Almost-Invariants: From Bugs in Distributed Systems to Invariants
EPFL Technical Report NSL-REPORT-2009-007

Maysam Yabandeh, Abhishek Anand, Marco Canini, and Dejan Kostić
School of Computer and Communication Sciences, EPFL, Switzerland
firstname.lastname@epfl.ch

Abstract

It is notoriously hard to develop dependable distributed systems. This is partly due to the difficulties in foreseeing various corner cases and failure scenarios while implementing a system that will be deployed over an asynchronous network. In contrast, reasoning about the desired distributed system behavior and the corresponding invariants is easier than reasoning about the code itself. Further, the invariants can be used for testing, theorem proving, and runtime enforcement.

In this paper, we propose an approach to observe the system behavior and automatically infer invariants which reveal implementation bugs. Using our tool, Avenger, we automatically generate a large number of potentially relevant properties, check them within the time and spatial domains using traces of system executions, and filter out all but a few properties before reporting them to the developer. Our key insight in filtering is that a good candidate for an invariant is the one that holds in all but a few cases, i.e., an “almost-invariant”. Our experimental results with the BGP, RandTree, and Chord implementations demonstrate Avenger’s ability to identify the almost-invariants that lead the developer to programming errors.

1 Introduction

The Internet has changed the way we communicate, obtain content, and do business. It is therefore a primary example of a distributed system that has to be highly dependable. Implementing and deploying such systems is difficult for a number of reasons, including the sheer system size, concurrency issues, the number of unforeseen events, and the difficulty in structuring protocols that run over asynchronous networks.

The approaches for making distributed systems more reliable have evolved from debugging using log inspection to more complex techniques such as property checking [10, 27, 37, 44], model checking [22, 44], and enforcing the invariants at runtime [44]. The latter approaches require the developer to specify the desired system behavior in the form of invariants that are supposed to hold at all times. Although reasoning about invariants is arguably easier than reasoning about the source code itself, the developer is still expected to provide the invariants. This task becomes more and more difficult as the system gets larger and more complicated, and as the developer starts dealing with various corner cases. For example, in distributed systems in which various network failures can occur, reasoning about an invariant that holds under all failure conditions can be difficult. While some distributed systems have been written with invariants in mind [25], many have not. Although others have shown that it is possible to discover invariants [13, 14] and even specifications [2, 11] of single-machine code, invariant inference is still an open and important challenge in distributed system implementations.

We observe that, due to the difficulty of dealing with various issues in the deployment environment, there exist a potentially large number of important distributed system invariants that only get violated under certain conditions (and would be discarded if the existing tools for single-machine code were to be applied). We refer to such properties as “almost-invariants”. Most often, an almost-invariant gets violated due to a rare manifestation of a programming error (bug) that needs to be fixed.

In this paper, we introduce a new tool, Avenger, for inferring almost-invariants in distributed system imple-

---

1 A property expresses a relation between some variables (including iterator variables) which can be evaluated as true or false. Throughout this paper, we use the term invariant to refer to a property that is never observed to be violated. Some related work refers to invariants as safety properties. We introduce the term almost-invariant for a property which holds in all but a few cases in which an inconsistency violates it.

---

*current address: abhishek.anand.iitg@gmail.com, IIT, Guwahati, India, (work done during an internship at EPFL)*
Avenger improves reliability in distributed systems from two aspects: testing and resilience:

- Our tool aids in testing by exposing the behavior of the distributed system via the inferred almost-invariants. In addition, it can help reduce the time spent in development and code maintenance.
- Our tool aids resilience by generating relevant almost-invariants that, once the underlying programming errors are fixed, can be used in static analysis, model checking, or enforcement at runtime [44].

1.1 Contributions

We make the following contributions:

- We introduce a new automatic testing technique for distributed system implementations based on identifying the almost-invariants in execution traces. The inferred almost-invariants are those that are likely to point to programming errors.
- We demonstrate that our tool helped identify new programming errors in the code that was previously debugged using live executions, model checking, and a combination of both. In addition, the tool exposed a problem in handling 4-byte AS numbers in the BGP protocol implemented in C++ within the XORP open-source router [17].
- Finally, we show that our tool can be used to gain insight into the functioning of a distributed system by revealing the transient loops created by the BGP protocol implemented in XORP.

1.2 Example

Next, we use an example to discuss: 1) the difficulty in manually defining distributed system invariants, and 2) the importance of automatic inference of almost-invariants in pursuit of programming errors.

In this example, we highlight an almost-invariant inferred by Avenger that led to the discovery of a previously unknown programming error in the Mace [21] implementation of the random overlay tree (RandTree). Overlay trees have successfully been used in a variety of scenarios, including point-to-multipoint data dissemination [7] and data collection/monitoring environments [20]. In particular, the RandTree protocol maintains a degree-bounded tree among a set of participating nodes. The state of each node consists of the root, the parent, the list of children, and the list of siblings. Although the per-node state is seemingly simple, maintaining a consistent tree structure in the face of node failures, network outages, and delayed messages is a difficult task.

```plaintext
1 proc NewRequestHandler(JoinReqMsg, children)
2 if (children.size() < MAX_CHILD_SIZE)
3     children.add(msg.sender);
4 else
5     ...
```

Figure 1: Example of a buggy program

![Figure 1: Example of a buggy program](image1)

Figure 1 presents a RandTree code snippet handling the Join message sent by another node that is attempting to join the tree. Since the tree is degree-bounded, a RandTree node accepts the requesting node as a child if it has room, or otherwise forwards the request to one of its children. The developer has correctly taken into consideration the case in which the children list is full (Line 2). However, there exists another rare case that the developer has not thought of: due to failures and non-determinism typical of distributed systems, it is possible that the join request which was originally sent by the parent of node \( N \) is mistakenly forwarded to \( N \). By accepting the join request from its parent, \( N \) causes an inconsistency in the system. As shown in Figure 2, the parent of the node has become one of its children. The detailed scenario which leads to this inconsistency is presented in Section 3.

This kind of mistake can be detected (e.g., by model checking) if the appropriate invariant is in place: \( \text{parent} \notin \text{children} \). Although this invariant seems obvious in retrospect, it was neglected by the MACEDON [35] and Mace [21] developers, as well as the CrystalBall [44] researchers that defined some additional invariants for RandTree during their testing efforts.

1.3 Avenger overview

The almost-invariants we wish to identify can refer strictly to a local node state, or to state belonging to multiple nodes. An example of a local invariant is the one we described in Section 1.2: the identity of the parent node must not be present in the children set of a tree node. A classic example of a multiple-node invariant is the one in Paxos [25]: if a value is chosen in the instance of the protocol, no two nodes should choose different values.
Our approach for the automatic inference of distributed system almost-invariants consists of these steps:

1. defining a grammar for distributed properties,
2. automatically generating a large pool of possible properties,
3. subjecting the possible properties to scrutiny by checking their validity over state traces, and
4. carefully choosing a small set of almost-invariants to be reported to the developer at the end of the inference process.

Our key insight is in realizing that the complexity of distributed environment (with its failure modes and the underlying asynchronous network) makes it hard to reach bullet-proof distributed system implementations. The last 1% bugs manifest very rarely which makes them even harder to detect. For example, a bug can be introduced when the developer fails to take a particular sequence of inputs, external events, or failures (corner case) into consideration when writing the program, and as a consequence the system does not behave as the developer intended. It is also possible that some emergent behavior [28] could be the cause for less than perfect system operation.

Our approach leverages the rarity of the manifestation of inconsistencies and emergent behaviors in complex distributed systems and looks for the almost-invariants across long and varied distributed system executions. The potential properties are evaluated for validity both over time and spatially, e.g., across the system state. After generating and checking a large number of properties, Avenger ultimately reports a handful number of almost-invariants that hold in most of (but not all) the cases. Even though this approach might also identify some transient properties, manually checking them can reveal interesting characteristics of the system dynamics.

**Discussion.** It might appear to be counterintuitive that we only report properties that get violated at least once. However, the reason behind this decision is that we identify properties that are very likely to be real invariants because they hold the vast majority of the time. This design decision does however place more responsibility on the construction of the workloads and faultloads that are used to evaluate the system in question.

By concentrating on almost-invariants that hold in most, but not all, the cases, Avenger is most likely to be useful when the code has somewhat matured. In the early phases of development, the distributed system behavior can be substantially different from its target, making it unlikely that the potential invariants will hold in the majority of the encountered states. This reduces the effectiveness of our tool in such cases. Given an increasing number of options for getting the early bugs out of the system, we decided to concentrate on the arguably most difficult part of helping the developer (and the system) get rid of the last 1% of the bugs.

Others have developed techniques for automatically inferring invariants of single-machine code. Daikon [13, 14] and related approaches generate a large number of possible properties and then proceed to keep those that hold all the time. Thus, they make an important assumption, namely that the code or specification that is used to generate the properties is correct. The tools designed for single-machine code cannot generate and deal with distributed invariants that span state across multiple nodes. Moreover, these tools cannot deal with invariants in the spatial domain (where only some machines in the system are violating a property at a given point in time).

Approaches exist that look for a violation of patterns of use [12, 18] in single-machine code. The inconsistencies in distributed systems can result from spatial and temporal interleavings of actions made by multiple nodes and it is not likely that checking deviation from common patterns of use in single-machine code can cover the entire space of distributed system properties.

No tool can guarantee to find all bugs, which means that Avenger is a heuristic. As with any heuristic, false positives are an inseparable part of the method. Moreover, Avenger targets a very particular type of bugs that manifest very rarely, which in turn increases the risk of false positives (e.g., [3] contains a discussion on the necessity of false positives in anomaly-finding tools). To avoid requiring excessive effort from the developer to check the reported properties, we limit the number of reported almost-invariant to a small number (10 in our experiments). Despite the false positives, Avenger is helpful as a tool for making distributed systems more robust.

Our experience presented in Section 3 demonstrates that Avenger helped to identify new bugs in multiple systems, which were not detected by existing techniques.

### 1.4 Challenges

Automatic property inference for distributed systems presents a number of challenges: 1) In order to arrive at a complete set of properties, the tool needs to have a good starting point in the form of a large number of potential properties. 2) Checking the validity of potential properties has to be done quickly and efficiently; 3) An issue related to quickly checking a large number of properties is the ability to filter out some useless properties; 4) An important challenge for an automatic tool is to obtain a large number of distributed system traces that can expose the operation of the distributed system under a variety of scenarios; 5) The set of properties that will be ultimately reported to the programmer, or passed further for automatic verification or enforcement, should be
2 Avenger

Figure 3 shows that a programmer uses Avenger by first providing input to it by describing the relevant node state specification, non-standard predicates and iterators, as well as configuration parameters that control the type and the number of potential properties that will be generated. Avenger is then driven by the state traces. Entries in the trace are system snapshots recorded over time, while the snapshot is a consistent set of nodes’ states. Finally, the node state is represented as an object that contains relevant state variables.

Figure 4 provides a high-level overview of Avenger by depicting the key components and the type of data flowing between them. The Property Manager uses a system specification to produce a pool of potential properties. The State Iterator steps through the system states and feeds the consistent, global system state to the Property Checker. The property checker checks the properties against each state, updates their statistics and invokes the Property Filter at coarse timescales. Using the property statistics, the filter attempts to reduce the size of the property pool in an effort to speed up overall execution time. At the end of the state iteration, the Property Reporter module selects a handful of the almost-invariants in the property pool and reports them to the developer for further inspection. We now describe each of the Avenger modules in more detail, along with the key challenges we faced in designing them.

2.1 Property Manager

The Property Manager uses a developer-provided specification of the system to generate the set of potential properties. The main challenge here is to support the generation of all the relevant properties that can hold in the distributed system under scrutiny. In the following sections, we describe our approach to meeting this challenge.

2.1.1 Properties

Avenger’s properties are expressed using a grammar similar to first-order logic. The grammar is sufficiently expressive to derive complex system properties which, for example, iterate over container data types while tying together states from different nodes, as later shown in Section 3. Fundamentally, each property is a disjunction of predicates or their negation. As in first-order logic, a predicate represents a relation between variables and can be evaluated as true or false. For example, \( n_1.hashId \in n_0.successor_list \), is a predicate which is evaluated to true if the node \( n_1 \)’s hash ID is a member of the node \( n_0 \)’s successor list. A predicate consists of an operator, e.g., the membership operator (\( \in \)), and variables, such as the hash ID and the list of successors in the above example. The universal quantifier (\( \forall \)) indicates the iteration over a set of items. At the top level, a property iterates over the set of nodes in the distributed system, potentially with multiple nested iterations. Further, a property may iterate over arbitrary container-type variables, such as arrays, vectors, lists, sets, and maps.

Formally, the general syntax of a property is presented in the following formula:

\[
Pr_i : \forall n_0 \in \text{nodes} \ldots \forall n_m \in \text{nodes} : \\
\forall s_0 \in n_{q_0}.v_0 \ldots \forall s_k \in n_{q_k}.v_k : \\
il_0 \lor l_1 \ldots \lor l_t
\]

which selects \( m + 1 \) nodes, \( n_i (0 \leq i \leq m) \), and \( k + 1 \) variables, \( v_i (0 \leq i \leq k) \), as the universes of quantifications, and calculates the disjunction of \( t + 1 \) literals, \( l_i \), \((0 \leq i \leq t)\). Each literal represents a predicate \( P \) or its negation \( \neg P \). In the following, we explain each property component in detail and explain the reasons for adopting this grammar.
The complete set of properties that can be generated is prohibitively large. For example, n binary literals can be combined to form \(2^n\) distinct properties. In an effort to reduce computational cost, we do not consider conjunction of predicates. This decision has limited impact on the Avenger's ability to identify important almost-invariants. According to the distributive law, the disjunction operator distributes over conjunction of literals:

\[ P_0 \lor (P_1 \land P_2) = (P_0 \lor P_1) \land (P_0 \lor P_2) \]

Hence, every first-order logic formula \(F\) can be converted to a conjunction of some other first-order logic formulas \(F_i\), where \(F_i\) does not include any conjunction. On the other hand, after obtaining the holding rate of properties, Avenger selects the top almost-invariants to report to the user (more details are in Section 2.5). Because the rank of each conjunction-free formula, \(F_i\), would be higher than the whole formula, \(F\), formula \(F\) will not be selected in the process anyway. If the programmer so desires, she can include a conjunction in a custom predicate implementation.

2.1.2 Developer’s Input

Avenger generates properties referring to the variables that represent the state of system nodes. Some development frameworks make the state of a node explicit [21], which in turn makes it possible for Avenger to automatically extract the relevant variables. In the general case of arbitrary C++ code, Avenger uses a developer-provided system specification to become aware of the relevant pieces of state. As detailed below, a system specification consists of two parts: Variables and Predicates.

Variables The state-related input to Avenger is the Variables.h file which contains the specification of the structure of a system node state. Avenger provides a simple syntax to express the state structure: the user simply needs to write the identifiers of the state variables and their corresponding data types. An identifier is basically a variable name. Global functions or public class methods with empty parameter lists can also be used as identifiers provided that calling these functions will not change the state. In these cases the identifier is the function name followed by the symbol (). A reference to a basic data type is expressed with the //VARIABLE keyword while a container data type is referenced with the //CONTAINER keyword. An example is shown in the following code excerpt:

```cpp
//VARIABLE mace::MaceKey meIPv44;
//VARIABLE mace::MaceKey /*1*/ myhash;
```

Avenger supports all data types in C++ ranging from built-in data types such as integers, floats, and booleans, to complex user-defined classes and structs.

Further, the syntax allows a few annotations that appears in between the */ and */ symbols. One of such annotations allows to assign a different numeric sub type to a certain data type (by default the sub type is 0 and can be omitted). This gives the user a chance to provide simple annotations in the case of applications where a single data type is used for semantically different purposes (in such a case, comparison between these variables is meaningless and can only slow down the whole inference process). For example, the integer type is sometimes (mis)used to store both IPv4 addresses and hashes/checksums. In the example above, the type of variable myhash is distinguished from the type of variable meIPv4 using the sub type 1.

A second kind of annotation, */iterator*/, informs Avenger whether a certain data type is iterable. Automatically, Avenger supports standard container data types such as STL array, vector, set, map, etc., however, extending Avenger to new iterable data types is straightforward and just requires an implementation of the iterator interface provided by Avenger and a simple annotation in Variables.h.

Predicates Similarly to other invariant inference tools [13], Avenger provides predicates that correspond to standard operators (equality, membership, etc.). In addition, Avenger is extensible by allowing the user to specify additional predicates of interest. The user does this by populating the Predicates.h file with predicate templates and providing the corresponding logic in Predicates.cc.

In general, a predicate combines the variables using operators, e.g., parent ∈ children. A predicate template expresses an operation over particular data types. For example, the following predicate template:

```cpp
bool contains(const NodeSet &,
              const mace::MaceKey &);
```

defines the membership test of the type MaceKey in the set type NodeSet that ends up being important for the discovered bug in our example (Section 1.2). Having this template, Avenger generates a set of predicates using the list of all variables of types MaceKey and NodeSet (recall
that the list of variables is read from Variables.h). For example, the predicate parent ∈ children is generated by this template, since parent is of type MaceKey and children is of type NodeSet. Later during the evaluation of properties, Avenger calls the implementation of the predicate template in the Predicates.cc file.

For example, the corresponding logic for the aforementioned predicate template is:

```cpp
bool predicateContains(
    const NodeSet &nodeSet,
    const mace::MaceKey &node) {
    return nodeSet.containsKey(node);
}
```

Initially, the predicate declaration and definition files are pre-populated with the common operators. The extensible design also allows the developer to specify additional predicate templates.

### 2.1.3 Property Generation

The following presents a high-level description of how Avenger generates the initial properties. First, Avenger parses the list of variables and recursively expands the container data types to produce a tree of variables and iterators, each of which is annotated with its type and sub type. The role of this tree structure is explained later. Avenger then reads in the predicate templates and generates a first set of predicates by instantiating predicate templates with all the possible combinations of variables with the appropriate data types. These generated literals make use of a single system node reference, that is, any combination of variables is scoped within the state of a single node (e.g., contains(n.successor_list, n.hashId)). During a second iteration, Avenger increases the number of system node references and instantiates a second set of literals with all the possible and appropriate combinations of variables whose scope is now enlarged to address the state of two system node states (e.g., contains(n0.successor_list, n1.hashId), contains(n1.successor_list, n0.hashId)). This procedure is repeated until the number of node references reaches the maximum allowed (set by the MAX_NF control parameter, 2 in our experiments). And so, one set of predicates is created for each number of node references. Note that the number of generated predicates increases polynomially w.r.t. MAX_NF with degree equal to the arity of the predicates. For example, for binary predicates, the number of predicates increases quadratically with MAX_NF.

Finally, Avenger iterates over each set of predicates and generates the properties by combining through disjunction all the possible combinations of predicates (and their negation) taken one at a time, then two at a time and so on. The maximum number of predicates in a single property is controlled by a parameter MAX_NT. Rather than leaving the user to guess a suitable value of MAX_NT, Avenger provides her the option of specifying the maximum number of properties that she wants to generate in each class. The number of properties with n literals is of the order $O(n^{\text{MAX}_N_T})$.

Lastly, Avenger uses the tree of variables and iterators to produce a valid C++ qualifier for each state variable which is plugged into the properties’ internal representation format in order to produce a series of C++ statements each of which implements the evaluation of a predicate.

### 2.2 Trace Collection and State Iteration

An important challenge is obtaining a sufficiently varied number of distributed system traces. These traces should document the operation of the distributed system under a variety of expected workloads. Perhaps more importantly, the distributed system should be subjected to complex failure scenarios (faultloads). Traditionally, these traces come from live executions or simulations.

However, achieving the desired variety, especially in the case of failures, might require substantial resources and execution time. An existing test suite is of limited use to Avenger as the programming errors it triggers have presumably been fixed.

Avenger assumes that an external module generates a trace of globally consistent system states. These states can be obtained by periodically recording the global state of a live execution; this approach is therefore similar to the one used successfully in WiDS [27]. Alternatively, a module could iterate over the traces of distributed system events, create the global state after each step [16] (e.g., handler execution on a node), and feed it to the state iterator. The traces of the system events can be obtained from the log files recorded during live deployments.

The recent availability of model checking tools for distributed system implementations [22] offers the possibility of quickly obtaining a large number of long executions that explore many possible interleaving of the node actions and failures in so-called random walks. The state space exploration moves quickly between the system states because the model checker acts as a simulator with instantaneous message delivery.

A downside of using random walks is the potential for missing states that are likely to be explored in the actual live runs of the system. A recently proposed alternative that overcomes this shortcoming is to run the state space exploration from live system state snapshots [44]. When using this technique, the messages in the live run are instrumented to carry the checkpoint sequence numbers that force checkpoints to be taken and written to disk when required by the happens-before relationship established by messages [24]. After the live run finishes, the
consistent checkpoints are gathered to form global state snapshots. Then, bounded state space exploration starting from each snapshot can generate state traces that are fed to Avenger.

The consistent snapshots are fed to Avenger one at a time as a set of objects that describe node state (recall that the Variables.h file describes the relevant state). This means that Avenger can be applied to systems written in a variety of programming languages, provided that these systems collect consistent snapshots and convert them to C++ objects to present them to Avenger. Calling existing functions on the state variables might require linking Avenger with parts of the application code that defines the data structures.

2.3 Property Checker

The main task of the Property Checker is to update property statistics that enable filtering and reporting later on. This component is invoked for each global state snapshot. Upon each invocation, the Property Checker evaluates every property (which yields a boolean value) and assigns an individual property score. Finally, it adds the score to the cumulative holding rate which is kept for every property in the pool.

A simple property accounting strategy is to score a property with one if the property holds at all nodes in the snapshot, and zero otherwise. The holding rate statistic is later used by the property filter module, as well as during the final ranking of the properties.

Spatial Accounting One major difference between checking the properties in distributed systems vs. single-machine is that some properties might hold at a subset of nodes but not for all of them. Using the example property \( \forall n \in \text{nodes} : P_1 \), the predicate \( P_1 \) can evaluate to true for only some of the nodes in the global snapshot. We handle cases like this using spatial accounting which computes a fractional score. Specifically, we set the score to the number of times that the property holds divided by the number of combinations of nodes on which the property is evaluated in a snapshot.

Fast Accounting As the property detection process is likely to start with a large number of possible properties, checking the validity of all properties has to be done quickly and efficiently.

Avenger uses the tree structure of variables and iterators used for property generation to derive a dependency structure between predicates and properties for each iterator. Further, the tree maintains information about nesting of iterators which is used to sort the iterators topologically in the sense that if iterator \( A \) is the ancestor of iterator \( B \), the rank of \( A \) is higher. During execution, all the iterators are inserted in a master iterator in this topologically sorted order. For evaluating a set of properties with \( k \) node references, \( k \) copies of node iterators are added to the master iterator. Iterating over the master iterator is just like counting: only a suffix of the digits changes at each increment. Similarly, here at each iteration of the master iterator, there exists \( i \) such that only the iterators with rank less than \( i \) are affected. For each affected iterator, the dependency structure is looked up to re-evaluate its corresponding predicates and properties. Further, in the presence of nested iterators over container variables, all iterators with rank less than \( i − 1 \) are rebound to their corresponding iterable sets.

Scalability The complexity of checking a single snapshot in the trace is directly dependent on the number of nested loops over the system nodes, e.g., with two nested loops (our most common setup), the corresponding complexity of checking a single snapshot in the trace is \( \Omega(n^2) \), where \( n \) is the number of nodes in the system and for the purpose of this discussion it is not changing. The complexity further depends upon the size of the iterable sets in a node’s state. Let us assume that there are \( k \) iterable sets in a node’s state where the size of the \( i^{th} \) set as a function of number of nodes in the system is given by \( S_i(n) \). Then, the time complexity of snapshot evaluation w.r.t the number of nodes can be stated as \( (n * \prod_{i=1}^{k} S_i(n))^{MAX, NF} \). For example, the size of the Chord finger set is \( O(\log(n)) \) and the size of the successor list is \( O(1) \), and thus the resulting complexity is \( O((n \log(n))^2) \). The overall Avenger time complexity increases linearly with the number of snapshots if we assume that the sizes of iterable sets remain the same.

Load Balancing Given the ubiquity of cheap clusters of multi-core machines, an important challenge lies in harnessing the available computational power to speed up checking and, ultimately, the entire property inference process. We have parallelized the property checking task by assigning to each CPU core in the cluster the responsibility of a disjoint subset of snapshots. For this purpose, we use the operation modulo \( N \) over \( S \), where \( N \) is the number of checking processes and \( S \) is the explored state index, so that process \( x \) checks the snapshots for which \( S \mod N = x \). We find this simple load-balancing technique to work well in practice. In our evaluation (Section 3), 10 quad-core machines were sufficient to bring Avenger execution time to less than a few hours for even the longest state traces (Section 3).
The processing units are processes (as opposed to threads). Using the process as the checking unit makes it possible to easily parallelize checking over a cluster of machines. Moreover, because the processes have different address spaces, there is no contention over the shared data structures inside a machine, making it easier for the operating system to perform efficient scheduling. For example, we have observed that the speed of each of the Avenger checking processes running on a multi-core CPU is equal to the speed of a single checking process when it is the only process on that machine. In the end, out scripts take average of the ratios from each process for each property.

2.4 Property Filter

Many generated properties are not likely to be almost-invariants. Keeping them in the pool and evaluating them after every step of the trace would only waste CPU and memory resources. The aim of the property filter is to detect these useless properties early and remove them from the property pool to speed up the checking process. However, care needs to be taken to prevent premature elimination of relevant properties.

Recall that the properties in our interest are the ones which have a holding rate very close to 100%. The properties which have held much less are unlikely to be chosen at the end of the property inference process. When the property filter is invoked, we remove the properties that hold less than a threshold. Overall, there is a trade-off between the tool accuracy and the speed of execution which can be controlled by the threshold filtering value, FILTER_HOLD_RATIO. At the extreme, one could complete the run of the tool while keeping all the properties.

One other parameter we use is the MIN_EVAL_CNT threshold; if the number of performed evaluations is less than this threshold, we refrain from applying the filter.

2.5 Property Reporter

Recall that the Avenger’s main goal is to help identify hard-to-find programming errors by inferring the almost-invariants that get violated during rare manifestation of these errors. Beside the challenges in generating and checking the potential properties, perhaps the most difficult challenge lies in choosing the almost-invariants that will ultimately be reported to the developer. The set of reported properties should ideally reveal all the manifested bugs in the traces of the system run. This set should not be too large, as inspecting too many properties can overwhelm the developer. Accordingly, the falsely reported properties waste developer’s time due to the required efforts to reject them. Thus, one of our design goals is to report only a small number of properties. The property reporter module accomplishes this task, and its REPORT_SIZE parameter can be used to change the number of reported properties (in our experiments, we set this parameter to 10).

Prioritizing properties Given our observation that the bug-manifesting input sequences are usually very rare, we look for properties that hold in most of, but not all, cases. Thus, we sort the potential almost-invariants by their holding rate in descending order. The property reporter then iterates over the candidate list of almost-invariants and applies the following techniques.

Eliminating equivalent properties An important step for improving the quality of the reported almost-invariants is to reduce the number of equivalent properties by reporting only one property from each equivalence set. The properties with the same holding ratio (to a high precision) are likely to be equivalent. We use this simple statistic apart from mathematical equivalence because there could be two properties that are not mathematically equivalent, but are in practice checking the same system aspect. We therefore put the almost-invariants with the same holding rate in the same equivalence set and output only one almost-invariant chosen at random. The original equivalence set can still be delivered upon request. We believe that if a developer finds a property to represent a bug, its equivalence set can give additional insights about the bug.

Simplifying properties Finally, simplify the properties to help reduce the human effort in analyzing the almost-invariants. Recall that each property $Pr$ is a disjunction of some predicates and thus can be split into two or more properties $Pr_1 \lor Pr_2 \lor \ldots \lor Pr_n$ where each simpler property $Pr_i$ contains a disjoint subset of the predicates in $Pr$. Because of what we refer to as the Or Effect, we need to make sure that $Pr$ does not subsume any simpler property that is in the candidate list.

Suppose the simple case where $Pr = Pr_1 \lor Pr_2$. The Or Effect is the case in which property $Pr$ holds at all except few times, but when we take the combined property $Pr$, then $Pr$ covers more states and is invalid fewer times. Hence, $Pr$ would be ranked higher when we sort the candidate list, although $Pr_1$ and $Pr_2$ are independent properties, expressing different aspects of the system. To address this problem, we use a simple independence statistical test. Let $H(Pr_1)$ and $H(Pr_2)$ be the holding rate of $Pr_1$ and $Pr_2$, respectively. The expected holding rate of $Pr$, $\hat{H}(Pr)$, assuming $Pr_1$ and $Pr_2$ are independent, can be calculated using probability theory as:

$$\hat{H}(Pr) = H(Pr_1) + H(Pr_2) - H(Pr_1) \times H(Pr_2)$$

If $Pr_1$ and $Pr_2$ are independent, we expect that the actual holding rate $H(Pr) \approx \hat{H}(Pr)$, then instead of $Pr$ we report the simpler property $Pr_1$. Otherwise, we conclude that $Pr$ has more information than $Pr_1$ and $Pr_2$ individually and thus $Pr$ itself is reported.
Stopping Criteria The stopping criteria specifies when Avenger stops and reports the top-N properties to the developer (this is more applicable for the state traces generated by random walks, as the traces of the running system are typically finite.) We consider two criteria: i) stability of the top almost-invariants, ii) time budget. In the former approach, Avenger stops if the set of the top-N properties does not change after a threshold number of checked snapshots. The latter approach is to stop after running out of the time budget. None of these approaches are sound and complete in the sense that they do not guarantee the almost-invariants of interest to be in the top-N set. We used the second approach for the experimental results reported in Section 3.

3 Evaluation

Our experimental evaluation addresses the following questions: 1) Can Avenger help discover programming errors? 2) Can Avenger uncover distributed system behavior? 3) How long does it take for Avenger to produce the distributed almost-invariants?

3.1 Property Inference for XORP

Given the importance of ubiquitous Internet connectivity, it is paramount for the routing protocols to be error-free. However, recent events have shown [1] that several routing platforms have substantial reliability problems. In this section, we report our results of applying Avenger to XORP [17], a popular open-source routing platform that includes implementations of several important routing protocols (OSPF, BGP, etc.). The XORP version we use is 1.6 (the last version as of Sep. 2009). We concentrate on analyzing the BGP implementation as BGP is the standard inter-domain routing protocol in the Internet.

Implementation Details The changes we made to the XORP source code were minimal, took about two weeks to perform for two researchers (who had no previous experience with the XORP platform), and consisted of: 1) incorporating an existing snapshot algorithm from [44], and 2) writing C++ serialization and deserialization routines for the Routing Information Base (RIB). These modifications allowed us to obtain lightweight, consistent state snapshots over time of all routers involved in the experiment. Avenger’s state iterator processes the deserialized consistent RIB snapshots (in the form of C++ objects) to update the property statistics.

To enable Avenger to generate the potential properties, a researcher that was new to the project spent less than a week to define the variable description file (relevant state) and add new predicates to the predicate list. The variables included in the description file simply reflect the state of the XORP RIB process: the routing tables and their route entries, each made of a network prefix, a metric, a next hop, a pointer to an egress interface, and policy flags. We defined equality predicates for simple data types (e.g., IP addresses, route prefixes) and applied basic knowledge of routing domain to define i) the reachability predicate that checks the membership of a route prefix in a routing table (essentially a route lookup, 1 trivial line of code (LoC) to access a non-standard data structure), and ii) the loop predicate that given a route prefix and an origin node verifies whether following the route prefix in other node’s routing tables leads to a path with loops (20 LoC).

Experimental Setup We deploy multiple instances of XORP routers on a single 48-core machine with 64 GB of RAM, running GNU/Linux 2.6.30-bpo.1-amd64. Our testbed makes use of virtual interfaces to enable multiple XORP instances to communicate over a synthetic topology that is installed within the machine. This relatively simple setup does not delay or drop packets, but it allows us to bring interfaces down and up to simulate link and node failures as to expose BGP behavior. Our experiments assume that all BGP speakers export all the routes they learn, as is the case within the Internet core comprised of tier-1 ISPs. In our experiments, we configure multiple BGP instances in clique (full-mesh) and b-clique topologies with 4, 8, 16 and 32 nodes. These topologies have been used in previous BGP studies (e.g., [31]) as they can expose problematic BGP behavior under certain failure conditions. We conservatively marked snapshots as steady state if there were no UPDATE BGP messages in the past minimum route advertisement period (30 s by default). Inspired by the BGP problems reported in [1], we mixed 16-bit and 32-bit AS numbers in the attempt to uncover unexpected BGP behaviors in our testbed. Finally, we drove the experiments with a script that can advertise and withdraw arbitrary routes, and trigger link and node failures.

3.1.1 Internet Routing Properties: BGP

Starting from the small set of RIB variables and added predicates, Avenger generated 3,700 properties. We measured the time and space required for checkpointing the RIB. Using our unoptimized checkpointing code, an empty RIB takes 0.5 ms to serialize, and consumes 1.8 KB. A RIB with 100 route entries takes 1.5 ms to produce a checkpoint of 6.6 KB. These figures demonstrate that the overhead of checkpointing is small. Avenger identified two relevant almost-invariants (shown in Table 1), out of a total of four reported (10 requested).

Property BGP1: Universal reachability. The first almost-invariant covers complete reachability which is

3During development, the programmer can run Avenger overnight (as with regression suites) and examine the top-N list in the morning. Immediately prior to deployment the testing time is also limited.
One of the most important features of the Internet. This invariant is close in spirit to path visibility, an important BGP property defined in [15], which specifies that every router that has a usable path to a destination learns at least one valid route to that destination. While it is possible that the property is violated during route propagation, it is a necessary requirement for inter-domain routing to converge to full reachability. Surprisingly, in our experiments we found the property being violated a few times during steady state. After careful investigation, we found that the property was systematically violated when two or more 32-bit AS BGP speakers were in the topology and at least one of these speakers was advertising one or more owned route prefixes. Recall that we used a script to automatically generate route announcements and withdrawals at run time.

Note that 32-bit AS BGP speakers are ought to be completely backward compatible with 16-bit AS only BGP speakers [41], although they are identified with the same AS number (called AS_TRANS) from legacy BGP speakers. We inspected the routing table of 32-bit AS BGP speakers and saw that they were not populated of any routes. In fact, any legacy BGP speaker peering with the 32-bit AS BGP speaker would not forward the route announcement if this had an AS_PATH attribute with a reference to another 32-bit AS BGP speaker (the AS_PATH already contained AS_TRANS). We verified that this is a wrong behavior of the BGP implementation in XORP by substituting the legacy BGP speakers with the latest stable version of Quagga (which does not support 32-bit AS numbers), another popular routing platform. As routing tables were now being populated, we grew confidence that Avenger uncovered an unknown bug in XORP and we reported it to the XORP developers who stated that 32-bit AS support has been recently added to XORP and the code has not been tested thoroughly. The current XORP version under development also includes this fault.

Finally, this almost-invariant is evidence that Avenger has all the required expressiveness to produce two iterations over the nodes and one iteration over a custom container type with small amount of developer input (to access a non-standard data structure). In addition, this almost-invariant ties together state from multiple nodes in a distributed system implementation. Property BGP2: No loops. The second almost-invariant states that BGP should not have route loops. BGP is based on a path vector algorithm to ensure no route loops are created. However, protocol convergence is not instantaneous and thus transient route loops might get installed under certain conditions [31]. For example, the b-clique topology causes a loop to exist after an interface failure, which triggered Avenger to identify this anomalous system behavior.

### 3.2 Property Inference for Mace

Mace [21] allows distributed systems to be specified succinctly and outputs high-performance C++ code. In addition, the Mace distribution includes MaceMC [22], a model checker that incorporates exhaustive search and random walk state space exploration techniques. Mace also comes with publicly available implementations of several distributed systems.

In the remainder of this section, we describe the inconsistencies found by Avenger in implementations of RandTree, Chord, and Paxos in Mace [21]. These systems cover important classes of distributed applications: overlay trees, distributed hash tables, and replicated state machines, respectively. The RandTree and Chord implementations were previously manually debugged both in local- and wide-area settings over a period of three years, as well as debugged by MaceMC before being reported in the corresponding paper [22]. Avenger identified the new inconsistencies in the previously debugged [22, 44] Mace code because it was capable of automatically generating the invariants which ended up being violated. In CrystalBall [44] we inspected the code and manually specified additional invariants that were useful in bug-finding, but we missed the important almost-invariants Avenger reported.

### Implementation Details

We have used two options for collecting the traces of distributed system state for Mace. First, our prototype initiates random walks within a model checker from the initial state (like MaceMC does). We altered MaceMC to support long random walks that expose the system to broken TCP connections.

Second, we use CrystalBall [44] to collect a consistent snapshot of the distributed system, and then start a state-space exploration heuristic (consequence prediction) from each of the snapshots. We expose the system in the live run to node churn while the nodes are recording the consistent checkpoints to disk. After the live run finishes, we gather the checkpoints from all system participants. Then, Avenger iterates over the collected global states and uses each of them as the initial state of consequence prediction. Consequence prediction explores many possible interleaving of the node actions as well as connection failures.

We list the details about the overhead of state checking for each system separately. Overall, it takes less than a few seconds to generate the properties for each system. It took even less time to prepare the systems for Avenger.
<table>
<thead>
<tr>
<th>Id</th>
<th>Property</th>
<th>Description</th>
<th>Order/Tot</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGP1</td>
<td>∀n₁ : ∀n₂ : ∀r ∈ n₂.ebgpTable : r ∈ n₁.ebgpTable</td>
<td>Complete network reachability</td>
<td>3/4</td>
<td>11,885</td>
</tr>
<tr>
<td>BGP2</td>
<td>∀n : ∀r ∈ n.ebgpTable : ¬loop(r, n)</td>
<td>Absence of route loops</td>
<td>2/4</td>
<td>11,885</td>
</tr>
<tr>
<td>RandTree1</td>
<td>∀n : n.parent ∈ n.children</td>
<td>Parent is not a child</td>
<td>3/10</td>
<td>20,000</td>
</tr>
<tr>
<td>Chord1</td>
<td>∀n : n.max_succ ≠ n.me ∨ n.pred = n.succ</td>
<td>If the node does not have a successor but itself, it should be the only node in the ring</td>
<td>9/10</td>
<td>36,400</td>
</tr>
<tr>
<td>Chord2</td>
<td>∀n : n.max_succ ∈ n.myRange ∨ n.succ ∈ n.myRange</td>
<td>No successor can be between node and its predecessor unless it is a single node ring</td>
<td>3/10</td>
<td>860,000</td>
</tr>
<tr>
<td>Paxos1</td>
<td>∀n₁ : ∀n₂ : ∀ch₁ ∈ n₁.chosenValues : ∀ch₂ ∈ n₂.chosenValues : ch₁.key ≠ ch₂.key ∨ ch₁.value = ch₂.value</td>
<td>The values chosen by two nodes for the same key (consensus instance) must be the same.</td>
<td>8/10</td>
<td>124,000</td>
</tr>
</tbody>
</table>

Table 1: The almost-invariants detected by Avenger for XORP, RandTree, Chord, and Paxos (of up to 10 requested for each system) that helped in identifying unknown programming errors or system behavior. The "Order" column represents the order of the almost-invariant in the reported list. The last column shows the minimum number of states that need to be inspected to observe the property on the top-10 list. Even the property with most states required took less than four hours to be identified.

Experimental Setup

Our experiments make use of 10 machines with 2.83 GHz Xeon X3360s with 8 GB of RAM, giving Avenger a chance to run in parallel across 40 cores. These machines run GNU/Linux 2.6.26-1. All machines are interconnected by a full-rate 1-Gbps Ethernet switch. In the experiments in which we collect checkpoints from a live run, we run the distributed system on top of the ModelNet [39] network emulator. We use an INET [6] topology that we further annotate with bandwidth capacities, while keeping the default link latencies.

3.2.1 Overlay Tree Properties: RandTree

In this section we complete the discussion of a programming error identified via an almost-invariant that we started in our example (Section 1.2). Using Avenger, we generated 13,096 potential RandTree properties and used long random walks (more than thousands steps) to generate execution traces of a RandTree system consisting of 15 nodes. The average size of a RandTree node checkpoint is 176 bytes. On average, it takes 3.75 ms for Avenger to evaluate the generated properties on each Randtree state. Avenger identified the following almost-invariant:

Property RandTree1: Parent is not a child. As depicted in Table 1, Avenger found the almost-invariant stating that the parent node should not appear as a child. A violation occurs when a node $N$ considers the node $P$ to be both its parent and its child. This inconsistency has not been reported before.

In the violation scenario, node $N$ sends a Join message to node $P$. Upon receipt of the Join message, $P$ adds $N$ into its children list and replies with a JoinReply message. The message can be potentially subject to long delays of transmission and processing. Before receiving the JoinReply message, the join timer of node $N$ expires and sends another Join message to node $R$. $R$ also adds $N$ to its children list and replies by another JoinReply message. $N$, however, rejects the second JoinReply, because $N$ is already joined under node $P$. Therefore, $N$ sends a Remove message back to $R$ to remove $N$ from its children list. Meanwhile, $P$, which has been looking for the global root of the tree, finds $R$ and joins under $R$ by sending a Join message to it. At this time, the Remove message sent by $N$ has not been received by $R$ yet and hence the child pointer to $N$ is still present at $R$. Thus, $R$ forwards the Join message to its child $N$. $N$ accepts the join request of $P$ and adds it to its children list. Now, $N$ points to $P$ both as its parent and its child.

The inconsistency can be fixed by checking the join request before accepting it; if the sender is the current parent, then the request should be rejected. Alternatively, $N$ can choose not to accept messages from $R$ until it acknowledges the receipt of the Remove message that was sent from $N$. This new problem was not reported previously, even though the RandTree implementation had been thoroughly tested in previous works [44, 22].

3.2.2 Distributed Hash Table Properties: Chord

We next describe the results of applying Avenger to Chord [38], a distributed hash table (DHT) that provides key-based routing functionality. DHTs underpin a large number of existing and proposed distributed systems [34, 36], as well as new networking designs [23].
Chord nodes are given an id (key), typically computed by hashing a node’s IP address. Chord places the nodes on a logical ring, in order of increasing ids. The position of the node on the ring determines the set of key-value pairs it is responsible for storing. To help maintain the ring structure, each node keeps pointers to its predecessor, pred, and to the set of its successors, succList. There are separate pointers to the immediate successor, succ, and the maximum successor, max_succ, in the successor list. The variable myRange maintains the range of hash ids between a node’s predecessor and the node itself. When the two sides of the range are equal, the range is considered to be \((-\infty, \infty)\) and hence should include any hash id. For faster lookups, a node also maintains a set of “fingers” that cover exponentially larger distances within the ring. A node joins the ring by querying with its id to identify the node closest to its target position. It then inserts itself in the ring by communicating with its predecessor and successor nodes to let them update their corresponding pointers.

Using Avenger, we generated 51,752 potential Chord properties and used long random walks (with more than thousands steps) as well as the consequence prediction heuristic from a previously recorded consistent snapshot of system state to generate execution traces of a Chord system consisting of 15 nodes. On average, it takes 0.37 ms for Avenger to evaluate the Chord generated properties on each state encountered during random walks in MaceMC. The overall time for the discovered almost-invariants to show up in the top-10 list was reasonable; even in the case of the Chord2 property that required inspection of 860,000 states, the running time of the tool was about 3.5 hours. The average size of a Chord checkpoint is 1028 bytes. The result of running Avenger on Mace Chord is as follows (summarized in Table 1):

**Property Chord1:** \(\text{max}_\text{succ} = \text{me} \rightarrow \text{pred} = \text{succ}\).

By definition, the maximum successor is the largest member of the successor list. The only case where its id can be equal to the node id is the single node ring; i.e., when the node does not have any predecessor or successor. In this case, the id of the predecessor and the successor is equal to the node id.

This almost-invariant led us to find a new inconsistency in the Chord implementation where max_succ was not set properly. The current implementation of the module for updating the max_succ is based on the assumption that the id of the node itself is not included in the successor list. This assumption is inconsistent with the assumptions in the rest of the system that allow the id of the node to be included in the successor list. Due to this inconsistency, the update of max_succ does not work properly in the rare case where the successor list includes the id of the node itself. Then, the max successor becomes equal to the node id, while the node believes

it has some neighbors. This inconsistency was detected using the traces that were obtained by running the consequence prediction heuristic from a previously recorded consistent snapshot of system state.

**Property Chord2:** \(\text{max}_\text{succ} \leq \text{pred} \leq \text{me}\). By definition, the predecessor must be the adjacent node on the ring. Then, no successor s must lie between the node and its predecessor, i.e., \(s \notin \text{myRange}\). Otherwise, the successor is more eligible to be known as the predecessor. Note that this property is defined over local variables and each process must locally ensure consistency of its data. The only exceptional case is the single node ring where there is only one successor and its id is equal to the node id. In this case, the range represented by myRange is infinity and thus includes any id. This almost-invariant was obtained by examining the states encountered during random walks in MaceMC.

In the violation scenario, node n receives find_pred_reply message and updates its successor list accordingly. It, however, does not update the myRange variable. Therefore, in the rare cases where the max_succ happens to be inside the range, there are still other nodes such as the immediate successor, succ, which are not in the range. To summarize, Avenger helped to identify two new programming errors in the Chord implementation. These inconsistencies were not reported previously, even though the implementation had been thoroughly debugged by two different research groups [22, 44].

### 3.2.3 Replicated State Machine Properties: Paxos

In general, the consensus protocol forms the foundation for the replicated state machine approach to fault-tolerant computing. One well-known algorithm for achieving consensus is Paxos [25]. A number of systems deployed in data centers are using Paxos [5, 27].

The Paxos protocol (and thus any implementation) observes the following “built-in” invariants: 1) Only a value that has been proposed may be chosen, and 2) Only a single value is chosen.

We used the baseline Mace Paxos implementation that includes a minimal set of features, with each node playing all the roles (leader, acceptor, and learner). To reflect the fact that implementations can contain errors even when the specification used as a starting point is correct, we injected a previously reported inconsistency [27].

We used Avenger to generate 36,594 potential Paxos properties, and had them checked in model checker’s random walks exercising three nodes. The executions were up to 50 steps long (as these are long enough to cover all node proposals and related messages). Avenger identified the following almost-invariant:

**Property Paxos1:** The values chosen by two nodes for the same key (consensus instance) must be the
same. As Table 1 shows, Avenger inferred the fundamental Paxos invariant. Specifically, if a value is chosen in an instance of consensus, no two nodes should choose a different value. This result shows that Avenger’s grammar is expressive enough to specify this important property, and that the property filtering is able to identify it.

### 3.3 Discussion

Given the difficulty of building reliable distributed systems, a tool that helps to identify hard-to-find programming errors is useful. Avenger was effective for every system we examined. Moreover, two out of four reported invariants turned out to be relevant in the case of XORP, with one leading to a previously unknown problem. However, one might be concerned by the false positive rate for other systems. We note that this kind of rate is typical in anomaly-finding tools [3]. Further, to reduce the false positives it might simply be necessary that more errors exist (the systems we examined were already debugged with a variety of tools). In general however, pointing to more programming errors might require increasing the number of reported almost-invariants, which could further increase the false positive rate.

### 4 Related Work

#### Using invariants during development and deployment.

Some distributed systems and algorithms have been designed from ground up with the invariants in mind. The best example is perhaps Paxos [25], a distributed algorithm for achieving consensus over asynchronous networks.

Killian et al. [22] have manually specified safety and liveness properties in their MaceMC model checker for distributed system implementations, and used them successfully to identify bugs in several systems. WiDS [27], MaceOBB [10] and the work by Singh et al. [37] similarly look for violations of known invariants.

Recently, the work on CrystalBall [44] demonstrated the ability to enforce invariants at runtime to increase the resilience of distributed systems. CrystalBall researchers manually specified all additional safety properties they used in testing. In all these tools, the developer first comes up with the invariants and then tries to find a violation of the specified invariants in the system traces or live state. Avenger goes in the opposite direction in the sense that by checking the traces of the system execution, it detects the inconsistencies and proposes the invariants corresponding to the observed inconsistencies.

#### Invariant inference for single-machine code.

Static analysis [8], which involves source code examination without execution, can be used to infer properties. This approach is typically sound, but, due to some issues (e.g., pointers), static analysis is in practice too conservative.

Dynamic approaches overcome the shortcomings of static analysis by i) generating a large number of possible properties, and ii) relying on test cases to exercise the code behavior. The most prominent system in this domain is Daikon [13, 14]. Daikon filters out any automatically generated property that is violated during a test run. It makes an important assumption, namely that the code or specification that is used to generate the properties is correct. This assumption does not always hold because: i) the source code can simply contain errors, ii) a programmer can introduce errors while transferring an otherwise correct specification to source code [27], or iii) the specification itself can contain errors. Many properties are filtered out after their first violation at the early steps of the experiment. Therefore, Daikon does not face the challenges that Avenger encounters in efficiently evaluating a very large number of properties. To choose among the properties that remain, Daikon uses a statistical test to remove the properties that hold simply by chance. Avenger uses a different insight, namely that the properties that warrant inspection in distributed systems are those that hold in the vast majority of, but not all, the cases. In contrast with Daikon, Avenger does not assume that the traces obtained by test cases are bug-free. It actually takes advantage of the inconsistencies which manifest in the traces to identify the (almost) invariants. Further, Avenger also deals with properties in the spatial domain where only some machines in the system are violating a property at a given point in time.

Instead of falsifying the generated invariants, DySy [9] uses symbolic execution to produce abstract conditions over program variables that the concrete tests satisfy during their execution. Symbolic execution has difficulty working on large programs due to the exponential explosion of possible code paths. Thus, it is not surprising that DySy has been evaluated on short single-machine code, StackAr (stack algebraic data type implemented using an array).

There has also been work on automatically inferring temporal properties for program evolution [45] and specification inference [2, 11, 29]. Other techniques such as program [33] and value spectra [43] have also been used successfully to get a glimpse into the way the code behaves.

Machine learning can help understand program behavior too. In [4], the authors train a machine learning algorithm by using correct as well as faulty source code to predict which properties are most likely to help identify bugs. The performance of such techniques is dominated by the quality of the training set. For example, machine learning could be used to infer safety properties similar to those initially specified by the programmer. However, it would have difficulty in correctly classifying the properties that uncover an unforeseen behavior.
**Self-consistency.** Our work is somewhat related in spirit to checking the self-consistency of source code [12]. This work uses manually-defined templates and performs static analysis of single-machine source code to detect common patterns in sequences of commands. It then reports the deviations from these patterns as inconsistencies to the developer. However, it cannot deal with the bugs that are not a deviation from the similar programming patterns in the rest of the code. For example, the first Chord bug reported in Section 3 is due to the faulty implementation of the function which updates the maximum successor variable. This update happens only in this function and there is no other manipulation of this variable in the rest of the code from which the mentioned tool could detect the deviant behavior. Avenger performs dynamic analysis on the distributed system states to look for almost-invariants, and it successfully reported the almost-invariant that pointed to the bug in the maximum successor update function. DIDUCE [18] is somewhat similar in spirit to Avenger and the work in [12], but it works on single-machine Java code and has significant execution overhead (10x slowdown). ClearView [32] is a system for automatically patching violations of invariants inferred by Daikon. By enforcing invariants at runtime, ClearView is more similar to CrystalBall [44]. In addition, it is operating on single-machine code.

**Test input generation.** Eclat [30] tries to selects a small relevant subset from large set of potential test inputs. It first uses Daikon to infer the operational model of software’s *assumed correct* execution. It then proceeds to reject the test inputs that do not violate the operational model and hence are not likely to reveal faults in the actual program as well. Pacheco and Ernst demonstrate that Eclat successfully generates unit tests for Java classes. This work makes an important assumption that the execution traces which are used to generate the invariants are error-free. Avenger does not make this assumption, and it actually takes advantage of the manifested inconsistencies in the traces to identify the almost-invariants.

**Statistical debugging.** In contrast with the work on generating the test inputs that might point to bugs, the work on statistical debugging tools assumes the existence of so-called bug profiles, and then tries to help the developer in root cause analysis. The bug profiles are assumed to be available because they could have been collected from crash reports or they could have been described by the developer as undesired behavior. Liblit et al. [26] check the validity of a large number of predicates inferred mostly from branch conditions in the program over both faulty and non-faulty execution traces. They then try to isolate predicates that are likely to discover the root cause of the bugs in the single-machine code. The predicates that are presented to the programmer for further inspection are at the level of assertions derived from the existing single-line code statements.

In contrast, the inconsistencies in distributed systems are not usually manifested by crashes and hence detecting them is a challenge of its own. In addition, Avenger does not require the trace to be known as fault-revealing or not. In fact, by identifying the high-level almost-invariants of the system it helps the developer to identify the invariants and consequently the fault-revealing traces which violated them. In addition, the properties Avenger identifies are one step closer to the high-level properties that the programmer reasons about. After the almost-invariants are reported to the developer, he or she can use various debugging techniques, including the work on statistical debugging, to analyze the root cause of the bug.

**Invariant inference for distributed algorithms.** The SPIN [19] model checker can be used to check models (but not implementations) of distributed systems. The work on automatic detection of properties in SPIN [40] can verify whether two variables are related by basic operators. The work that is perhaps closest to ours is [42], whose goal is to infer the specifications of distributed algorithms, which are then passed further to a theorem prover. This work simulates the execution of multiple IO automata that contain a specification of a distributed algorithm in an abstract form. It then uses Daikon to infer the safety properties. In addition to the previously described differences from Daikon, our work produces properties which are corresponding to inconsistencies in distributed system implementations. Working on implementations is important because the programmers can introduce errors even while converting a specification that is proven to be correct in to source code [27]. In addition Avenger deals with properties in the spatial domain.

**Summary.** In summary, the complexity of distributed system implementation opens new challenges in designing invariant inference tools. First, the inconsistencies in distributed systems can result from interleavings of interactions made by multiple nodes and it is not likely that checking deviation from common patterns of use in a single machine can cover the entire space of distributed properties. Second, distributed system invariants tie together pieces of state across multiple nodes; existing tools cannot generate and check such properties. Moreover, the tools designed for single-machine code are not able to deal with properties in the spatial domain (where only some machines in the system are violating a property at a given point in time). In contrast, Avenger successfully deals with such properties in distributed system implementations.

5 Conclusions

In this paper, we tackle an important, but neglected problem of automatically inferring distributed system prop-
erties. Our tool, Avenger: i) generates a large number of potential properties, ii) checks them within the time and spatial domains using traces of system behavior, and iii) chooses only several almost-invariants that warrant inspection by the programmer. Avenger stands to increase the resilience of distributed systems as inspection of these properties can uncover programming errors and system behavior. Our evaluation shows that Avenger indeed helped us identify four new programming errors.

References


