the application requirements for applying DiCE to BGP. Specifically, (i) we marked the symbolic inputs (only a few LoC), (ii) we added support for taking snapshots and managing shadow connections (about 1300 LoC), and (iii) we exposed certain properties based on the local state that are queried by the controller in order to detect faults (about 200 LoC). We now discuss each of the implementation details.

Inputs For the reasons given in Section 2.2.2, we choose to treat UPDATE messages and policy configuration changes 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. As the format of BGP messages is well-defined in the RFC [38], we apply grammar-based fuzzing [22] to the path attributes and we mark the Network Layer Reachability Info (NLRI) region of the message as symbolic. An UPDATE message can carry several path attributes each of which is encoded as a type, length, and value field. To fuzz message attributes, we create two symbolic inputs for each attribute present in the initial message[4]. With respect to the fuzzed message, we assign to these inputs the meaning of attribute presence and length, respectively. In other words, if Oasis picks a non-zero value for the first input we include the attribute, otherwise we remove

[4]Except for mandatory attributes which we cannot exclude as a message without them is an invalid input.

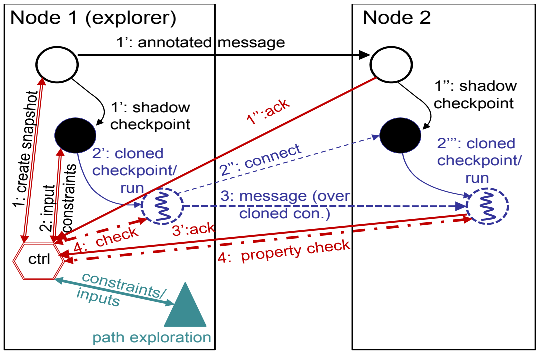

Figure 3: DiCE prototype in action. Thin lines correspond to the checkpoints being created from live state, dashed lines denote cloned state and connections, and the dash-dotted lines are DiCE controller actions.

it from the message; the attribute's length is matched to the second input. Therefore, Oasis can produce fuzzed messages based on all combinations that these symbolic inputs can have. In addition, Oasis can change the content of the NLRI based on the set of recorded constraints.

We define a further symbolic input that represents changes to a route preference as it would be caused by a configuration change. Specifically, this input reflects for a given route whether that is the most preferred route or not. Thus, Oasis can explore system behaviors for different preferences in the explorer's route selection process.

Snapshot We use Figure 3 as the guiding example of DiCE's operation in one round of exploration. Recall that the explorer initiates the exploration by triggering a checkpointing phase which results in the creation of a consistent snapshot [33] (Figure 3 step 1). In the current implementation, this is done by taking a checkpoint at the explorer and sending an UPDATE message annotated with a custom path attribute (step 1'). Enclosed in the attribute are the checkpoint number, the IP address of the explorer, a counter that is decreased at each hop to confine the exploration scope, and a weight used for termination detection (explained below).

A router that receives this annotated message interprets the custom attribute and, if this is the first time it sees the current checkpoint number, it takes a checkpoint (step 1"). As BIRD runs a single process, the procedure to take a checkpoint is simply implemented using the `fork` system call. This way of checkpointing allows us to create a large number of checkpoints with a small memory footprint. When a checkpoint is created from the BIRD process running in production, DiCE isolates the forked process from its parent by closing the open sockets[5] and marking them as shadow sockets. Also, the checkpoint is isolated from the FIB. Finally, DiCE opens a new socket to listen for incoming shadow connections on a different port from that used by the production instance of BIRD. With some implementation effort, the same techniques could be applied to other, more complex routing software[6].

The dissemination of the checkpoint message is achieved by announcing a route to a dedicated prefix so that every router eventually receives the checkpoint-annotated message. However, the explorer needs to be acknowledged when the checkpointing phase ends. For this purpose, we use a variation of the weight-throwing algorithm for termination detection in a distributed system [32]. Briefly, the explorer starts by sending the checkpointing message with an initial weight (e.g., 1). When a router receives the message, it keeps a part of the weight for itself (e.g., $weight \cdot 1/(\#neighbors + 1)$) and, while propagating the message, equally shares the remaining part of the weight among its neighbors. A router that does not propagate the message further keeps the received weight for itself. Meanwhile, every router reports its weight to the controller (step 1"). When the reported weights sum up to the initial weight the controller concludes termination of checkpointing and starts exploring by running the Oasis concolic engine.

Exploration Oasis collects constraints along the branches it encounters in the code. In our prototype, the constraints come from: (i) the BIRD C code that deals with UPDATE processing, (ii) the code for fuzzing path attributes, (iii) the code for injecting policy changes, and (iv) the BIRD configuration interpreter. Note that BGP router's behavior is a result of not only the code but also configuration. This is why the concolic engine records constraints for the interpreted configuration. Therefore, the explored execution paths are comprehensive of both code and configuration.

To perform path exploration, Oasis negates one constraint at a time and produces a new assignment of symbolic inputs (step 2) which are used to drive one *execution* of exploration. First, an isolated BIRD process is forked from the previously established shadow checkpoint (step 2'). Recall we term a process forked from a shadow checkpoint as cloned checkpoint. Before a message exchange between cloned checkpoints can take place, a connection is required to be setup. This is done by connecting[7] to the shadow checkpoint of the message destination (step 2") which creates a cloned checkpoint (step 2"'). Note that only one cloned checkpoint per node is created for each execution: the first connection is handled by the shadow checkpoint, any subsequent connection is managed by the cloned checkpoint itself. Then, messages are exchanged over these connections (step 3). When it receives the first message, each pro-

[5] Of course, this does not affect the production instance of BIRD.

[6] For example, Quagga [6] is structured as a set of processes, one per routing protocol. DiCE can be applied by controlling per-protocol shadow connections, and by isolating the processes from the FIB.

[7] This requires a 2-way handshake to avoid race conditions.

cess ignores the previously existent information about the route(s) contained in the message. This is to ensure that messages are propagated as they would in production, because otherwise the BGP selection would ignore the announcement.

The messages are extended to carry weight information so that the same termination detection algorithm described before is used to detect BGP convergence in the cloned snapshot (step 3'). However, routing may not converge if BGP is in an ill-state [24] within the snapshot. Therefore, during exploration, we use the method in [18] to prevent persistent BGP oscillation under arbitrary filtering (explained in Section 4.2). Lack of convergence due to system dynamics (session failures) are tolerated by shutting down the failed BGP session at the node at which a BGP error occurs.

When the controller detects that one execution terminates, it queries (step 4) all routers that participated in the exploration for properties that allow for fault detection as explained in Section 4. Then, exploration can progress with another execution based on the next input. When each execution terminates, the processes involved in the exploration can terminate as well and release the resources. The exploration then concludes when Oasis has covered the paths reachable by controlling the composite set of recorded constraints. At the end of the exploration all checkpoint processes are terminated as well.

Legacy routers and deployment To capture a system-wide snapshot, the annotated message has to propagate through the network and reach all routers within the exploration scope. This can be easily achieved by reserving a prefix for this purpose which is announced to trigger the checkpointing and withdrawn afterwards. This does not require any modification to BGP because custom route attributes are allowed in the protocol specifications, making it possible to pass-through legacy routers.

DiCE could be deployed incrementally on BGP routers. An ISP might configure a DiCE-enabled router to send exploration messages to spare equipment which can run in isolation and be monitored for observable errors (e.g., through system logs). In addition, an ISP could check for misconfigurations by deploying a single DiCE-enabled route server configured with the ISP policy and connected with similar machines or DiCE-enabled routers at the neighboring ISPs.

3.2 Integration with DNS

We now build upon the first case study and describe the important differences for applying DiCE with another crucial system for the Internet infrastructure: Domain Name System (DNS). DNS [34, 35] realizes a name resolution service for the Internet that maps host names to IP addresses. DNS is a distributed database composed of a large number of hierarchically organized, autonomously administered zones, each of which is a subspace of the global namespace that is authoritative for the names that share the same suffix with the zone's domain name. Each zone maintains a list of so called *Resource Records* (RRs) for the domains under the zone's authority. For example, the A records map names to IP addresses; the NS records identify authoritative name servers (ANSs). Typically, name resolution is carried out by a DNS resolver. In the basic form, given a name, the resolver queries one of the ANSs belonging to the name's domain.

Implementation Using the lessons learned during BGP integration, it took us less than a week to integrate DiCE with MaraDNS [2] 2.0.02, an open-source DNS server. This time includes all the efforts to compile the codebase, implement light-weight checkpointing, setup an experimental testbed, read the DNS code, decide what to make symbolic, and implement the code that drives exploration. Overall, we added 74 LoC to MaraDNS to integrate with the concolic engine, and another 78 LoC to enable symbolic inputs.

We leverage the fact that DNS servers process queries that do not change their state (we neglect the impact of caching). This simplifies the integration because the deployed nodes form a snapshot. We only instrument the recursive resolver, and integrate it with DiCE.

Inputs In DNS, local node actions do not result in state changes at remote nodes (we do not consider security exploits). In principle, therefore, a single node cannot be responsible for an event like system-wide session resets such as in BGP. However, node behavior is not only driven by code but also by configuration. In the case of DNS, errors lurking in the system configuration are an example of a cause of misbehavior that can be problematic for system reliability (e.g., the impossibility of resolving certain domains [37]). In the absence of state-changing operations, subtle misconfigurations manifest themselves as the result of specific interleaving of node actions. For DNS, that is the particular path (ordering of nodes) in which a DNS resolver attempts to resolve a domain name. Note that this path is also affected by failures of DNS servers or routing instabilities. We therefore recognize the importance of achieving systematic exploration of the system-wide execution paths during DNS resolution.

To drive the exploration, we change the way the resolver decides which ANS to query when it has multiple choices. We introduce a `get_server()` function that for each ANS list, maintains a subset of active servers. Each time the resolver needs a server from that list, the function selects one from the active subset. This way, DiCE tries all the possible server subsets and all the possible server combinations. In doing so, it mimics the remote server failures that could cause different

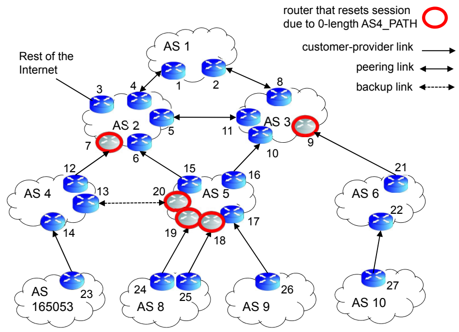

Figure 4: The topology we used in our BGP experiments with harmful global events.

local choices, as well as the different random choices in choosing a server.

4 Evaluation

In this section, we describe the way we evaluated our DiCE prototype, including a detailed description of the properties we use to detect faults.

4.1 Experimental setup

BGP setup In our experiments, we make use of a 48-core machine with 64 GB of RAM, running Linux 2.6.30. We install virtual interfaces on this machine, and use them to configure and run multiple BIRD router instances. We have previously quantified the memory overhead (37%) and performance impact (8% in a stress test; negligible in normal operation) of using a concolic engine for path exploration of a BGP router [12].

The 27-router topology we use is shown in Figure 4, and it corresponds to the topology used in [25]. We further annotate the topology with link latencies (30 ms among the routers in the same AS, 5 ms otherwise[8]) and link capacities (620 Mbps), and install it into the ModelNet [40] network emulator running FreeBSD 4.9 on a separate machine. This setup allows us to subject individual packets to link latencies and queuing delays that mimic Internet conditions.

The BIRD instances within an AS are connected in a full-mesh IBGP (internal BGP) topology, while the interdomain protocol is EBGP (external BGP). This setup does not use route reflectors. The network includes a mix of AS types (tier-1 ASes, tier-2 ASes and small stub ASes) interconnected with either customer-provider, peering or backup relationships as indicated by the arrows in Figure 4. We loaded 319,355 prefixes into the topology by replaying a BGP trace obtained from RouteViews (RIB dump plus 15-min updates trace from routeviews.eqix at April 1, 2010, 17:28 UTC). BGP policies were configured as in [25]: if a route is imported from a customer, it is exported to all neighbors; if a route is from a peer or provider, it is exported to customers only. This set of policies adheres to the Gao-Rexford conditions for stable routing [21] and therefore policy-induced oscillations are not possible. In our network, all ASes except AS 3 use customer route filtering to prevent their customers (including customers' customers and so on) from injecting advertisements for prefixes they do not own. We already showed in [12] how concolic execution can exercise a BGP router's behavior in a way that locally exposes potential origin misconfigurations (route leaks). Because of space limit, we do not present its extension to a system-wide property.

[8]This mimics the 60 ms RTT across the continental US an ISP in the US would experience.

DNS setup We run 5 nodes in the DNS testbed: one recursive DNS server and four authoritative ones. The recursive one uses the "deadwood" resolver from the MaraDNS software package that we integrated with DiCE. Other nodes run standard "maradns" servers, without any source changes. The nodes and their roles in our DNS experiments are as follows:

- Client: wants to determine the IP address of `foo.dd.aaa` and knows only Node1.
- Node1: recursive server. Knows Node2 and Node5.
- Node2: authoritative server for the `aaa` zone. Publishes the following NS records for `dd.aaa`:
  1. `ns.dd.aaa` (Node3)
  2. `ns2.dd.aaa` (Node4)
  3. `ns3.dd.aaa` (unreachable server)
  4. `ns.dd.bbb` (glueless record)
- Node3: authoritative server for the dd.aaa zone. Has the IP address of `foo.dd.aaa`
- Node4: redundant authoritative server for the `dd.aaa zone.` Has the IP address of `foo.dd.aaa`
- Node5: authoritative server for the dd.bbb zone. Publishes one NS record:
  1. `ns.dd.aaa` (glueless record)

The glueless record is a record that spans different domains without providing "glue" (similar to the A record) in the form of (name, IP address) that could be used to reach a name server.

4.2 Testing configuration changes for possible policy conflicts in BGP

BGP has evolved over time to allow each ISP to independently decide on the set of routes that will be announced to each neighboring AS using a set of *policies*. These policies capture business decisions and are often private.

However, conflicting policies can cause undesired persistent routing oscillations [41] which negatively impact end-to-end performance of the Internet. We use the case of checking for the effect of a change in the local policy to showcase DiCE's baseline "what-if" testing ability.

Note that policy conflicts are due to a design flaw in BGP and only by changing the protocol itself the problem can be definitely addressed. In fact, a recent approach for resolving policy conflicts advocates changing BGP [18]. Here we describe our approach for detecting policy conflicts in a way that does not require protocol modifications for the production traffic. Detecting conflicts is important, for example, to avoid oscillations due to the use of backup routes that were not checked.

Griffin *et al.* [24] formally analyzed the Stable Paths Problem (SPP) which is an abstraction of the problem that BGP intends to solve in a distributed fashion. They attribute policy-inflicted oscillations to a circular set of conflicting rankings between various nodes which form a *dispute wheel*. They show that the absence of a dispute wheel ensures the existence of a unique, stable and robust SPP solution. The work in [21] suggests that ISPs observe a set of best practices to avoid the dispute wheel, but these are unnecessarily restrictive and difficult to check in a distributed fashion. In order to prevent policy-induced oscillations from occurring, Ee *et al.* [18] augment BGP with the *global precedence value*, which is carried as an additional attribute in route announcements and is used as discriminator in the BGP decision process with higher importance than local policy preferences.

The key idea for detecting policy conflicts is to leverage the global precedence value to detect a policy conflict in a cloned snapshot while exploring a large number of possible behaviors. Specifically, we implemented the global precedence value [18] and we used it to ensure that the routing protocol converges[9]. If during convergence a route announcement contains a non-zero global precedence value, it means that the snapshot contains a dispute wheel, and therefore a policy conflict exists.

For our experiments with policy conflicts, we construct a 5-node topology presented in [24], known as GOOD GADGET. This topology presents a Stable Paths solution. However, a single switch in the ranking of paths (policy change) transforms it into a BAD GADGET topology that has a dispute wheel.

DiCE successfully detected the possibility of a policy conflict in this topology by systematically exploring the consequences of one change at a time in the route preference assignments. Overall, there were 75 iterations that took 39 seconds to explore.

Benefits of using DiCE The ability to detect policy conflicts (and the resulting routing oscillations) before they happen on the Internet is beneficial for allowing the freedom of policy decisions in order to accommodate the complex objectives that govern route choices for ISPs.

[9]Outside of exploration, the usual BGP decision process runs with the node's policy and unmodified protocol messages.

4.3 Testing under failures and random choices: cyclic dependencies in DNS

Pappas *et al.,* [37] identified three important classes of configuration errors in DNS, and the one they report being particularly difficult to identify is the cyclic zone dependency. This error involves configuration of multiple servers, and importantly, cannot be detected by inspecting the individual server configurations. The error is also insidious in that in the normal course of operation the system functions correctly notwithstanding possible delays. However, a particular failure pattern of authoritative servers can cause resolution to take place via other servers in a domain's configuration, which then leads to a cyclic dependency involving two or more servers that cannot be resolved. This error results in domain names becoming unresolvable and ultimately unavailable.

Using DiCE's second testing mode (the *choice* function) we successfully detected a cyclic dependency in our testbed after executing 502 explorations that took 532 seconds. The cyclic dependency is locally detected at the resolver. In this case, the `get_server()` function chooses an execution path in which Node3 and Node4 are not queried (e.g., because they are considered to have failed). Query resolution proceeds via Node5, but unfortunately Node5 redirects the query back to Node2, where the same decision to avoid Node3 and Node4 is made again and again. The cyclic dependency would manifest itself in the production system if Node3 and Node4 were both to fail. This experiment demonstrates how DiCE systematically explores system behavior under possible failures in a case when it is not possible, or difficult, to cause these failures to occur in the production system.

4.4 Testing BGP router code for possibly causing harmful global events

To showcase DiCE's ability to systematically use the code itself to test it, we install a property that checks for the presence of a particular event at a global scale, akin to emergent behavior. An example that affected BGP is the session reset problem [4]. The core of the problem is the fact that the affected routers had difficulty handling an update in which the AS4_PATH attribute had zero (0) length. The router receiving such an update would not crash, but it would reset the session with the sender of the message. In strict isolation, this seemingly valid handling of a semantically confusing message would not have a far-reaching impact. Unfortunately, a large fraction of routers were not affected by this programming error and were effectively multicasting the session reset signal. Each session reset is followed by a new session establishment that triggers a full routing table download and route processing that is CPU intensive. Moreover, the routing updates containing the confusing AS4_PATH attributes would then be redelivered to the affected routers, causing another round of session resets, and so on. As a result, the peak update traffic in the Internet was increased by more than a factor of 10 (1000%) [4]. Operators had to manually install packet filters to prevent this fault from recurring.

To enable DiCE to detect this type of a fault, we install the property which signals a possible fault whenever a router resets the session (seen as an increase in the BGP error count) in response to an exploratory message. To trigger this fault in our testbed, we replicate the previously described scenario [4] to the extent possible as we do not have access to the Cisco IOS code. We introduce code into BIRD (already 4-byte AS-compliant) that resets the session when a zero-length AS4_PATH attributes arrives. The routers that are configured to be affected by the confusing attributes are marked with a circle in Figure 4. Note that AS 165053 is using a 4-byte AS number. The update containing the zero-length AS4_PATH was generated using a fuzzed message.

Detection results We instructed the routers in our testbed to perform exploration after finishing loading the 319,355 prefixes. The routers use an actual message to record the constraints. Different routers explore a different number of iterations on the same code because of the: 1) different messages that end up being used to initially record the constraints, and 2) different C code that gets instantiated by BIRD to enforce the filtering commands. However, Oasis reported that it had explored all paths at each router. The maximum number of explorations was 2002, minimum was 7 while the average number was 763 and std. dev. of 586. Routers 23 and 27 explored with 7 inputs because they only accept a default route and have no filtering enabled. The maximum observed time was 670 s to explore the total of 1156 explorations. The average time to explore was 243 s with a std. dev. of 204 s. We also measured the exploration time without accounting for network delays by repeating the experiments with the same initial messages, but without Modelnet. Overall, the times are smaller with an average of 155 s and std. dev. of 113 s.

Given the timescale over which the Internet incidents occurred due the erroneous configuration files and software (likely to be in place for weeks if not months), the time DiCE took to detect these faults is negligible.

Benefits of using DiCE Armed with a property that checks the BGP error count (which can be increased due to a variety of different reasons), DiCE produces a list of possible actions that can cause the systemwide error count to go over the threshold. Each ISP would benefit from this advance warning and could take a number of actions, including: 1) notifying the router vendor and requesting a patch, 2) manually fixing the code if the source is available, and 3) automatically installing filters to filter out the offending message (if the action is caused by message). Without DiCE, this kind of repair was undertaken only after the reset incident took place across the Internet and was diagnosed after several hours [4].

5 Related Work

Model checking. CrystalBall [44], and MODIST [45] represent the state-of-the-art in model checking distributed system implementations. CrystalBall [44] proactively predicts inconsistencies that can occur in a running distributed system due to unknown programming errors, and effectively prevents them. It works for systems implemented in the Mace [29] framework. CrystalBall nodes periodically collect a consistent snapshot of system state, and locally run a model checking heuristic on the set of state machines instantiated from the snapshot. MODIST [45] is capable of model checking unmodified distributed systems. One could use MODIST to orchestrate state space exploration across a cluster of machines in an isolated (non-deployed) scenario.

DiCE goes beyond these approaches in several important aspects because it: 1) can uncover faults due to inputs that are different than those fed by the model checking harness, 2) deals with the issues arising from federation (need for privacy, inability to retrieve state and configuration), and 3) incorporates the intrinsic heterogeneity of the system (nodes behave differently either due to different implementations, or configurations).

Symbolic execution. Symbolic [11] and concolic execution [10, 17, 23] are effective in discovering bugs in single-machine code by trying to achieve complete coverage of possible code paths. However, they are limited in their ability to reach faulty states as they cannot handle large inputs in long-running systems and realistic configuration (e.g., Klee [11] works with only several bytes of input to achieve good path coverage). In addition, these engines are only successful in searching for violations of local assertions (e.g., memory violations). Thus, without the spatial awareness achieved by DiCE, it is not possible to judge the system-wide impact of node actions.

KleeNet [39] builds a test harness that accommodates messaging and fault injection on top of Klee [11]. To search for bugs, KleeNet arranges for path exploration among the set of TinyOs nodes running in isolation on one machine, prior to deployment. This approach is thus similar in spirit to model checking that starts from initial state, along with the shortcomings in dealing with long-running, federated, and heterogeneous systems.

Relative to symbolic execution approaches, DiCE: 1) explores system behavior starting from live state and configuration which is crucial for overcoming the path explosion problem in long-running systems, 2) provides a way to control inputs to a single node that explore relevant federated system states, 3) adapts to the federated environments by providing a narrow interface for sharing the information of local checks, and 4) accommodates system heterogeneity by allowing each administrative domain to separately integrate DiCE.

There have been proposals for performing path exploration at selected points in time on a single-machine [15]. Extending this work to handle networked systems require developer's involvement in eliminating inconsistent or impossible states. DiCE provides this information naturally, using the system itself. We have highlighted challenges of fault detection in federated, heterogeneous systems [13]. DiCE is a system that addresses these challenges. In our short paper [12], we present a preliminary DiCE design, and detail our experiences in integrating a BGP router with a concolic execution engine. This paper goes beyond our short paper in that it: 1) shows how to carefully checkpoint the system to allow the concolic engine to extend its reach across the network, 2) shows how to control the inputs to the concolic engine to enable it to reach relevant federated system states, 3) includes a privacy-preserving scheme for checking properties, 4) presents a disjoint set of experimental results involving BGP, and 5) details our experience of integrating DiCE with DNS, along with the accompanying experimental results.

MAX [31] uses symbolic execution to find protocol manipulation attack which can be harmful to a network participant. In contrast, DiCE determines the impact of a node's actions on the remainder of the system.

Castro *et al.,* [14] use symbolic execution and constraint solving to randomize inputs that cause application crashes in an effort to improve privacy of bug reports. DiCE applies a similar idea, but to a different domain and for new functionality (fault detection).

Application-specific fault detection and prevention Tools that look for faults in the set of router configurations using static analysis [19] can be quite effective, but cannot check live state spanning multiple nodes, and their configuration (which can differ from the statically checked files). The work on NetReview [25] posits that is difficult to prevent all classes of faults, and argues that the best we can do for the general case is to detect faults in BGP after they occur. DiCE goes one step further in that it detects important classes of faults *before* they manifest themselves.

Alimi *et al.* advocate use of shadow configuration as a network management primitive [7]. This approach installs an alternative configuration within a single ISP's routers, and checks its validity. DiCE's shadow snapshot bears resemblance to this "shadow config" primitive, but: 1) is lightweight (works on existing router processes), 2) is automatically created without operator involvement post-deployment, 3) can span multiple ISPs, and 4) serves to detect faults due to unanticipated inputs [4], bugs, or operator mistakes before they are tried out or put into effect. DiCE can benefit from virtual network substrates (e.g., [9]) to simplify shadow and cloned snapshot creation.

Bug-Tolerant Routers [27] run multiple router implementations in parallel using virtualization, and mask faults by voting and changing the environment of the router processes. In contrast, DiCE possesses the necessary spatial awareness to detect semantic faults that span multiple routers, systematically explores node behavior, and does not require multiple router implementations.

Proposals exist for dealing with specific BGP faults, e.g., oscillations [18]. We argue that it is better to detect a large class of faults before they occur. It is then possible to devise general or specific solutions for preventing them. Our work is complementary to the numerous security extensions to BGP (e.g., [28]) which prevent certain classes of attacks. However, these works cannot guard against programming errors or policy conflicts.

Pappas *et al.,* [36] have proposed and implemented a third-party tool that periodically downloads DNS resource records belonging to a large number of domains, and checks them for cyclic dependencies (as well as other misconfigurations). We demonstrate DiCE's ability to automatically accomplish a similar task within DNS itself, where there is a clear incentive for the DNS administrators to identify and eliminate cyclic dependencies.

6 Conclusions

We presented the design and implementation of DiCE, a system for detecting faults in the long-running, heterogeneous, and federated distributed systems. DiCE enables system operators to first specify properties that capture the desired system behavior. DiCE then: 1) automatically and systematically explores a large number of relevant executions, 2) checks their system-wide impact in isolation while respecting privacy among different administrative entities, and 3) reports safety property violations. We integrated DiCE with two systems crucial for Internet's operation: BGP and DNS. This paper describes the lessons we learned on how to control inputs fed to nodes in order to explore relevant system-wide state. Our evaluation demonstrates DiCE's effectiveness and ease of integration with existing software written in C. Specifically, our prototype quickly detects faults that can occur due to policy conflicts, misconfigurations, and programming faults.

7 Acknowledgments

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. Dejan Novaković is supported by the Swiss NSF (grant FNS 200021-125140).

References

[1] How To Build A Cybernuke. `http://www.renesys.com/blog/2010/04/how-to-build-a-cybernuke.shtml`.

[2] MaraDNS. `http://www.maradns.org`.

[3] Research experiment disrupts Internet, for some. `http://www.computerworld.com/s/article/9182558/Research_experiment_disrupts_Internet_for_some`.

[4] Staring Into The Gorge: Router Exploits. `http://www.renesys.com/blog/2009/08/staring-into-the-gorge.shtml`.

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

[6] The Quagga Routing Suite. `http://www.quagga.net`.

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

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

[9] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *SIGCOMM*, 2006.

[10] 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.

[11] 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.

[12] M. Canini, V. Jovanović, D. Venzano, B. Spasojević, O. Crameri, and D. Kostić. Toward Online Testing of Federated and Heterogeneous Distributed Systems. In *USENIX*, 2011.

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

[14] M. Castro, M. Costa, and J.-P. Martin. Better Bug Reporting With Better Privacy. In *ASPLOS*, 2008.

[15] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *ASPLOS*, 2011.

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

[17] 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, EPFL, 2009.

[18] C. T. Ee, V. Ramachandran, B.-G. Chun, K. Lakshminarayanan, and S. Shenker. Resolving Inter-Domain Policy Disputes. In *SIGCOMM*, 2007.

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

[20] P.-A. Fouque, G. Poupard, and J. Stern. Sharing Decryption in the Context of Voting or Lotteries. In *FC*, 2001.

[21] L. Gao and J. Rexford. Stable Internet Routing Without Global Coordination. *IEEE/ACM Trans. Netw.*, 9(6):681–692, 2001.

[22] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *PLDI*, 2008.

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

[24] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, 2002.

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

[26] A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *OPODIS*, 2009.

[27] E. Keller, M. Yu, M. Caesar, and J. Rexford. Virtually Eliminating Router Bugs. In *CoNEXT*, 2009.

[28] S. Kent, C. Lynn, and K. Seo. Design and Analysis of the Secure Border Gateway Protocol (S-BGP). *DARPA Information Survivability Conference and Exposition,*, 2000.

[29] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[30] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[31] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding protocol manipulation attacks. In *SIGCOMM*, 2011.

[32] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[33] D. Manivannan and M. Singhal. Asynchronous Recovery Without Using Vector Timestamps. *J. Parallel Distrib. Comput.*, 62(12):1695–1728, 2002.

[34] P. Mockapetris. Domain Names - Concepts and Facilities IETF RFC 1034. 1987.

[35] P. Mockapetris. Domain Names - Implementation and Specification IETF RFC 1035. 1987.

[36] V. Pappas, P. Fältström, D. Massey, and L. Zhang. Distributed DNS Troubleshooting. In *NetT*, 2004.

[37] V. Pappas, D. Wessels, D. Massey, S. Lu, A. Terzis, and L. Zhang. Impact of Configuration Errors on DNS Robustness. *IEEE J.Sel. A. Commun.*, 27:275–290, 2009.

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

[39] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *IPSN*, 2010.

[40] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*, 2002.

[41] K. Varadhan, R. Govindan, and D. Estrin. Persistent Route Oscillations in Inter-Domain Routing. *Computer Networks*, 1996.

[42] Q. Vohra and E. Chen. BGP Support for Four-octet AS Number Space IETF RFC 4893. 2007.

[43] M. Yabandeh, A. Anand, M. Canini, and D. Kostić. Almost-Invariants: From Bugs in Distributed Systems to Invariants. In *SRDS*, 2011.

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

[45] 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.

[46] J. Yang, C. Sar, and D. Engler. EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors. In *OSDI*, 2006.

A Anonymous property checks

We now describe a proof-of-concept scheme that uses a protocol for Secure Multy-Party Computation (SMPC) based on the threshold variant of Paillier's cryptosystem in [20]. Let D denote the set of participating domains, N_j be the nodes of domain $j \in D$. We assume there exists an out-of-band mechanism for disseminating a shared public key PK and a list of private keys $SK_1, \cdots, SK_{\|D\|}$. Each domain j sends the cyphertext $E_{PK}(\sum_{i \in N_j}[f(\Theta_i)])$ to all other domains. Next, each domain leverages the homomorphic property of the cryptosystem [20] to compute $c = E_{PK}(\sum_{i \in N}[f(\Theta_i)]) = \prod_{j \in D} E_{PK}(\sum_{i \in N_j}[f(\Theta_i)])$. The decryption of c is shared across all domains. Specifically, each domain j runs a decryption algorithm using SK_j that produces a decryption share c_j and sends it to other domains. Finally, each domain inputs $c_j, \forall j$ to a combiner algorithm that outputs $\sum_{i \in N}[f(\Theta_i)]$. Comparing this value with the threshold th gives the global check.

We implemented the above protocol in Java using "thep"[10] as a starting point. We ran a micro-benchmark to evaluate its performance using the same experimental setup used for BGP. With respect to the experimental topology (Figure 4), we only used one node per AS because we assume that nodes inside the same domain would trust each other. In summary, we obtained that the times needed for one secure computation are 417 ms and 1979 ms for running without and with ModelNet, respectively. This result leads us to a conclusion that a version of DiCE prototype supporting SMPC should implement secure computations as a pipeline running in parallel to the system exploration process, and batch multiple computations together.

[10] A Java implementation of Paillier's cryptosystem `http://code.google.com/p/thep/`.