Inferring and Asserting Distributed System Invariants

Stewart Grant
University of British Columbia
Vancouver, BC, Canada
stewbertgrant@gmail.com

Hendrik Cech
University of Bamberg
Bamberg, Germany
hendrik.cech@gmail.com

Ivan Beschastnikh
University of British Columbia
Vancouver, BC, Canada
bestchai@cs.ubc.ca

ABSTRACT

Distributed systems are difficult to debug and understand. A key reason for this is distributed state, which is not easily accessible and must be pieced together from the states of the individual nodes in the system.

We propose Dinv, an automatic approach to help developers of distributed systems uncover the runtime distributed state properties of their systems. Dinv uses static and dynamic program analyses to infer relations between variables at different nodes. For example, in a leader election algorithm, Dinv can relate the variable *leader* at different nodes to derive the invariant $\forall$ nodes $i, j,\ leader_i = leader_j$. This can increase the developer's confidence in the correctness of their system. The developer can also use Dinv to convert an inferred invariant into a distributed runtime assertion on distributed state.

We applied Dinv to several popular distributed systems, such as etcd Raft, Hashicorp Serf, and Taipei-Torrent, which have between 1.7K and 144K LOC and are widely used. Dinv derived useful invariants for these systems, including invariants that capture the correctness of distributed routing strategies, leadership, and key hash distribution. We also used Dinv to assert correctness of the inferred etcd Raft invariants at runtime, using these asserts to detect injected silent bugs.

1 INTRODUCTION

Developing correct distributed systems remains a formidable challenge [7, 58]. One reason for this is that developers lack the tools to help them extract and reason about the distributed state of their systems [44, 49]. The state of a sequential program is well defined (stack and heap), easy to inspect (with breakpoints) and can be checked for correctness with assertions. However, the state of a distributed execution is resident across multiple nodes and it is unclear how to best compose these node states into a coherent picture, let alone check these distributed node states for correctness.

In this work we propose a program analysis tool-chain called *Dinv* for inferring likely data properties, or invariants, between variables at different nodes in a distributed system, and for checking these distributed invariants at runtime.

Dinv-inferred invariants help developers reason about the distributed state of their systems in various ways. In particular, they can confirm expected relationships between variables separated by a network to improve developer confidence in their system's correctness.

For example, consider a two-phase commit protocol [4] in which the coordinator first queries other nodes for their vote and if all nodes, including the coordinator, voted Commit then the coordinator broadcasts a TX Commit, otherwise it broadcasts a TX Abort. At the end of this protocol all nodes should either commit or abort. To check if several non-faulty runs of the system are correct, a developer can examine the Dinv-inferred distributed state invariants for this set of executions. In this case they can check whether Dinv mined the invariant $coordinator.commit = replica_i.commit$ for each replica i in the system. This would mean that the commit state across all nodes was identical at consistent snapshots of the system. They can also use Dinv to add a runtime assertion to check this invariant in future runs.

Dinv is the first *automated* end-to-end tool to infer distributed system state invariants. The closest prior work by Yabandeh et al. [56] requires developers to manually identify variables to log, instrument their systems, identify distributed cuts, and so on. Dinv automates the entire process and requires minimal input from the developer. Dinv is also complementary with many existing tools for checking distributed systems like Modist [57] and D3S [35]. These tools expect the developer to manually specify properties to check; Dinv can make these tools easier to use. Finally, Dinv includes a light-weight and probabilistic assertion mechanism that can detect invariant violations with low, controllable, overhead.

Dinv works by first statically instrumenting the system's code, either automatically or with user-supplied annotations. Dinv uses static program slicing to capture those variables that affect or are affected by network communication at each node. During system execution, Dinv instrumentation tracks these variables, collects their concrete runtime values, tags them with a vector timestamp, and logs the values at each node. Once the developer has decided that the system has run long enough (e.g., execution of a test suite), they run Dinv on the generated logs. Dinv uses vector timestamps in the logs to compose distributed states, and then merges these states using three novel strategies into a series of system snapshots. Dinv then uses a version of the Daikon tool [19] to infer likely distributed state invariants over the tracked variables in the merged snapshots.

Our approach with Dinv is pragmatic: it does not require the developer to formally specify their system and it scales to large production systems and long executions. Although Dinv uses dynamic analysis, which is incomplete (Dinv cannot reason about executions it does not observe), we believe that it is useful because (1) most distributed systems developers today use dynamic analysis to check their systems (e.g., with testing) and (2) we have been able to use Dinv to infer and assert useful properties in several large systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180199

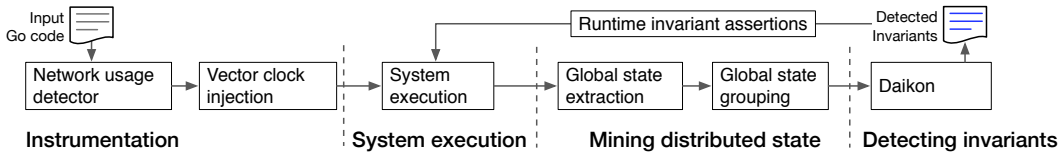

Figure 1: Overview of the invariant inference steps in Dinv.

We evaluated Dinv by using it to study four systems written in Go: Coreos's etcd [14], Taipei-Torrent [27], Groupcache [20], and Hashicorp Serf [25]. We targeted different safety and liveness properties in these large systems and ran Dinv on a variety of executions of each system. For example, etcd uses the Raft consensus algorithm [42] and we checked that the Raft executions we induced satisfy the strong leader principle, log matching, and leader agreement. We used Dinv over several iterations to infer distributed state invariants that confirmed that the executions we studied satisfy each of the targeted properties. We also used Dinv's assertion mechanism to catch bugs injected into Raft that silently violate three key Raft invariants. We evaluated Dinv's overhead and found that it can instrument etcd Raft in a few seconds and that 5 logging annotations in a Raft cluster of 6 nodes induced a 13% system slowdown and used 1KB/s of extra bandwidth.

To summarize, this paper makes the following contributions:

- ⋆ We propose a static analysis technique for automatically detecting variables which comprise distributed state.
- ⋆ We propose a hierarchy of three strategies for grouping global system states (snapshots) for invariant inference and show that each strategy is useful for detecting different types of distributed invariants.
- ⋆ We describe a runtime distributed invariant checking mechanism based on real-time snapshots. We evaluate its efficacy by using it to find injected silent bugs in etcd Raft.
- ⋆ We implemented the above techniques in Dinv, an open source tool [17], and evaluate it on a variety of complex and widely used Go systems that have between 1.7K and 144K LOC.

2 DISTRIBUTED STATE BACKGROUND

In this section we overview our model of distributed state and how it can be observed from the partially ordered logs of an execution.

We consider a system composed of a number of *nodes*, each of which has an independent thread of execution. During a node's execution, each instruction is an *event* and an *event instance* refers to a specific *event* (we sometimes use *event* for both). The *state* of a node at an event instance is the set of values for all the variables resident in the nodes memory. The state of a node can be recorded by writing the variable values to a log. Event instances on a single node are totally ordered.

In this paper we consider message passing systems in which sending and receiving events create a partial ordering of event instances across nodes. Dinv uses vector clocks to establish this happened-before ordering [39]. Using a log of vector clocks, causal chains of events from an execution can be analyzed. We use *log* to refer to the sequence of node states paired with vector clocks generated in a single execution of the system. Section 3.1 discusses how Dinv automatically instruments systems to write states, and vector clocks to a log.

Most of Dinv's analyses run on a log produced by a system after it has executed. Detecting meaningful distributed invariants requires determining valid combinations of concrete node states to use for inference. For a given log, a *consistent cut* is a set of events such that if an event e happened before event f (according to vector clocks), then if f is in the cut, e is also part of the cut. Local states on the frontier of a cut form a *global state*. The complete set of global states which occur during the execution of a system form a lattice. A point in this lattice is an n-tuple of local node states composing a single global state. A lattice edge connects two global states, $g \rightarrow h$, if g happened before h (again, according to vector clocks) and the vector clock timestamps of g and h are separated by a single increment in logical time. A *ground state* is a global state in which all messages sent up until that point have also been received (i.e., no messages are in flight) [1]. Sections 3.2 and 3.3 cover Dinv's global state analysis. Next, we detail Dinv's design.

3 DINV DESIGN

Automatically inferring distributed invariants requires resolving three research challenges:

(1) What state should be logged and when?
(2) How to infer distributed invariants from logged state?
(3) How to enforce inferred distributed invariants?

Dinv's analyses are a step by step procedure for solving these challenges (Figure 1). This section details each step of the analysis.

3.1 System instrumentation

Challenge 1: *What state should be logged and when?* Determining state to log is difficult because state with interesting distributed properties is hard to identify. For example, some state is local to the node and is unaffected by other nodes in the system; distributed invariants over such state are uninteresting. We propose and use the following heuristic: *interesting distributed state must have dataflow to or from the network.*

Variables which interact with the network can be detected statically using program slicing [43, 52]. Forward slices rooted at network reads, and backward slices from network writes identify the affected and affecting variables, respectively. We developed an interprocedural slicing library for Go, and use Go's networking conventions to statically identify network reads and writes. Variables contained in slices rooted at network calls comprise the set of *network interacting variables*. Figure 2 lists partial code from Serf [25] that implements the SWIM protocol [16]. We will use this example throughout the paper. Logging point L1 on line 12 logs variables

Instrumentation strategy	Location choice	Variables choice
Function entrances/exits	Auto	Auto
Network calls	Auto	Auto
User-defined annotations	Manual	Manual or Auto

Table 1: Instrumentation strategies and the control (automatic/manual) offered by each strategy for selecting state logging location and the set of logged variables.

transitively affected by the network read on line 3. Affected variables are underlined in the listing.

When should state be logged? Network interacting variables may be used throughout a system's codebase. Invariant inference depends crucially on where in the code the values of these variables are logged — logging at different points may produce wildly different invariants. For example, to infer the mutual exclusion invariants variables must be logged inside a critical section; if not, then the captured state would not reflect that the node ever executed a critical section, a critical omission!

Dinv provides developers with three mechanisms to control where to log state (Table 1). Two of these automate the choice of locations (function entrances/exits or network calls) and choice of state (all network interacting variables). The third strategy provides the developer with fine-grained control over where and what state to log.

Figure 2 illustrates two logging annotations: *L2* a *//@dump* annotation (line 6) and *L1* a *parameterized Dump* statement (line 12). The first logs distributed state when a *Ping* is received, the second logs state before checking for timeouts.

Tracking partial order. Vector clocks are a canonical means for recording the happens-before relation in a distributed system [39]. Dinv automatically instruments any Go program that uses Go's networking *net* library with vector clocks. It does this by detecting uses of the *net* library and by mutating the abstract syntax tree. Dinv supports common protocols like IP, UDP, TCP, RPC, and IPC. Vector clocks are appended or stripped from network payloads, and the original function is executed on the instrumented arguments. For example, a network write like `conn.Write(buffer)` is transformed into `dinv.Write(conn.Write,buffer)`.

In summary, our solution to challenge 1 is to log only the variables that interact with the network, at automatically generated, and user specified lines of code. Next, we explain how the logs generated by the execution of instrumented nodes are analyzed to infer distributed invariants.

3.2 Extracting global states

Challenge 2: *How to infer distributed invariants from logged state?* An omnipotent observer can establish a total order on all distributed events. In practice, an ordering that is feasible to derive is the *happens-before* relation, defined by the causal precedence of sent, and received messages. The happens-before is a partial ordering on events. We use *lattice* to refer to this ordering.

Each point in the lattice is a consistent cut (defined in Section 2) of the logged execution, and the entire lattice is the set of all consistent cuts [3, 12]. Figure 3 relates a message sequence diagram, to its

```
1  func (s serfNode) serf(conn UDPConnection) {
2   for true {
3    msg := conn.Read()
4    switch msg.Type {
5    case PING:
6     //@dump                                                    L2
7     conn.WriteToUDP("ACK", msg.Sender)
8     break
9    case GOSSIP:
10     s.Events = append(s.Events, msg.Event)
11   }
12   dinv.Dump("L1",msg.Type,msg.Sender,msg.Event,s.Events)      L1
13   timeout := s.CheckForTimeouts()
14   switch timeout.Type {
15   case PING:
16    conn.WriteToUDP("PING",timeout.Node)
17    break
18   case GOSSIP:
19    gossip(s.Events)
20    break
21 }}}
```

Figure 2: Code excerpt from Serf with underlined network interacting variables contained in the forward slice from *conn.Read()* on line 3. Line 6 (*L2*) and line 12 (*L1*) are example instrumenting annotations.

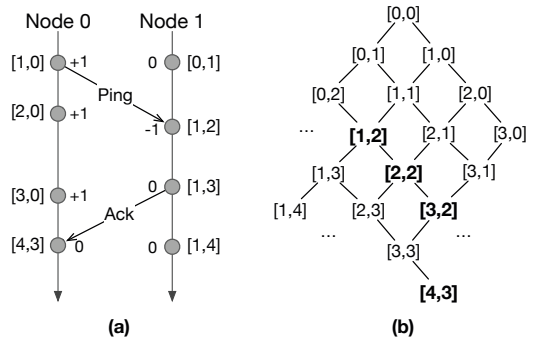

Figure 3: (a) Message-sequence diagram of an execution of code in Figure 2 with two nodes, their local vector clocks paired with each event (in brackets), and message deltas computed for each event (-1,0,+1). (b) (Partial) lattice of the execution. Each vertex displays the corresponding vector time and ground states are bolded.

corresponding lattice for an execution of the code in Figure 2 with two nodes. A lattice point, is a global state composed of an n-tuple of local states. Global invariants, or invariants that hold across multiple nodes, are testable by extracting the corresponding global state from a log, and asserting the invariant on that state.

Due to its generality, this lattice analysis incurs significant complexity. In the worst case, an execution without messages with n nodes and e events will produce a lattice of size e^n.

Because testing invariants on an exponential number of states is infeasible, we propose ground states [1] as a heuristic for reducing the number of lattice points to test for invariants. A *ground state* is a consistent cut with the additional constraint that all sent messages

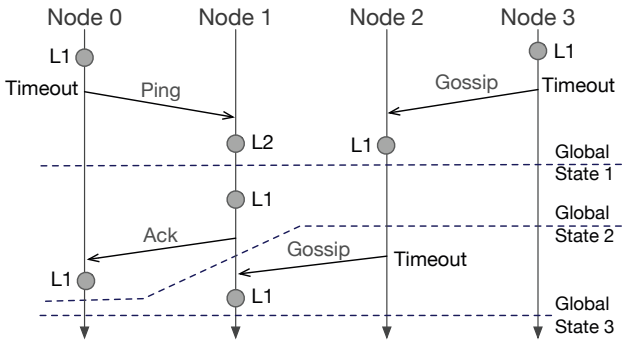

Figure 4: Execution of SWIM code from Figure 2. Node 1 responds to a *Ping* from Node 0. Concurrently Node 2 propagates Node 3's Gossip message. *L1* & *L2* mark local logged state (messages & events in Figure 2). Dashed lines mark three global states (which are also ground states), each an n-tuple of the closest local node states above the dashed line.

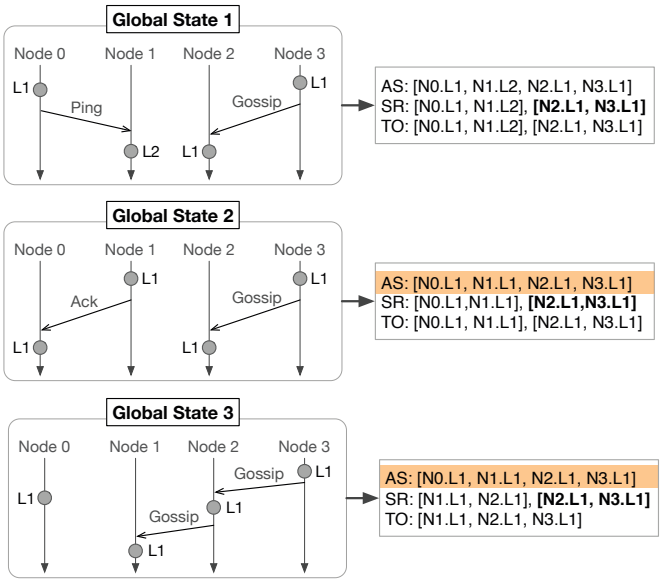

Figure 5: The merging of local states from 3 global states by our three grouping strategies. On the left each Global State corresponds to a dashed line from Figure 4. Logged local states are marked by gray circles, each contains the variables listed on Figure 2 line 12. On the right are group id's from the respective strategies AS: all-states, SR: send-receive, and TO: total order. Highlighted is a group of states merged by all-states. In bold is a group of states merged by send-receive.

have been received (i.e., no messages are in flight). Intuitively a ground state is a line through a message sequence diagram which does not cross any messages (see Global States in Figure 4).

Analyzing just the ground states reduces completeness: a global state which is not a ground state may violate an invariant. However, distributed algorithms are typically specified with invariants over ground states, when the system has acquiesced and all messages have been processed. As well, inference on ground states reduces analysis time by orders of magnitude, and provides a quality sample of global states in real systems[1].

A variety of lattice construction algorithms exist [21]. Long-running executions of loosely communicating systems can easily generate lattices larger than main memory. To avoid this, Dinv utilizes a sequential antichain algorithm which generates a lattice level by level. Generating level n of a lattice requires a log and level $n-1$ of the lattice. This allows Dinv to flush most of the lattice to disk as just two levels must be maintained in memory.

Ground states are computable with a linear scan of both a log and lattice. First, a log is scanned and a *delta* of $(sent - received)$ number of messages is calculated per local event on each node. For example, if by some event e a node had sent 3 messages, and received 1 message, e's delta is +2. The lattice is then scanned, and for each global state the deltas of each local event in a global state are summed. A sum of 0 identifies a ground state (e.g., Figure 3).

Lost messages pose a theoretical threat to ground state analysis: a single message loss rules out future ground states. In practice, lost messages do not affect receiver's state and are functionally equivalent to local events at the sender. Dinv handles executions with lost messages by detecting lost messages using vector clock timestamps and omitting them from the ground state computation.

The first component of our solution to the challenge of inferring distributed invariants from logged state is to infer invariants over ground states. We choose this approach because it is scalable and makes no assumptions about the system.

3.3 Strategies to group global states

Some invariants hold globally at all times, but others hold during protocol-specific event sequences and between select nodes. Without apriori system knowledge, many possible combinations of global states may support or refute an invariant. The possible combinations of global states in a lattice is 2^{e^n}, which is intractable to analyze (and also largely redundant) for real systems. Our goal is to automatically tease apart distinct protocols, group the global states in which they executed, and infer invariants on the group. Our solution relies on the observation that many protocols are specified as causal chains of events. One of our research contribution is a set of 3 strategies for grouping global states for invariant inference:

(1) group all global states together (all-states)
(2) group states by sending and receiving node pairs (send-receive)
(3) group states by totally ordered message sequences (total-order)

Figure 4 is an example execution showing two protocol specific causal chains (*Ping-Ack* and *Gossip-Gossip*), included are three global states (shown as dashed lines). Figure 5 further shows how the global states in Figure 4 are decomposed by each strategy, and how the state tuples with matching identifiers (highlighted) are grouped for further invariant inference. We will use Figure 5 to explain each strategy.

[1] For completeness Dinv allows users to analyze all global states at the cost substantial runtime. All results in this paper are based on ground state analysis.

All-states strategy. The most general form of system invariant, is one which holds on all observable state, and between arbitrary nodes. *All-states* merges all local states of a global state together regardless of causality between them. Each local state has an ID: (*nodeID.logID*), where nodeID denotes the node and logID denotes the logging statement. A merged set of such IDs form a groupID: $[(node_0.log_i), \ldots, (node_n.log_j)]$. Merged local states which share a groupID, are grouped together for invariant inference.

Figure 5 shows *All-states* (AS) merging all local states from the three global states. The groupID resulting from all-states merging *Global State 1* is [N0.L1, N1.L2, N2.L1, N3.L1]; no other merged state shares this groupID. Merged states from *Global State 2*, and *Global State 3*, highlighted in orange, share the groupID [N0.L1, N1.L1, N2.L1, N3.L1]. GroupIDs produce multiple groups when there is more than one logging statement per node. Using the logID as a component of a groupID ensures that each merged state in a group contains exactly the same variables, as separate logging statements need not contain the same set of variables. Invariants inferred by the all-states strategy are the strongest, as they hold across the largest sample of observable states and across all nodes.

Send-receive strategy. Many protocols dictate the behavior between pairs of nodes. The send-receive strategy merges together the states of directly communicating pairs of nodes. Send-receive groupIDs have the form $[(node_p.log_i), (node_q.log_j)]$, where $node_p$ communicated with $node_q$ and log_i executed before the communication and log_j executed after the communication. For example, in Figure 5 Global State 1 send-receive (SR) has two groups: the first group, [N0.L1,N1.L2], corresponding to the *Ping* between Node 0 and Node 1; and, group [N2.L1,N3.L1] captures the *Gossip* message.

An advantage to this strategy is that properties which hold after communication, but not at all times, are tested on subsets of states. Invariants like those that depend on eventual consistency require this for detection. From our running example; each SWIM node maintains a list of *events* which are synchronized with *Gossip* messages. If the invariant *N1.events = N2.events* was tested on the *AS* group from Figure 5 highlighted in orange, the invariant would be violated because in Global State 2 Node 1 has not yet received the *Gossip* message. In contrast, if the same invariant was tested on the send-receive group [N1.L1, N2.L1] from Global State 3, it would hold because Node 1 synchronized its events after receiving the message.

Total-order strategy. Fine-grained protocols, such as leader election, dictate a causal behavior across multiple nodes. The total-order strategy merges the local states from causal *chains* of communicating nodes. The groupID for this strategy has the form $[(node_p.log_i), \ldots, (node_q.log_j)]$, such that the local state $(node_p.log_i)$ happened before $(node_q.log_j)$ and all intermediate (*nodeID*, *logID*) pairs. In Figure 5 the *TO* merged group [N1.L1,N2.L1,N3.L1] from *Global State 3* is the result of merging all local states along the *Gossip* messages causal path from Node 3 to Node 1.

Total-order has the same ability as send-receive to detect eventual consistency in Serf, but it detects it in a stronger context. In the case of *Global State 3* group [N1.L1,N2.L1,N3.L1], the invariant *N1.events = N2.events = N3.events* would be inferred.

Our complete solution to challenge 2 is to infer invariants on groups of global states merged by one of three strategies. These strategies encode heuristics informed by best practices in distributed system design and radically decrease the space of possible groupings. Dinv further scales its analysis by, for example, discarding identical logged instances of node states which span separate global states since these provide no new information.

```
1  func AssertLeadershipAgreement() bool
2    for i:= 0; i < len(assert.Node) − 2 ; i++ // i is a nodeID index
3      if assert.GetVar(i, "leader") != assert.GetVar(i+1, "leader")
4        return false
5    return true
```

Listing 1: A distributed assertion that checks that all nodes in a cluster agree on the leader.

Next, we explain how Dinv infers invariants using a modified version of Daikon.

3.4 Inferring distributed invariants

Daikon [19] is designed for sequential system and does not support inference over partially ordered collections of states with disjoint variable sets. Further, Daikon includes templates for binary and ternary invariants, and does not support n-ary invariants necessary for distributed specifications.

Dinv uses Daikon by presenting it with a synthetic program point that corresponds to a distributed state. However, in a sequential program the same variables are always present at each program point and merged states may be composed of different sets of variables from various logging points. Our solution, reviewed in Section 3.3, is to only merge states with identical sets of variables[2].

We also added several n-ary templates, such as equality, to Daikon. Inferred invariants span the local state of all nodes. For example the group [N1.L1, N2.L1, N3.L1] from Figure 5's Global State 3 *TO* would have the invariant *Node1.Events = Node2.Events = Node3.Events*, rather than the two binary invariants. This reduces effort in comprehending relations spanning more than two nodes.

3.5 Asserting inferred invariants

Challenge 3: *How to enforce inferred distributed invariants?* Dinv includes an assertion library to help developers check inferred distributed safety properties at runtime with user-defined assertions (Listing 1 shows an example). Prior approaches to checking distributed predicates [35, 46] rely on variants of the global snapshot algorithm based on logical clocks [9]. By contrast, Dinv uses a light-weight *real-time global snapshot* algorithm for assertions.

Dinv's real-time assertions have 3 components: a round trip time (RTT) estimator, a physical clock synchronizer, and an assertion algorithm. The RTT estimator periodically pings other nodes and computes an estimate of RTT between each node. To synchronize clocks Dinv uses the Berkeley algorithm [24].

The assertion algorithm works as follows. When a node *A* executes an assertion statement, *A* blocks until the asserted predicate is resolved. First, using the RTT estimator, *A* schedules a state snapshot time t that is in the future by the largest RTT from *A* to the other nodes. Next, *A* sends a snapshot request with time t and requested variable names to all nodes. On receiving a request from

[2] As a more advanced heuristic, Dinv also supports analysis of intersections of logged variables

A, a node *B* creates a thread that sleeps until *t*. Once *B* reaches *t*, this thread snapshots the values of the requested variables and sends these to *A*. Once *A* has received all the snapshots it needs, it evaluates the asserted predicate.

Scheduling snapshots with physical clocks, even if they are synchronized, has the disadvantage that the resulting snapshot may violate the happens-before relation (e.g., node *C* snapshots its state, send a message to node *D*, and then *D* snapshots its state). To avoid inconsistent snapshots Dinv uses vector clock timestamps to determine if a snapshot represents a ground state for the system. If not, Dinv logs the failure to assert, skips the assertion, and will retry the assertion the next time it is reached.

Due to blocking semantics, asserts in frequently executed code can reduce performance. Dinv allows developers to mitigate this with *probabilistic assertions*, which execute with some user-defined probability. We expect developers to use high probabilities during testing, for precision, and low probabilities in production, for performance.

Our solution to challenge three is a mechanism for probabilistic distributed assertions.

4 IMPLEMENTATION

Dinv is implemented in 8K[3] lines of Go code [17]. It has been tested on Ubuntu 14.04 & 16.04, and relies on Go 1.6. Dinv implements an optimized version of the vector clock algorithm and uses a manually-constructed database of wrapper functions for Go's *net*. We use Go's AST library to build, traverse, and mutate the AST of a program for instrumentation. Dinv's state instrumentation builds on control and data flow algorithms in GoDoctor [28]. Dinv uses Daikon version 5.2.4.

5 APPLYING DINV TO COMPLEX SYSTEMS

In this section we describe our methodology for evaluating Dinv on complex systems in Section 6. For our example we use Serf's eventually consistent group membership property. Prior to this evaluation we had no knowledge about the systems we analyzed, with the exception of the paper describing Raft [42]. In the evaluation we used Dinv in concert with documentation and source code to understand each system. We highlight four techniques we used and that we believe make Dinv more usable.

When applying Dinv to a new codebase we used its completely automated facilities to **survey the systems invariants** to learn about its behavior. Initially, we instrumented Serf[4], which injected 400 logging statements that logged 20-40 variables each. We ran the system's test suite, and processed the logs with Dinv. The merging strategies parsed the log into 1000 groups of global states. In aggregate across all groups, Daikon inferred approximately 1 million invariants, many of which related constants. We used the massive set of invariants to **identify variables** relevant to consistency. Using line numbers as indexes into the source code we **refined our analysis** to a smaller set of functions.

A second execution, resulted in 50 groups and a total of 1,700 invariants. *all-states* invariants falsified node state equality, so we deduced that consistency did not hold globally at all times. We examined *send-receive* invariants for a fine-grained view of the system. The updates were fully observable by monitoring assignments to a single structure which maintained the cluster's health, so we **composed dump statements** to instrument this structure. Running the tests again, while logging only the cluster's health, resulted in 25 groups, with a total of 40 inferred invariants. *Send-receive*, and *total-order* outputs were composed of the desired equality invariants between the node states.

In our evaluation of Dinv with three other complex systems, we followed the above approach of iterativly refining the logged variables to zero in on properties over key distributed state.

To generate assertions from the inferred invariants we manually wrote boolean functions (e.g., Listing 1) to check an invariant across nodes at runtime. Section 7 describes our experiences with using the assertion mechanism.

[3] All LOC counts in the paper were collected using cloc [2]; test code is omitted from the counts.

[4] Serf uses encoders and required the manual addition of 20 lines, one for each sending and receiving line of code.

6 EVALUATION: INFERRING INVARIANTS

In the following section we use Dinv to analyze four systems: Hashicorp Serf [25] (our running example), Groupcache [20], Taipei-Torrent [27], and Coreos's etcd [14]. We describe each system and their properties, and report on invariants detected by Dinv and what they tell us about the correctness of the system. Table 2 overviews the invariants we targeted in our study.

Experimental setup. All inference experiments were run on an Intel machine with a 4 core i5 CPU and 8GB of memory, running Ubuntu 14.04. All applications were compiled using Go 1.6 for Linux/AMD64. Experiments were run on a single machine using a mixture of locally referenced ports, and iptable configurations to simulate a network. Runtime statistics were collected using runlim [47] for memory and timing information, and iptables for tracking bandwidth overhead.

6.1 Analyzing the SWIM protocol in Serf

Serf [25] is a system for cluster membership, failure detection, and event propagation. Serf has 6.3K LOC and is used by HashiCorp in several major products. Serf builds on a gossip protocol library based on SWIM [16].

Each SWIM node maintains an array of all other nodes liveness state: alive, suspected or dead. A suspected failure is gossiped when heartbeat messages are not acknowledged, and complete failures (dead) is gossiped once a specified subset of nodes suspect a failure. Throughout this process, node state updates are attached to pings, ping-reqs and acks, and spread by gossip messages to ensure eventual consistency. A receiving node *j* applies updates it receives only if it does not have more recent information. Dinv can observe this property by logging state changes, i.e., node *i* sets node *k*'s state to *alive*. The **send-receive** strategy inferred the invariant $node_i.stateOfK = node_j.stateOfK$, for all pairs of nodes *i* and *j*. Further, the **total-order** strategy inferred the invariant $node_i.stateOfK = node_j.stateOfK = \cdots = node_m.stateOfK$ on all transitive sequences of gossip messages on clusters up to 4 nodes. The detection of this invariant required all orderings of gossip propagation to occur many times and thus required the longest executions, and analysis time.

System and Targeted property	Dinv-inferred invariant	Description
Serf Eventual consistency	$\forall$ nodes i, j, $NodeState_i = NodeState_j$	Nodes distribute membership changes correctly.
Groupcache Key ownership	$\forall$ nodes i, j, $i \neq j$, $OwnedKeys_i \cap OwnedKeys_j = \emptyset$	Nodes are responsible for disjoint key sets.
Kademlia Log. resource resolution	$\forall$ request r, $\sum$ (msgs for r) $\leq \log(\lvert peers \rvert)$	All resource requests must be satisfied in no more than $O(log(n))$ messages.
Kademlia Minimal distance routing	$\forall$ key k, node x, if XOR(k, x) minimal, then x stores k	DHT nodes stores a value only if its ID has the minimal XOR distance in the cluster to the value ID.
Raft Strong leader principle	$\forall$ follower i, len(leader log) $\geq$ len(i's log)	All appended log entries must be propagated by the leader.
Raft Log matching	$\forall$ nodes i, j, if i-log[c] = j-log[c] $\rightarrow$ $\forall(x \leq c)$, i-log[x] = j-log[x]	If two logs contain an entry with the same index and term, then the logs are identical in all previous entries.
Raft Leader agreement	If $\exists$ node i, s.t. i leader, then $\forall j \neq i$, j follower	If a leader exists, then all other nodes are followers.

Table 2: Invariants listed by system, their corresponding distributed state invariants, and descriptions.

In instrumenting network calls, two code paths had to be considered, one for TCP and one for UDP. Dinv's automatic instrumentation worked for UDP. In the TCP case custom stream decoding prevented automatic instrumentation; instead, we wrote 20 LOC to insert/extract vector clocks.

We setup an execution environment where nodes were periodically partitioned to force frequent propagation of membership updates. Observing 3-4 nodes in an execution with 100 such partitions resulted in Dinv inferring all invariants. We were also able to observe and gather similar results about Serfs' behavior in more complex executions, i.e., round-robin partitions.

An execution with 100 partitions was running for 24 minutes[5] and produced 1.6 MB of log files, which Dinv analyzed in less than 2 minutes. The results and lack of contradicting invariants leaves us confident that Serf's update dissemination is correct.

6.2 Analyzing Groupcache

Groupcache is an open source Go implementation of memcached, written in 1786 LOC [20]. Groupcache nodes act as both clients and servers for key requests. Like memcached, Groupcache assigns key ownership to nodes, but nodes hold no state apart from multiple caches. Each node is responsible for a unique set of keys which it owns exclusively.

Groupcache requires users to provide a *Getter* function which maps keys to values. *Get* messages are encoded using Protobuf and exchanged over HTTP. Protobuf encapsulates Go's standard networking library, making the calls invisible to Dinv's vector clock instrumentation. We manually augmented the HTTP header with vector clocks, which required 6 additional LOC.

Because of Groupcache's static key partitioning, the invariant $node_i.keys \neq node_j.keys$ holds globally at all times, and not on precise protocol-specific sequences. Our Groupcache test program was run on configurations of 2–8 nodes, each of which requested 2K keys. Dinv detected the central key distribution property (see Table 2) with each merging strategy. Here *all-states* provides the strongest evidence of the invariants correctness, as the key ownership can be demonstrated to hold on all observable global states.

The key ownership property was quickly identified with Dinv by a third-year undergraduate student, who had little experience with Dinv or Groupcache.

6.3 Analyzing Taipei-Torrent

Taipei-Torrent is an open source BitTorrent client and tracker. Its client program uses Nictuku's implementation of the Kademlia distributed hash table (DHT) protocol to resolve peer and resource queries [29, 40]. Taipei-Torrent and Nictuku are implemented in 5.8K and 4.9K LOC, respectively.

Kademlia uses a virtual binary tree routing topology structured on unique IDs to resolve resources and peers. Peers maintain routing information about a single peer in every sub-tree on the path from their location to the root of the tree. Kademlia has 2 primary types of messages: *Store* and *Find_Value*.

Store instructs a peer to store a value. *Find_Value* resolves requests for stored values. A peer's response to a *Find_Value* query is the list of peers on its sub-tree with the closest XOR distances to the requested value. *Find_Value* is executed iteratively on the peer list until the value is found. These queries are resolved within $O(log(\lvert peers \rvert))$ where $\lvert peers \rvert$ is the total number of peers.

We automatically injected vector clocks into Taipei-Torrent in 3s. Manual logging functions were used because variables containing routing information were not readily available. We introduced our own counter with 2 lines of code to track the number of *Find_Value* messages propagated in the cluster. Taipei-Torrent has sparse communication between nodes; the result is a large space of partial orderings. Lattices built from the traces of Taipei-Torrent consisted of 20–100 million points. Log analysis took upwards of 15 minutes, with an upper limit of 2 hours, requiring frequent writes to disk as the lattice exceeded available memory. Dinv succeeded in analyzing these executions, although the communication pattern was a challenge for our techniques. Lattice inflation limited our analysis to executions with at most 7 peers.

Kademlia specifies that peers must store and serve resources with the minimum XOR distance to their IDs. Further, *Find_Value*

[5]Serf was given 7 seconds after and before each partition to detect and propagate membership changes.

requests must resolve to the minimum distance peer. To test the correctness of *Find_Value* requests we added a 5 line function which output the minimum distance of the peers and resources in the routing table and logged it. To test routing we ran clusters with 3–6 peers using a variety of topologies by controlling peer IDs. We logged state after the results of a *Find_Value* request were added to a peer's routing table. On each execution we found that ∀ peers i, j, $peer_i.min\_distance = peer_j.min\_distance$ in all total-order groups. This invariant, in conjunction with $O(Log(n))$ message bound, provides strong evidence for the correctness of Nictuku's implementation of Kademlia.

6.4 Analyzing etcd Raft

Etcd is a distributed key-value store which relies on the Raft consensus algorithm [42]. Raft specifies that only leaders serve requests, and followers replicate a leader's state. Followers use a heartbeat to detect leader failure, starting elections on heartbeat timeouts. Etcd is used by applications such as Kubernetes [32], fleet [13], and locksmith [15], making the correctness of its consensus algorithm paramount to large tech companies such as eBay. Etcd Raft is implemented in 144K LOC.

Etcd uses encoders to wrap network connections, so manual vector clock instrumentation was required. Log analysis took between 10-15s. Etcd was controlled using scripts. One to launch a clusters of 3-5 nodes, another to partition nodes, and one to issue a 30s YCSB-A workload (50% put, 50% get requests) [11].

Strong leadership. An integral property of Raft is strong leadership: only the leader may issue an append entries command to the rest of the cluster. This property manifests itself in a number of data invariants. A leader's log should be longer than the log of each follower. Further, the leader's commit index, and log term should be larger than that of the followers. We logged commit indices, and the length of the log. In each case the invariant $leader.logsize \geq follower.logsize$, and $leader.commitIndex \geq follower.commitIndex$ was detected by the *send-receive* strategy.

Log matching. Raft asserts "*if two logs contain an entry with the same index and term, then the logs are identical in all entries up to the given index*" [42]. This property is hard to detect explicitly because it requires conditional logic on arrays. We were able to detect that in all cases $node_i.commitIndex = node_j.commitIndex \wedge node_i.log[commitIndex] = node_j.log[commitIndex] \rightarrow node_i.log = node_j.log$ up to the *all-states* grouping. This shows that if any two nodes have the same log index, and the value at that index match, their entire logs match; this is evidence of the log matching property.

Leadership agreement. At most one leader can exist at a time in an unpartitioned network, and all unpartitioned members of a cluster must agree on a leader after partitioning. By logging leadership state variables when leadership was established, we were able to derive that: $node_i state = Leader \rightarrow \forall$ j $node_j.leader = node_i \wedge \forall j \neq i, node_j.state = Follower$. These invariants were detected in both *send-receive* and *total-order* groups. This indicates that after the partition occurred, all nodes agree on a leader, and that all nodes but the leader are followers.

Strong leadership, log matching, and leadership agreement are invariants of a correct Raft implementation. By checking their existence, we produced strong evidence for the correctness of etcd Raft.

Raft invariant	LOC	P=1.0	P=0.1	P=0.01
Strong leadership	11	0.07	0.05	2.96
Leadership agreement	13	0.36	0.34	6.75
Log matching	72	2.22	4.35	6.07

Table 3: LOC to implement and time (sec) to detect an invariant violation with probabilistic asserts.

Further, we have shown Dinv's ability to detect useful properties over distributed state of large and non-trivial system.

7 EVALUATION: ASSERTING INVARIANTS

Dinv-inferred invariants can be used for comprehension. However, they can also be converted into assertion predicates to find regression errors at runtime. Here we detail how we used the Dinv assert mechanism to check the inferred etcd Raft invariants at runtime.

We developed distributed assertions for each of etcd's invariants. We then evaluated the ability of these assertions to find bugs by using them with buggy versions of Raft. For this we manually created three bugs, each of which violates one of the three Raft invariants. All bugs cause a violation *without* causing Raft to crash, or impact its ability to serve client requests. That is, each bug produces silent errors and is difficult to detect.

Strong leadership bug. In Raft only the leader may issue the command to append entries to a replicated log. In our two line bug an unauthorized follower broadcasted append entries, and committed to its own log. Raft's algorithm tolerates this bug because the leader has authority to overwrite followers logs. However, once a leader has written to disk in a term, the system expects that all followers' logs are synchronized. Etcd does not verify synchronization so the bug causes the leader to perpetually issue log correction messages to the buggy follower. The invariant for strong leadership is that the leaders log size is greater than all the followers' logs if the leader has committed in the current term.

Leadership agreement bug. If a leader exists in a given term, all nodes must agree on this leader for leadership agreement to hold. We introduced a 4-line bug which caused followers to randomly select a leader from their list of peers post election. Etcd continues to execute with this bug. However, followers periodically time out waiting for messages from a false leader and initiate a new election. In a non-buggy execution if any two nodes agree on a leader for a given term then they agree on the same leader.

Log matching bug. Log matching is critical to etcd's fault tolerance. If log matching does not hold etcd's key value store returns inconsistent results depending on which node is the leader. Etcd assumes that all log entries written to disk are correct. To violate the log matching invariant we injected a 7-line bug to corrupt a committed log at a random place and time. With this bug etcd executes as normal and assumes that the nodes' logs are correct. The log matching invariant states that if any two nodes have an entry with the same term, index, and data, then all prior log entries match.

Table 3 shows our experimental results. Assertions for above invariants ranged in size. Log matching was the most complex (72 LOC): checking it requires a comparison of logs from every pair of nodes. The assertion iterates through all pairs of node logs, checking them for inconsistencies. The other two assertions were expressed in under 15 LOC.

Number of annotations	Executed annotations	Log size (MB)	Runtime (s)	Runtime overhead %
0	0	0	2.66	0
1	2.8K	3.2	2.70	1.5
2	5.6K	4.3	2.77	4.0
5	14K	9.7	3.01	12.9
10	28K	18.0	3.31	24.3
30	85K	51.7	4.48	68.0
100	261K	167.9	7.66	187.5

Table 4: Impact of Dinv annotations on Raft performance.

We ran Raft with each bug and used assertions with probabilities of 1.0, 0.1, and 0.01. We measured the average time delay between the instant a bug was injected and when it was detected. We found that all asserts found the bugs, but they took longer with lower probabilities. Considering the severity of these bugs, we believe that the delay of a few more seconds to detect the problem is reasonable (given no other alternative). We discuss the associated decrease in overhead with using probabilistic assertions in Section 8.1.

8 EVALUATION: DINV OVERHEAD

Dinv imposes several overheads. These include the time to instrument the system, runtime and network overheads due to logging and injected vector clocks, and the running time of the dynamic analysis that Dinv must perform on the collected logs. This section details these overheads.

Static analysis runtime. To benchmark the performance of Dinv's static analysis (detecting networking calls, adding logging code, etc) we used etcd Raft, which contains 144K LOC and thousands of variables. We measured instrumentation time with increasing counts of randomly located *dump* annotations. Instrumentation time remained constant at 3s until 4K annotations at which point it increased slightly to 3.2s. At 64K annotations (far beyond practical use) runtime was 4.7s.

Logging overhead. Logging state at runtime slows down the system. We instrumented etcd Raft with increasing number of logging statements, each one logging 7 variables. We benchmarked a cluster with 3 nodes, and a YCSB-A workload. Each cluster was run 3 times and we averaged the total running time. Table 4 shows a linear relationship between the number of logging statements and runtime. In practice just two annotations were sufficient to detect the Raft invariants. The average execution time of a single logging statement is 20 microseconds. In our local area network with a round trip time of 0.05ms while running etcd with 1 second timeouts we can introduce approximately 50K logging statements per node before perturbing the system.

Bandwidth overhead. Vector clocks introduce bandwidth overhead. Each entry in Dinv's vector clocks timestamp has two 32 bit integers: one to identify the node, and the other is the node's logical clock timestamp. The overhead of vector clocks is a product of the number of interacting nodes in an execution and the number of messages: $64bits \times nodes \times messages$. To evaluate bandwidth overhead in a real system we executed etcd Raft using the setup above while varying the number of nodes. The bandwidths of all nodes was aggregated together for these measurements. We found that adding vector clocks to Raft slowed down the broadcast of heartbeats and caused a *reduction* in bandwidth of 10KB/s for all nodes in a 4 node cluster. At 5 nodes and above the bandwidth overhead grew linearly with an overhead of 1KB/s for 5 nodes and 10KB/s for 6 nodes.

System runtime (s)	Raft log (MB)	Raft analysis (s)	GCache log (MB)	GCache analysis (s)
30	5.1	12.7	0.3	2.8
60	10.5	28.1	0.3	3.0
90	13.7	35.9	1.7	19.6
120	17.4	48.7	1.4	21.2
150	22.5	68.8	1.8	11.3
180	27.7	99.1	2.1	18.6

Table 5: Generated Dinv log size and Dinv's dynamic analysis running time for varying system run times, for two systems: etcd Raft and GroupCache (GCache).

Dynamic analysis runtime. Dinv's dynamic analysis runtime is affected by the size of the log and the number of nodes in the execution. To measure its performance versus the length of execution, we analyzed etcd Raft and Groupcache. We exercised them by issuing 10 requests per second to each system. To demonstrate how Dinv's analysis performs with regard to the length of execution, we analyzed the resulting logs of 3 node clusters, which were run for intervals in increments of 30s. Results in Table 5 show that Dinv's log analysis scales linearly with system running time.

To measure how analysis time is affected by the number of nodes in an execution we ran etcd for 30s, exercising it with 10 client requests per second and running clusters with increasing number of nodes. Our results show that Dinv's runtime grows exponentially with the number of nodes. We measured analysis times of 25s, 75s, and 725s for logs containing 4, 5, and 6 nodes, respectively. Dinv's runtime is exponential in the number of nodes due to the exponential growth of partial orderings our analysis techniques compute. This indicates that Dinv is currently limited to analyzing distributed systems with a small number of nodes.

8.1 Distributed assertions overhead

We evaluated the overhead of Dinv's assertion mechanism on Microsoft Azure. The setup consisted of 4 VMs (3 servers and 1 client), all running Ubuntu 16.04. The server VMs had 3.5GB of memory and a single core capable of performing 3200 IOPS. The client was used to saturate the servers and had 16 cores and 56GB of memory, and could perform 51200 IOPS. Below we measure the end-to-end latency of client requests to the etcd cluster.

We established a baseline using unmodified etcd. The system was exercised at 3 load levels: 100, 150, and 200 client request per second, each test was run for 100s. Each assert in Section 7 was run under the same conditions. Assertions were placed in etcd's inner event loop which executed on every received message and timer event. On average 5 events occurred per client request. We also ran experiments with probabilistic asserts with two probabilities: P=0.1 and P=0.01 and measured the median slowdown in client request response times.

The greatest slowdown was incurred when asserts were placed at bottleneck program points. For example, asserting strong leadership with P=1.0 caused a 52x slowdown, as each client request was forced to wait for multiple asserts to execute. Using P=0.1 reduced this to 2.5x, and P=0.01 reduced it further to 1.02x. Leader agreement and log matching asserts were performed by followers, which are not

on the critical path for client request processing. Both assertions with P=1.0 introduced only a 1.09x slowdown.

9 DISCUSSION

Effort in using Dinv. In our evaluation we considered four large distributed systems, none of which we were familiar with prior to this study. In each case we used all of the resources available to us (papers, source code, documentation) to understand the desired system properties and to interpret Dinv's output. As we had no prior knowledge about the four systems in our evaluation, and were successful in inferring interesting properties, we are confident that with proper training developers would be able to similarly instrument their systems.

Although we did not formally evaluate Dinv with developers, we do have two pieces of anecdotal evidence that Dinv is not difficult to use. First, graduate students with no prior distributed systems background successfully used Dinv on their systems in a distributed systems course. Second, Groupcache (Section 6.2) was analyzed by an undergraduate student who was familiar with distributed systems but not with Dinv. After installing Groupcache and becoming familiar with its test suit, he was able to isolate the key distribution invariant within a workday. Although anecdotal, we believe these experiences indicate that Dinv is usable by developers. We plan to evaluate Dinv with developers in our future work.

Dynamic analysis. Dinv infers *likely* invariants because it is a dynamic analysis approach that only considers a finite set of system behaviors. The inferred invariants are not a verification of the system, but they could be used for runtime checking (as we demonstrated in Section 7), or to bootstrap verification [41].

Executions containing failures. Dinv's inference pipeline was designed to infer invariants from executions with no node failures. Dinv's assertion mechanism, however, can detect invariant violations even when failures occur.

10 RELATED WORK

Mining distributed systems information. Dinv is a specification mining tool that builds on Daikon [19], which cannot mine distributed state invariants on its own. Daikon has been previously used to assist a theorem prover in verifying distributed algorithms by running over simulated execution traces [41].

The closest work to Dinv is work by Yabandeh et al. [56] who infer almost-invariants in distributed systems: invariants that are true in most cases. They also build on Daikon but the process of identifying variables to log, instrumenting the system, piecing together distributed cuts and composing logs from different nodes, is a manual process. Our approach actually instruments the system, computes ground states, and also checks invariants at runtime.

Other approaches that mine distributed/concurrent specifications produce symbolic message sequence graphs to group machines into classes based on similarities in their communication patterns [33], LTL properties relating events between nodes [6], and infer communicating finite state machines [5]. This prior work focuses on *events* and can trace its roots back to Cook and Wolf's original work that noted concurrency as a challenge [10]. None of these techniques can detect distributed *data* properties.

Other work mines a variety of distributed system information for SE purposes. For example, some work uses mining to detect dependencies [37], anomalies [53, 55], and performance bugs [48].

Other analysis of distributed systems. Dynamic analysis of distributed systems has yielded several tools to aid developers. For example, DistIA [8] implements impact analysis, Googles Dapper [51] analyzes traces to produce call graphs and performance information, and *lprof* [59] instruments Java bytecode with synchronized timestamps and uses logs to infer temporal properties.

Two prior tools use Daikon to derive invariants of networked and concurrent systems. InvarScope [23] detects invariants in JavaScript applications, but does not generalize beyond client-server systems. Udon [34] infers data invariants of multi-threaded programs where program state is shared between threads.

Monitoring systems such as Fay [18] and Pivot tracing [38] use dynamic instrumentation for real-time diagnosis of distributed systems by activating trace points at runtime. These tools do not infer properties from the traces they capture.

Formal methods for distributed systems. Unlike recent methods that use theorem proving to synthesize correct systems by construction [26, 50, 54], our work is immediately applicable to existing production systems. Previous work also considers checking existing system implementations directly [31, 57], or checking system properties at runtime or during program replay [22, 30, 35, 36, 45].

Dinv also includes a runtime assertion checking mechanism. But, in contrast to prior work like D3S [35], Dinv's mechanism *schedules* node state snapshots using synchronized physical clocks and uses probabilistic assertions to decrease overhead.

More fundamentally, previously work assumes that a developer can, and is willing to, specify properties of their system. By contrast, Dinv does not require the developer to formally specify their system and aims at elucidating the runtime properties of the system.

11 CONCLUSION

Distributed state is a key element of distributed systems that impacts consistency, performance, reliability, and other system features. However, distributed state is difficult to tease out, understand, and check. We presented a novel automated analysis approach to (1) identify distributed state, (2) instrument it and record it at runtime, (3) combine it using three different strategies, and (4) use it to infer likely distributed state invariants. We also introduced a lightweight probabilistic assertion mechanism to check distributed state invariants at runtime using real-time snapshots.

We realized our approach in Dinv, a tool for systems written in Go. We evaluated Dinv with four complex and widely used systems. Our evaluation demonstrates that Dinv can infer critical correctness properties of these systems, and that Dinv assertions can detect silent violations of these properties. For example, Dinv detected a violation of each of the three invariants of etcd Raft in under 7s with assertion overhead of just 1.02x.

Dinv is an open source tool [17].

Acknowledgments

This work was supported in part by an NSERC Discovery Grant, Huawei, and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

REFERENCES

[1] M. Ahuja, A. D. Kshemkalyani, and T. Carlson. A basic unit of computation in distributed systems. In *International Conference on Distributed Computing Systems (ICDCS)*, 1990.

[2] AlDanial. cloc: Count Lines of Code. https://github.com/AlDanial/cloc, 2016.

[3] O. Babaoglu and M. Raynal. Specification and Verification of Dynamic Properties in Distributed Computations. *Journal of Parallel and Distributed Computing*, 28(2):173 – 185, 1995.

[4] P. Bernstein and E. Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[5] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *International Conference on Software Engineering (ICSE)*, 2014.

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining Temporal Invariants from Partially Ordered Logs. *SIGOPS Oper. Syst. Rev.*, 45(3):39–46, Jan. 2012.

[7] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems: Challenges and options for validation and debugging. *Communications of the ACM*, 59(8):32–37, Aug. 2016.

[8] H. Cai and D. Thain. DistIA: A Cost-effective Dynamic Impact Analysis for Distributed Programs. In *International Conference on Automated Software Engineering (ASE)*, 2016.

[9] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, Feb. 1985.

[10] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-based Data. *ACM TOSEM*, 7(3):215–249, July 1998.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing (SoCC)*, 2010.

[12] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, 1991.

[13] CoreOS. A Distributed init System. https://github.com/coreos/fleet, 2013.

[14] CoreOS. Distributed reliable key-value store for the most critical data of a distributed system. https://github.com/coreos/etcd, 2013.

[15] CoreOS. Reboot manager for the CoreOS update engine. https://github.com/coreos/locksmith, 2014.

[16] A. Das, I. Gupta, and A. Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *International Conference on Dependable Systems and Networks (DSN)*, 2002.

[17] Dinv homepage. https://bitbucket.org/bestchai/dinv/.

[18] Ú. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *Symposium on Operating Systems Principles (SOSP)*, 2011.

[19] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[20] B. Fitzpatrick. Groupcache. https://github.com/golang/groupcache, 2014.

[21] V. K. Garg. Maximal Antichain Lattice Algorithms for Distributed Computations. In *Distributed Computing and Networking*, pages 240–254. Springer, 2013.

[22] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007.

[23] F. Groeneveld, A. Mesbah, and A. Van Deursen. Automatic invariant detection in dynamic web applications. Technical report, Delft University of Technology, Software Engineering Research Group, 2010.

[24] R. Gusella and S. Zatti. The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE TSE*, 15(7):847–853, July 1989.

[25] Hashicorp. Service orchestration and management tool. https://www.serf.io/docs/internals/gossip.html, 2014.

[26] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Symposium on Operating Systems Principles (SOSP)*, pages 1–17, New York, NY, USA, 2015. ACM.

[27] Jackpal. A(nother) Bittorrent client written in the go programming language. https://github.com/jackpal/Taipei-Torrent, 2010.

[28] R. A. Jeff Overbey. Go Doctor - The Golang Refactoring Engine. http://gorefactor.org/index.html, 2014.

[29] Y. Junqueira. Kademlia/Mainline DHT node in Go. https://github.com/nictuku/dht, 2012.

[30] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[31] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007.

[32] Kubernetes. Production-Grade Container Scheduling and Management. http://kubernetes.io/, 2014.

[33] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Inferring Class Level Specifications for Distributed Systems. In *International Conference on Software Engineering (ICSE)*, 2012.

[34] M. Kusano, A. Chattopadhyay, and C. Wang. Dynamic Generation of Likely Invariants for Multithreaded Programs. In *International Conference on Software Engineering (ICSE)*, 2015.

[35] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, 2008.

[36] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Symposium on Networked Systems Design & Implementation (NSDI)*, 2007.

[37] J. G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. *SIGOPS Oper. Syst. Rev.*, 44(1):91–96, Mar. 2010.

[38] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Symposium on Operating Systems Principles (SOSP)*, 2015.

[39] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[40] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[41] T. Ne Win, M. D. Ernst, S. J. Garland, D. Kırlı, and N. Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.

[42] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*, 2014.

[43] K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *SIGPLAN Not.*, 19(5):177–184, Apr. 1984.

[44] J. K. Ousterhout. The Role of Distributed State. In *In CMU Computer Science: a 25th Anniversary Commemorative*, pages 199–217. ACM Press, 1991.

[45] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[46] K. Romer and J. Ma. PDA: Passive distributed assertions for sensor networks. In *International Conference on Information Processing in Sensor Networks (IPSN)*, 2009.

[47] RunLim. RunLim. http://fmv.jku.at/runlim/, 2016.

[48] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[49] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[50] I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and Proving with Distributed Protocols. In *Symposium on Principles of Programming Languages (POPL)*, 2018.

[51] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[52] N. Walkinshaw, M. Roper, M. Wood, and N. W. M. Roper. The Java System Dependence Graph. In *International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[53] R. J. Walls, Y. Brun, M. Liberatore, and B. N. Levine. Discovering specification violations in networked software systems. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[54] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[55] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[56] M. Yabandeh, A. Anand, M. Canini, and D. Kostic. Finding Almost-Invariants in Distributed Systems. In *International Symposium on Reliable Distributed Systems (SRDS)*, 2011.

[57] 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 *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[58] P. Zave. Using Lightweight Modeling to Understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, Mar. 2012.

[59] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Symposium on Operating System Design and Implementation (OSDI)*, 2014.