Checking distributed systems using likely state invariants

Paper # 323

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 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.

Dinv works on systems implemented in Go and has negligible instrumentation overhead. We applied it to several popular distributed systems, such as etcd Raft, Hashicorp Serf, and Taipei-Torrent, all of which have thousands of lines of code and are widely used. For these systems Dinv derived useful invariants, including invariants that capture the correctness of distributed routing strategies, leadership, and key hash distribution.

1 Introduction

Developing correct distributed systems remains a formidable challenge. One reason for this is that developers lack the tools to help them extract and reason about the distributed state of their systems [31]. 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 states into a coherent picture, let alone check it for correctness.

In this work we propose a program analyses tool-chain called *Dinv* for inferring likely data properties, or invariants, between variables at different nodes in the system. For example, consider a two phase commit protocol 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.

Dinv-inferred invariants help developers reason about the distributed state of their systems in various ways. For example, they can improve developer understanding of their systems and confirm expected relationships between variables across the network. They can also be used by test-case generation techniques to drive the system towards invalid states (i.e., states that violate an inferred invariant). These invariants can also help with debugging: a mined invariant that violates the developer's mental model of what should be true can be tracked back to observed concrete values, and can be used to minimize the execution [36].

Dinv first statically instruments the system's code, either automatically or with user-supplied annotations. Dinv uses static program slicing to capture those variables that are affected by or effect 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., includes the behavior of interest), they run Dinv on the generated node logs. Dinv uses these logs to construct a lattice of distributed states. Our key technical contribution is to propose three heuristics for merging these distributed states into a series of system snapshots. Dinv then uses a version of the Daikon tool [10] to infer likely distributed state invariants over the tracked variables in these 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 ob-

serve), 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 validate useful properties in several large systems.

We evaluated Dinv by using it to study four systems written in Go: Coreos's etcd [6], Taipei-Torrent [17], Groupcache [12], and Hashicorp Serf [15]. 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 [29] 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 report on Dinv's overhead, finding that Dinv can instrument etcd Raft in a few seconds and that 10 logging annotations in a Raft cluster of 6 nodes induced a 24% system slowdown and 13KB total extra bandwidth on an 30 second execution.

To summarize, this paper makes the following contributions:

- We propose a method for inferring likely distributed state invariants in complex distributed systems. Our method relies on well-known program analyses, but combines them in a novel manner.
- We developed three heuristics for merging distributed node states into three kinds of distributed snapshots. We show that each heuristic is useful for inferring different types of distributed invariants.
- We describe Dinv, an open source tool that implements the proposed method, and evaluate it on several complex Go systems that are in widespread use.

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. Appendix A contains a more formal description.

During a system's execution, each instruction is an *event* and an *event instance* is a reference to a specific *event*. The *state* of a node at an event instance is the set of values for all the variables resident in memory. The state of a node can be recorded by writing the variable values to a log. A *state transition* is the change to some variable values in response to an event instance. An execution of a node is a totally ordered sequence of events corresponding to state transitions. In a message passing system the sending and receiving events allow a partial ordering of events across nodes. Vector clocks can establish this ordering [27]. We will use *log* to refer to the sequence of node states paired with vector clocks generated in a single execution of the system.

Most of Dinv's analyses run on a log produced by a system after it has executed. Determining the combination of states across the nodes that should be used to detect meaningful distributed invariants requires additional constraints. A *strongly consistent cut* is a consistent cut in which all messages sent up until that point have also been received (i.e., no messages are in flight). We also use strongly consistent cut to denote the corresponding set of system node states. Inferring state invariants over a strongly consistent cut ensures that no messages in flight can invalidate an invariant on arrival. The complete set of strongly consistent cuts which occur during the execution of a system provide a step-by-step view of the system's global state transitions on which Dinv performs its analysis.

To infer distributed invariants Dinv aggregates the state of nodes in a strongly consistent cut into a combined global state. It does this in several ways, each of which is a *merging strategy*. These produce alternative views of a system. We refer to points at which two or more states are merged by our log merging algorithm as a *distributed program point*. Next, we describe Dinv's design.

3 Dinv design

Applying Dinv to a system consists of four steps (Figure 1): (1) source code instrumentation, (2) system execution to generate runtime logs, (3) mining of distributed program states from the logs, and (4) determining likely distributed state invariants from distributed program points. This section details each of these steps.

3.1 System instrumentation

Dinv instruments a node's source code to produce a runtime log containing vector-timestamped node state (each process maintains its own vector clock). Maintaining vector clocks (see Appendix B for the algorithm) and logging of variables require separate forms of instrumentation. We developed automated techniques for both.

Injecting vector clocks. Dinv introduces vector clocks automatically by mutating the AST and by exploiting the conventions followed by Go's networking *net* library. This library implements TCP, IP,

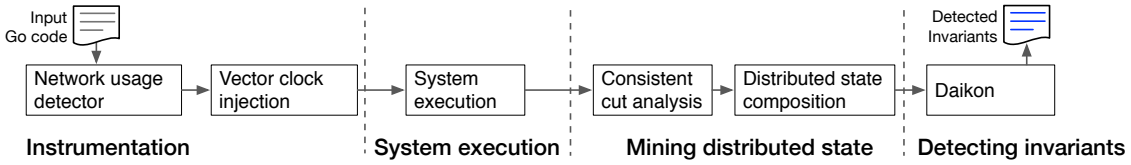

Figure 1: Overview of the steps in Dinv.

```
1  func (s serfNode) serf(conn UDPConnection) {
2   for true {
3    msg := conn.Read()
4    switch msg.Type {
5    case PING:
6     //@dump
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)
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: Example illustrating network interacting variables (underlined) contained in the forward slice from $msg := conn.Read()$. Two annotations on lines 6 and 12 are also highlighted.

UDP, RPC, and IPC protocols; all of these, except for RPC, share function signature conventions which Dinv wraps: 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)` becomes `dinv.Write(conn.Write,buffer)`. For Go's RPC, which required a different approach, we built a custom codec. By using these two techniques Dinv adds vector clocks to any Go program that uses the *net* library.

Logging node state. We designed Dinv with the assumption that interesting state in a distributed system is composed of variables which interact with the network. Data read from a network transitively affects variables which interact with it, while variables which affect the contents of network writes have a transitive influence on the behaviour of the node that receives the message. These *network interacting variables* influence the state of other nodes, and properties that range over these variables capture information about the consistency of distributed state. Dinv determines the network interacting variables statically using interprocedural program slicing [30, 38]. It provides developers with three options (Table 1): (1) automatically log these variables at all function enter/exit points, (2) automatically log these variables at developer-annotated points, or (3) log variables interesting to the developer at developer-annotated points.

Instrumentation strategy	Location choice	Variables choice
Entrance and exit of functions	Auto	Auto
User placed annotations	Manual	Auto
Paramatrized logging functions	Manual	Manual

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

Dinv also allows the developer to choose between two strategies for recording state: `dump` records values at the dump statement program point, while `track` delays recording of values until just before a network communication event. That is, track statements aggregate state across different program scopes for a more complete view, but they are not as precise as dump statements.

Figure 2 lists partial code from Serf [15] that implements the SWIM protocol [8]. We will use this example throughout the paper. This code has two logging points in the form of `//@dump` annotation (line 6) and a *paramatrized Dump* statement (line 12). The first logs state when a *Ping* is received, the other logs state before checking for timeouts. The code also illustrates the variables affected by a network read (underlined).

The remainder of this section explains how the runtime node logs are merged together, how the states of independent nodes are combined into distributed program points, and how Dinv infers distributed data invariants from these combined states.

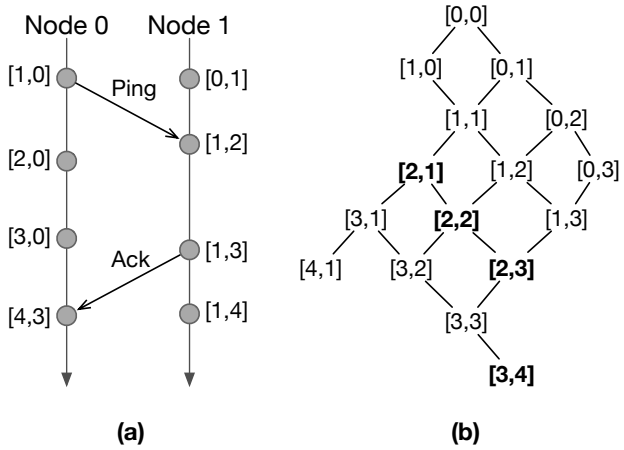

Figure 3: (a) Space-time diagram of an execution of code in Figure 2 with two nodes, Node 0 and Node 1. **(b)** Lattice corresponding to this execution with strongly consistent cuts indicated in bold.

3.2 Extracting strongly consistent cuts

A lattice contains all event orderings consistent with the happens-before relation [4]. This property constrains the exponential set of potential event orderings and supplies a complete view of a systems execution. A variety of different algorithms for constructing a lattice have been proposed [13]. Appendix D presents our algorithm for constructing the lattice.

Figure 3 shows the time-space diagram and corresponding lattice for an execution of two nodes, Node 0 and Node 1, each of which runs the code in Figure 2. Node 0 sends a *ping*, executes two local events, then receives an *ack* message.

Within this lattice, strongly consistent cut can be computed by enumerating sent and received messages. We use the set of strongly consistent cut in the lattice as our granularity for inferring invariants (highlighted in bold in Figure 3B). Merging the states of interacting nodes at the granularity of strongly consistent cut guarantees that the sum of node states is representative of the state of the system. Algorithm 6 shows our process for extracting consistent cuts from a lattice.

3.3 Merging cuts into distributed program points

To check state invariants on multiple nodes, their states are merged together and their variables are tested for invariants. All logged states are related to the program points where they were logged. We therefore define a merged set of states to be a *distributed program point*. Determining which states to merge when checking for particular invariants is difficult because distributed systems have a variety of communication topologies.

Data: A system log L
Result: A set of consistent node states S
$clocks$:= vectorClockArraysFromLogs(L)
$lattice$:= buildLattice($clocks$)
$deltaComm$:= enumerateCommunication($clocks$)
$cuts$:= mineConsistentCuts($lattice$, $clocks$, $deltaComm$)
S := statesFromCuts($cuts$, $clocks$, $logs$)
return S

Algorithm 1: High level log merging overview

For example, nodes in a client-server system execute different code, while nodes in a distributed hash tables are identical peers whose behaviour is dictated by their unique identifier. In each case the system's invariant properties hold at different levels, and require alternative views of the execution to test them. Checking that all nodes in an election agree on a result, or that all DHT peers agree on where to route a request requires a system wide view. Checking that a server responds idempotently to client requests only requires a view between a client and the server. Checking that state updates are propagated through a gossip protocol requires a view specific to the dataflow in the system.

Instead of a one-size-fits-all approach we use 3 heuristics for merging node state for inferring invariants: (1) merge the states of all nodes in a strongly consistent cut, (2) merge only the states of unique pairs of senders and receivers, and (3) merge the states of nodes that participated in a totally ordered communication.

Whole cut merge. This merge is performed by collecting the states of all hosts in a strongly consistent cut and testing them for invariants together. Invariants which are true for all nodes on all strongly consistent cut are checked at this granularity. For example in a distributed hash table where resources are not duplicated across multiple nodes an invariant such as $\forall i, j \in nodes$ $i - keys! = j - keys$ would be detected. In other cases whole cut merge will not detect invariant properties. Figure 5 shows the whole cut merged points from a Serf execution. Checking Serf's eventual consistency invariant with whole cut merge would not work. In the example, if the whole cut merge distributed program point from $C1$ and $C3$ were checked for invariants, Dinv would detect that $N_0 - Events! = N_3 - Events$ because the gossip message from N_3 had yet to propagate to N_0. Whole cut merge falsifies Serf's consistency invariant because at the granularity of a strongly consistent cut the property does

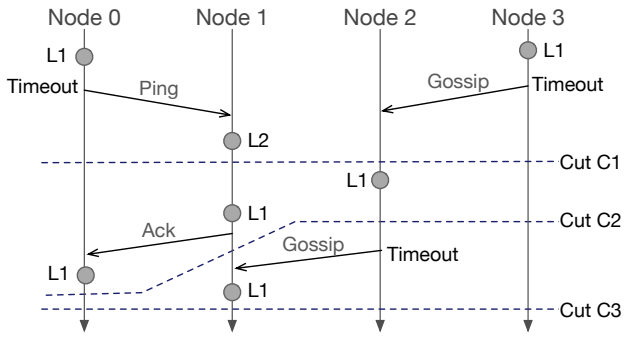

Figure 4: Sample SWIM execution with 4 nodes running code from Figure 2. This diagram is a subset of a larger execution. Points $L1$ and $L2$ on each node timeline represent the execution of two separate logging functions. $L1$ and $L2$ corresponds to lines 12 and 6 in Figure 2 respectivly. Dashed lines $C1$, $C2$, $C3$ are strongly consistent cuts of the execution including just these 4 nodes.

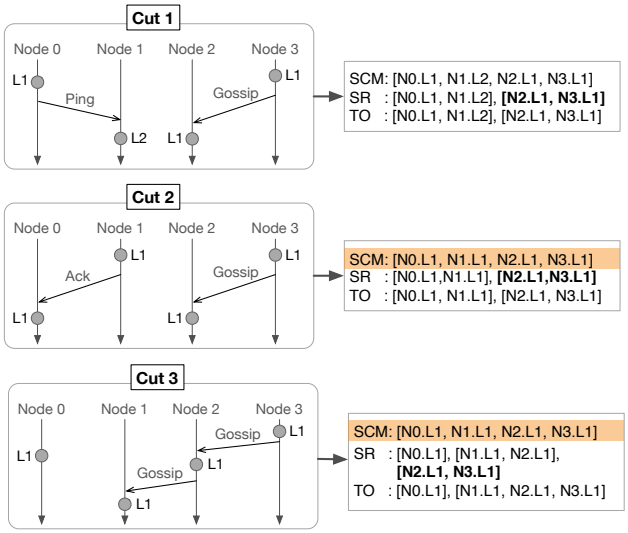

Figure 5: Illustration of the three merging strategies that convert cuts into sets of distributed program points. On the left, are cuts $C1$, $C2$, $C3$ from Figure 4 depicted as sets of logged state. The right side shows how the three merging strategies, **WCM**: whole cut merge, **SR**: send-receive merge, and **TO**: total order merge, produce different distributed program points. Highlighted are two matching distributed programs points.

not hold.

Send-receive merge. In some cases it is necessary to analyze the states of pairs of interacting nodes. In such cases merging the states of senders and receivers in a strongly consistent cut can be used to verify invariant properties of their isolated interaction. Figure 5 details how send-receive merge constructs distributed program points. The state of a sender immediately before a send is merged with the state of the receiver immediately after receiving. *Cut C3* details how states are merged in cuts where nodes both sent and received messages, as well as nodes which did not communicate. In the case of no communication the node state is left isolated. Whole cut merge fails to detect eventual consistency, but the invariant can be verified with send-receive merge. In the case of Figure 5 inferring invariants over the send-receive distributed program points would result in the invariants $N_3 - Events == N_2 - Events$, $N_2 - Events == N_1 - Events$, and $N_1 - Events == N_0 - Events$. At every instance that nodes communicate their knowledge of events is synchronized. Therefore, the invariant holds at a sending and receiving granularity. Send-receive merge is useful for detecting fine grain invariants, but because the state of all nodes is not analyzed together the properties lack generality, requiring users to confirm the invariant between every pair of nodes.

Total ordering merge. It is often useful to delineate between differing messaging protocols between nodes. In such cases merging node states which participate in a totally ordered interaction is useful. Vector clock values within a strongly consistent cut form a directed acyclic graph (DAG). A totally ordered set of clock values can be extracted by starting at the most recent clock in the cut, and backtracking along the first incident edges. When backtracking ends at a terminal node of the DAG, the path from the most recent clock to the terminal node, is a total ordering. The total ordering can be removed from the cut, and what remains is a sub cut of vector clocks, which is also a directed acyclic graph. This process can be performed iteratively to extract all total orderings within the cut, Algorithm 2 lists this procedure.

The set of all distributed program points is computed by running this algorithm on the set of all strongly consistent cuts. Figure 5 details how this merging strategy differs from send-receive merge. Specifically, in *Cut C3* the states of the three totally ordered nodes are merged together. Total ordering merge has the same ability as send-receive merge to detect eventual consistency in Serf, but it detects it in a broader scope. In the case of *Cut C3* the invariant is $N_3 - Events == N_2 - Events == N_1 - Events$.

Data: $State$
Result: Set of distributed program points within a state $DPP's$
$clocks$ = s.getClocks()
dag = DagFromClocks($clocks$)
while dag.*Root != nil* **do**
  $path$ = dag.BacktrackFromRoot()
  $point$ = getHostStatesFromClocks($State$,$path$)
  dag.extract($path$)
  $DPPs$.append($point$)
end
return $DPPs$

Algorithm 2: Extract the set of distributed program points from a consistent distributed state

3.4 Bucketing distributed program points by type

To derive arbitrary invariants multiple instances of variables and their values are required. Properties which hold on all instances are invariant. To test invariants on the state of two or more nodes the combined set of variables must be identical in each instance that an invariant is tested. Testing invariants on instances with non-identical sets of variables would identify correct invariants on the intersection of the sets, and potentially incorrect invariants on the sets symmetric difference. Because of this we only test invariants on identical sets of states.

The final process to prepare the logs for Daikon is to bucket distributed program points together. Distributed program points are identifiable by the triple *(nodeid, source, loc)* of the logging statements from which generated them, and the set of nodes from which are merged. Figure 5 highlights two distributed program points generated by whole cut merge from different points in Serf's execution, which would be bucketed together. The points emphasized in bold generated by send receive merge correspond to the same distributed program point merged from separate cuts, these points are also bucketed together. We consider all *Dump* statements to be unique as they do not necessarily contain the same set of variables. Therefore, regardless of merging strategy, the number of different buckets generated can be exponential in the number of *Dump* statements. *Dump* statements provide a precise view of particular functionality, but also cause an output explosion. *Track* consolidates each statement into one, the result is a collapse of exponential output at the cost of precision. For example using whole cut merge and 2 *Dump* statements on a log of 4 nodes results in a total of 16 distinct distributed program points, whereas *Track* in conjunction with single cut merge results in 1 distributed program point. This exponential reduction is true for all merging strategies.

The bucketing algorithm is as follows: distributed program points are placed into buckets using their id's as identifiers. Finally each bucket is written to a separate file for analysis by Daikon.

3.5 Using Daikon to infer invariants

Daikon is a tool to dynamically detect likely data invariants [10]. We use a Daikon on the logged concrete values in each bucket to infer invariants. Daikon infers likely data invariants based on data traces, referred to as *dtraces*. The result of running Dinv on a log, is a set of *dtraces* for each unique distributed program point present during the systems execution. Dinv output a dtrace file for each set of matching distributed program points. Daikon is run on each *dtrace* to detect invariants at the corresponding point, Daikon's output is a set of invariants with variables prefixed by the nodes id, source file, and line of code on which they were present to distinguish them from one another.

IB ►*Have to describe mechanisms inside of Daikon for handling spurious invariants.*◄

IB ►*Have to note somewhere (here?) that more executions that are diverse improve the quality of the mined invariants.*◄

Appendix F details Daikon's approach to invariant detection, and our methodology for translating Daikon invariants into first order logic.

4 Implementation

Dinv is implemented in 8k [1] lines of Go code. 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 functions in *net* and wrappers for these functions. We use Go's AST library to build, traverse, and mutate the AST of a program (e.g., to search for networking functions in our database). Dinv's state instrumentation builds on control and data flow algorithms in GoDoctor [18]. Dinv requires a working installation of Daikon version 5.2.4 or newer.

5 Applying Dinv to complex systems

Running Dinv on a system produces a set of output files containing the likely data invariants for each bucketed set of distributed program points. Reasoning about Dinv's

[1] All LOC counts in the paper were collected using cloc [1]; test code is omitted from the counts.

output and managing it's size is a non-trivial task. In the following section we describe our methodology for applying Dinv to the systems in Section 6 using Serf as an example, and identify 5 general techniques (listed below) for performing invariant inference with Dinv.

When applying Dinv to a new codebase we made use of Dinv's automated facilities to take a **survey of the systems invariants** to learn about its general behaviour. Encoder-wrapped connections are not handled by our instrumentation. When encoders are used we manually applied vector clocks. Serf makes uses of encoders and required manual instrumentation. We initially automatically injected logging code into Serf. The result was the addition of 400 statements ranging from 20-40 variables. We executed the system and used *whole cut merge* on the logs. Dinv merged 1,000 distinct distributed program points, with a total of 1,000k invariants. Our goal was to check the eventual propagation of membership changes. The invariant output **identified variables** containing node state, and used their corresponding line numbers to **refine our analysis** to a small set of functions. A second execution using *whole cut merge*, resulted in 50 distinct distributed program points with a total of 1,700 invariants. The output falsified node state equality, so we **deduced that at a whole cut granularity** changes were not fully propagated at all time. We switched to *send-receive merge* to apply a fine grained approach. The updates were fully observable by monitoring assignments to a single variable, so we **determined dump statements** would be sufficient to check the invariant. Running the cluster again with these setting merged 25 distinct distributed program points, with a total of 40 invariants. The output was composed of equality invariants between the node states. We followed this approach of iterative refinement to check invariants on all systems in the next section.

6 Evaluation: checking systems

In the following section we use Dinv to check different aspects of four systems: Coreos's etcd [6], Taipei-Torrent [17], Groupcache [12], and Hashicorp Serf [15]. We describe each system and the properties we target, and then report on the 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. We ran all experiments on an Intel machine with a 4 core i5 CPU and 8GB of memory. The machine was running Ubuntu 14.04 and all applications were compiled using Go 1.6 for Linux/amd64. Experiments were run locally on a single machine using a mixture of locally referenced ports, and iptable configurations to simulate a network. Runtime statistics were collected using runlim [33] for memory and timing information, and iptables for tracking bandwidth overhead.

6.1 Checking SWIM protocol in Serf

Serf [15] is a system for cluster membership, failure detection, and event propagation using gossip-style communication. HashiCorp uses it in all of their major products, including terraform, nomad and vault. Serf builds on a library that implements the gossip protocol SWIM [8], and incorporates multiple improvements.

In SWIM each node is in one of three states: alive, suspected, and dead. SWIM has a failure detector and an update dissemination component. The failure detector periodically pings a target node, which is expected to acknowledge the message. On timeout, the probing node asks other nodes to repeat the ping. In case one of them receives a response from the target, the acknowledgement is sent back to the original node and the failure detection cycle ends with no change of target node state. In case none of the nodes receives a response from the target, a timeout is set during which the target node's state is set to suspected. If any node receives a message from the target during this time, the timeout is cleared and the node's state is changed back to alive. Otherwise the node is marked dead.

Throughout this process, node state updates are attached to ping, ping-req and ack messages and disseminated by dedicated gossip messages to ensure eventual consistency. A receiving node j applies state 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 $alive$, and processing them with the **send-receive merge strategy**: $stateOfK_i = stateOfK_j$.

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 from message buffers.

We setup an execution environment where one node was partitioned off and then re-joined the network, to force frequent propagation of membership updates. Observing 3-4 nodes in an execution with 100 such partitions, resulted in Dinv inferring all invariants. In addition, we were able to observe and gather similar results of Serfs' behavior in more complex executions, i.e., round-robin partitions.

An execution with 100 partitions was running for 24

System Targeted property	Dinv-detected invariant	Description
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.
Kademlia Log. resource resolution	$\forall$ request r, $\sum$ (msgs for r) $\leq$ log($\|peers\|$)	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.
Groupcache Key ownership	$\forall$ nodes $i, j, i \neq j$, $OwnedKeys_i \cap OwnedKeys_j = \emptyset$	Nodes are responsible for disjoint key sets.
Serf Eventual consistency	$\forall$ nodes i, j, $NodeState_i = NodeState_j$	Nodes distribute membership changes correctly.

Table 2: Invariants listed by system, their corresponding data invariants, and descriptions

minutes[2] and produced 1.6 MB of log files, which Dinv analyzed in less than 2 minutes. The gathered results and the lack of contradicting invariants leaves us confident that Serf's update dissemination is correct.

6.2 Checking etcd Raft

Etcd is a distributed key-value store which relies on the Raft consensus algorithm [29]. Raft specifies that only leaders serve requests, and followers replicate a leaders state. Followers use a heartbeat to detect leader failure, starting elections on heartbeat timeouts. Etcd is used by applications such as Kubernetes [22], fleet [5], and locksmith [7], making the correctness of its consensus algorithm paramount to large tech companies such as eBay. Etcd Raft is implemented in 288K 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 64 put and get requests over 30s. to the cluster. We used the **whole cut merging strategy** for our analysis.

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

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 indeces, and the length of the log. In each case the invariant (leader-logsize) $\geq$ (follower-logsize), and (leader-commitIndex) $\geq$ (follower-commitIndex) was detected.

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*" [29]. This property is difficult 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). These relations show that if any two nodes have the same log index, and the value at that index match, then their entire logs match, which is evidence that log matching is satisfied.

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

post 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. This invariant show that post partioning all nodes agree on a leader, and that all nodes but the leader are followers.

Strong leadership, log matching, and leadership agreement are each invariants of a correct raft implementation. By checking their existence, we have demonstrated strong evidence for the correctness of etcd Raft. Further we have shown Dinv's ability to detect invariant properties of large and non-trivial system.

6.3 Checking Taipei-Torrent

Taipei-Torrent is an open source BitTorrent client and tracker, which uses the nictuku distributed hash table (DHT) [19] for routing resources, and peer detection. Taipei-Torrent and Nictuku are implemented in 5.8K and 4.9K LOC, respectively. Nictuku implements the Kademlia DHT protocol [28].

In Kademlia peers and resources have unique 160 bit IDs. Kademlia uses a virtual binary tree routing topology structured on IDs. 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 4 types of messages: *Ping*, *Store*, *Find_Node*, and *Find_Value*. *Ping* checks node liveliness, *Store* instructs a peer to store a value. *Find_Node* is a peer request. The response to a *Find_Node* query is a k-sized bucket of peers with the closest XOR distance to the requested ID. *Find_Value* operates identically to *Find_Node* with the exception that the peer storing a resource with the requested ID is returned.

Resources are stored by peers with the lowest XOR distance between their IDs. To find peers and resources iterative requests are issued to the closest known peer. These queries have the property that they are resolved within $O(log(|peers|))$ where $|peers|$ is the 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. To track the property we introduced our own counter with 2 lines of code for the number of *Find_Node* and *Find_Value* messages propagated in the cluster. Taipei-Torrent has sparse communication between nodes; the result of this is a large lattice of partial orderings. Lattices built from the traces of Taipei-Torrent consisted of 20–100 million points. Constructing and processing a lattice of that size took upwards of 15 minutes, with an upper limit of 2 hours, requiring frequent writes to disk as the structure exceeded available memory. Dinv succeeded in analyzing these executions, although the communication pattern was a challenge for our techniques. Lattice inflation due to Kademlia's communication pattern limited our analysis to executions with at most 7 peers. In all cases we were able to establish the logarithmic routing bound.

Kademlia specifies that peers must store and serve resources with the minimum XOR distance between their ID's. Further, *Find_Value* requests should 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 in a routing table, and a resource and logged it in a *Dump* statement. To test the routing we ran clusters with 3–6 peers using a variety of topologies by controlling peer IDs. hashed We logged state immediately after the results of a *Find_Value* request were added to a peer's routing table. On each execution we found that $\forall$ peers i, j, $min\_distance_i == min\_distance_j$. This invariant in conjunction with the O(Log(n)) message bound provides strong evedience for the correctness of Nikuku's impelmentation of Kademlia.

6.4 Checking Groupcache

Groupcache [12] is an open source distributed cache-filing library written in 1786 LOC and used by dl.google.com. Groupcache nodes are not run as separate services but integrated into Go programs; and a node acts as both a client and a server. Like memcached, Groupcache assigns key ownership to nodes, but nodes hold no state apart from multiple caches. All values must be computable by every node, i.e., fetchable from a shared database. Cluster membership changes must be handled by the user and result in changes to key ownership. A key's value is only computed once by the node that is responsible for that key. Subsequent gets from other nodes result in fetching the pre-computed value from the owner.

The network communication in a cluster of multiple nodes which request the same set of keys consists of equally distributed request-response exchanges. The number of keys nodes request should be similar, given a load-balancing hash function (caching causes slight deviations).

Integrating Groupcache into a codebase requires the developer to provide a *Getter* function to map keys to values. Messages are encoded using Google's Protocol Buffers and exchanged over HTTP. Dinv's automatic vector clock instrumentation was not applicable since it does not support Protocol Buffers. We augmented the HTTP header with vector clocks, which requires 6 additional LOC.

To show the central key distribution property (see Ta-

ble 2) we added a *track* statement in the *Getter* function to capture the requested key. The same key calculated on multiple hosts, possibly because of a faulty hash function or an implementation error, would directly invalidate this invariant.

The test program was run on 2–8 nodes, each of which requested the same set of up to 2K keys. By using the *whole cut merge* strategy, the invariant was inferred for all pairs of nodes. This hints at the correctness of Groupcache's key loading and work splitting process, which is central to the system.

Logging annotations	Annotations executed	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 3: Runtime performance of logging code.

7 Evaluation: Dinv overhead

Instrumentation introduces runtime execution overhead. Dinv adds network overhead by adding vector clocks to network payloads, and adds runtime overhead with logging code. Dinv performs static analysis during instrumentation time, and dynamic analysis on the logs produced during execution. The performance of instrumentation is a product of the number of logging annotations and the size of the source code, while the performance of log analysis depends on the pattern of communication during execution. This section details these overheads.

Static analyses. To benchmark the performance of Dinv's static analyses (adding logging code and detecting/wrapping networking calls) we used etcd Raft, which contains 288K LOC. We added `dump` statements and networking calls to Raft's source code and ran our instrumentation with progressively larger counts of instrumentation points. Annotations and vector clock instrumentations were run separately. Runtimes are reported in Figure 6 and show a nearly constant time of a few seconds. Dinv's static analysis is fast.

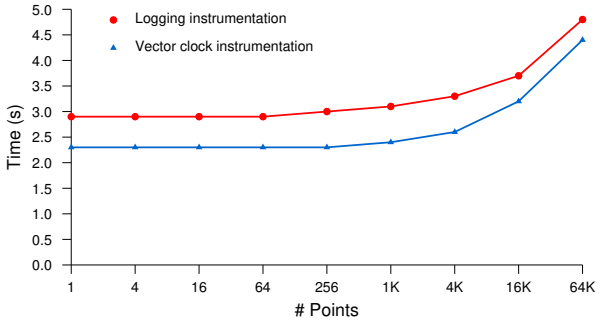

Figure 6: Time required to instrument Etcd Raft, for logging state and vector clocks.

Logging overhead. Logging state at runtime slows down the system, possibly perturbing its execution. We instrumented etcd Raft with increasing counts of logging statements each one logging 7 variables. We benchmarked a cluster with 3 nodes, and invoked 64 *get* and 64 *put* requests. Each cluster was run 3 times and the total running time was averaged. Table 3 shows a linear correlation between additional logging statemens and runtime. 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 execute 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 nodes interacting during execution and the number of messages: $64bits * nodes * messages$. In practice adding vector clocks to messages slows down each message, which can impact the behaviour of the system. To measure bandwidth overhead in a real system we executed etcd Raft using the setup described above. In the experiment we incremented the number of nodes. The bandwidths of all nodes was aggregated together for these measurements. Adding vector clocks to Raft slowed down the broadcast of heartbeat messages and actually caused a *reduction* in bandwidth of 10KB for all nodes in a 4 node cluster. As the number of nodes in the system grew, the size of the vector clocks overcame the bandwidth saving. Figure 7 depicts the difference in bandwidth overhead between instrumented and un-instrumented Raft averaged over 3 runs.

Dynamic analysis. The runtime of Dinv's dynamic analysis is affected by the size of the logs being analyzed and the number of nodes in the execution. To measure how our performance was affected by the length of execution we analyzed etcd Raft, and Groupcache. We exercise both systems by issuing 10 requests per second to each system. To demonstrate how Dinv's analysis per-

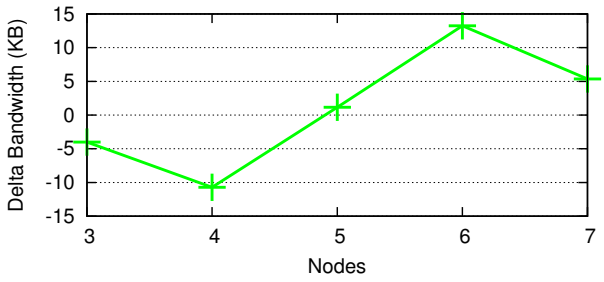

Figure 7: Difference in bandwidths between instrumented and un-instrumented etcd Raft.

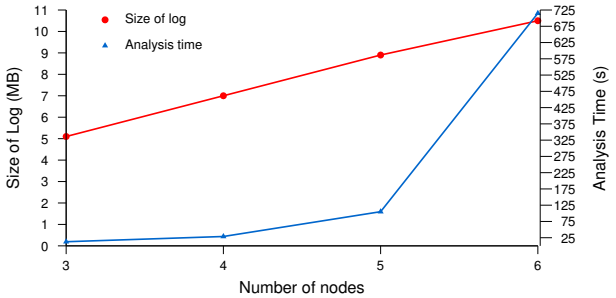

Figure 8: Dinv's dynamic analysis runtime vs the number of nodes in etcd Raft

System runtime (s)	Log size Raft (MB)	Analysis time Raft (s)	Log size GCache (MB)	Analysis time GCache (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 4: Dinv's dynamic analysis runtime vs the runtime of etcd Raft and GroupCache (GCache).

forms with regard to the length of execution, we analyzed the logs of 3 node clusters, which were run for incremental intervals of 30s. Results in Table 4 show that our analysis techniques scale linearly with the runtime of a system. 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 incremental node counts. Figure 8 shows an exponential relationship between the number of nodes and Dinv's analysis runtime. Dinv's runtime is exponential in the number of nodes due to the exponential growth of partial orderings our analysis techniques compute.

8 Discussion

Using Dinv. Dinv infers invariants that are *likely* because it is a dynamic analysis approach that only considers some finite set of system behaviors. The inferred invariants are not a verification of the system. But, Dinv is pragmatic when considering today's software development practices that include widely used dynamic analyses like testing. Unlike testing Dinv helps developers understand what happened. Dinv can be extended towards testing by having developers assert expected Dinv-inferred invariants for a set of curated executions.

To use Dinv a developer must have access to the system's source code, and they must be familiar with the system. 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. We have not carried out a formal usability study on Dinv, but we do have some anecdotal evidence that it is not difficult to use: students with no prior distributed systems background successfully used it in a graduate distributed systems course.

Depending on design of the system distributed state may be difficult to identify and encode, particularly when this requires reasoning about program paths that lead to network actions. 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.

Merging strategies. Dinv's three merging strategies (Section 3.3) were motivated by our evaluation experience. These cover different views on the systems and allow Dinv to detect a wide range of invariants, from properties true across all hosts during the entire execution, to eventual consistency properties observed on the level of send-receive pairs. Even so, these strategies are not exhaustive. All three merging strategies were implemented in less than 15 LOC; we therefore think that custom merging strategies offer a reasonable way to extend Dinv.

Daikon templates for distributed system invariants. Daikon is designed to infer invariants for program points in a sequential system. Dinv uses Daikon by presenting it with a synthetic program point that actually corresponds to a distributed state. This approach has its limits. In particular, Daikon supports a fixed set of invariant templates

that at most describe 3-ary relations. For an invariant mentioned earlier, $coordinator.commit = replica_i.commit$ for each replica i, Dinv mines as many individual invariants as there are replicas in the system. We experimented with adding support for n-ary properties to Daikon and see promising results. These properties would help to compactly describe properties of larger systems, like those with many nodes.

Analyzing executions containing failures. A major challenge for distributed systems is dealing with failures. In our experiments with the SWIM protocol in Serf we have shown that Dinv can establish properties of a system during network partitions. However, in general, Dinv is limited in the kinds of system failures it can be used to study. Dinv's decision to split the execution into strongly consistent cuts and translating the happens-before graph into a lattice structure assumes a relatively stable execution environment. For example, a run with a network partition that is not resolved before the end of execution, may not result in any cuts from the point where the partitioned occurred. Supporting other failure cases may require changing Dinv's analyses; a nuanced, but feasible project.

9 Related work

Distributed system state has a long history [3, 35, 14]. Prior work uses concrete state of a system to check the system against known properties [42, 21]. To be more immediately useful to developers who are attempting to understand complex systems, we think that distributed state requires abstraction. Dinv is one way to achieve this abstraction.

Mining distributed systems information. At its core DInv is a tool to mine information about a distributed system. Other work in this domain detects dependencies [25], temporal properties [2], anomalies [40], and performance bugs [34]. However, this prior work focuses on events, and not state. The closest prior work in the sequential domain is Daikon [10], which cannot be applied to distributed systems.

Yabandeh et al. [41] infer almost-invariants in distributed systems: invariants that are true in most cases and assume these invariants only are violated due to bugs. They require the user to provide a list of variables and functions for invariant inference. DInv can infer distributed state variables automatically. They also assume that an external module generates a trace of globally consistent cuts with distributed state for their algorithm; our approach actually instruments the system and generates these consistent cuts.

Other analysis of distributed systems. Dynamic analysis of distributed systems has yielded a number of tools to aid developers. Googles Dapper [37] analyzes traces of distributed systems to produce call graphs and report performance information. Another profiling tool, *lprof*, developed by Zhao et al. [43] automatically instruments Java bytecode by injecting information method information into existing log statements. Based on the generated execution logs lprof builds a call graph and infers temporal properties about the log. Lprof's instrumentation relies on synchronized timestamps rather then vector clocks, which limits its applicability to distributed systems.

Monitoring systems such as Fay [9] and Pivot tracing [26] use dynamic instrumentation for real-time diagnosis of distributed systems by activating trace points at runtime. These tools do not infer properties, such as invariants, that are present in the traces they capture.

Formal methods for distributed systems. Unlike recent methods that use theorem proving to synthesize correct systems by construction [39, 16], our work is immediately applicable to existing production systems. Previous work also considers checking existing system implementations directly [42, 21], or checking system properties at runtime or during program replay [32, 14, 24, 20]. This 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.

10 Conclusion

Distributed state is a key element of distributed systems that impacts consistency, performance, reliability, and other system features. However, in a large implementation distributed state is difficult to tease out, understand, and check. In this work we proposed an approach to help developers understand and check the validity of distributed state in their systems by inferring likely distributed state invariants. We realized our approach in Dinv, a static and dynamic program analysis tool for systems written in Go. We applied Dinv to four large systems and demonstrated that it can be used to validate useful properties in these systems. Dinv is an open source tool.

A Formal definitions

A distributed system can be described by n hosts, indexed from 1 to n.

Definition 1 (Host event types) *For a host i, the host event types set $E_i \supseteq \{START_i, END_i\}$ is a finite set (alphabet) of event types that can be generated by i.*

Definition 2 (System event types) *The set E of all possible system event types is $\cup E_i$, for all hosts i.*

Definition 3 (Event instance) *An event instance is the triple $t = \langle e, i, k \rangle$, where $e \in E_i$, i is a host index, and k is a non-negative integer that indicates the order of the event instance, among all event instances on host i.*

A host trace (Definition 4) is the set of all event instances generated at host i. This includes event instances of type $START_i$, and END_i, which respectively start and end the host trace.

Throughout, we use the notation e_i to represent an event type on host i, that is $e \in E_i$, and we use the notation $\hat{e_i}$ to represent the corresponding event instance $\langle e, i, k \rangle \in L$.

Event instances are ordered in two ways. First, the host ordering (Definition 5) orders any two event instances at the same host. Second, the interaction ordering (Definition 6) orders dependent event instances at different hosts. For example event instances of a host sending a message, and another receiving it are ordered. Both orderings are partial orderings, otherwise the distributed system can be more simply modelled as a serial execution.

Definition 4 (Host trace) *For all hosts i a host trace is a set T_i of all event instances $t = \langle e, i, k \rangle$, such that an event of type e was the k^{th} event generated on host i. The host trace T_i includes two event instances $\langle START, i, 0 \rangle$ and $\langle END, i, n \rangle$, such that n is the largest k for all $t \in T_i$. The k induces a total ordering of event instances in T_i We denote this total ordering as $<_i$. More formally, $\forall \hat{e_1} = \langle e_1, i, k_1 \rangle \in T_i, \hat{e_2} = \langle e_2, i, k_2 \rangle \in T_i, (\hat{e_1} <_i \hat{e_2} \iff k_1 < k_2)$.*

Definition 5 (Host ordering) *A host ordering $\prec_{host}$ is the union $\cup <_i$.*

Definition 6 (Interaction ordering) *An interaction ordering $\prec_{interact}$ partially orders pairs of event instances $\langle e_1, i, k_1 \rangle$ and $\langle e_2, j, k_2 \rangle$ such that $i \neq j$ (i.e., instances are at different hosts).*

A system trace is the union of a set of host traces (one per host), which corresponds to a single execution of the system. This union respects the host ordering and the interaction ordering.

Definition 7 (System trace) *A system trace is the pair $S = \langle T, \prec \rangle$, where $T = \cup T_i$, and $\prec = \prec_{host} \cup \prec_{interact}$.*

Definition 8 (Log) *A log L is a set of system traces.*

A common way of ordering event instances in a system trace is to associate a vector timestamp with each event instance. These timestamps make the partial order, $\prec$, of event instances in the system trace explicit.

The state of a host (Definition 9) is determined by the contents of its variables at an instant in time. Distributed systems the lack a unified clock, which leads to the inconsistent event orderings between hosts. Global snapshots [23] provide a mechanism for recording the state of individual hosts at a consistent point in time. Snapshots capture the state of a system by saving both the state of all hosts and the state of all messages being passed through communication channels. Implementing the snapshot algorithm requires the modification of the hosts underlying message passing system to capture transient messages. A less invasive method is to determine points during execution when the state of the system is entirely resident within the hosts. Such points occur when there are no messages in transit. The set of such points form a *consistent cut* (Definition 13), and represent a cohesive snapshot of the network.

Definition 9 (Host state) *A host state σ is the set of variable values at an instance during the execution of a host. The state of host i with m variable values, is described by an m tuple $\sigma_i = (v_0, v_1, \ldots, v_{m-1})$.*

Definition 10 (Execution) *An execution is an alternating sequence of events and states. We say that on a given host state σ_1 is the result of event instance $\hat{e_1}$. More formally $\forall \hat{e_i} < \hat{e_j}, \exists \sigma_i, \hat{e_i} < \sigma_i < \hat{e_j}$*

Definition 11 (Distributed state) *The distributed state is an n tuple of host states $\Sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$*

Our goal is to establish a temporal relationship between host states. We do this by defining a *cut* (Definition 12) to be a set of event instances spanning all hosts. We further define a *consistent cut* (Definition 13) to be a cut in which all interacting event instances over all hosts are contained within the cut. More plainly a cut is consistent if each event instance can be exactly ordered with at least one other event instance, and no subset of events is disconnected from any other.

Definition 12 (Cut) *A cut S is a set of event instances of size n such that for each host i, $\exists \hat{e_i} \in S$.*

Definition 13 (Consistent Cut) *Let L be a log and S be a cut for L. We say that S is consistent if $\forall \hat{e_i}, \hat{e_j} \in L$, if $\hat{e_i} \prec_{interact} \hat{e_j}$ and $\hat{e_i} \in S$, then $\hat{e_j} \in S$.*

Within a consistent cut, subsets of one or more event instances can be totally ordered. We consider the state of hosts at such events to be directly related, and define this subset of events to be a *distributed program point* (Definition 14). This stands in contrast to events which can only be partially ordered, or occur concurrently.

Definition 14 (Distributed Program Point) *Let S be a consistent cut, then a distributed program point p is defined as. $p \subseteq S$, where $\hat{e_i} \notin p \iff \exists \hat{e_j} \in p, (\hat{e_i} \not\prec \hat{e_j} \wedge \hat{e_j} \not\prec \hat{e_i})$*

Finally we define *consistent distributed state* (Definition 15) to be the set of host states in each distributed program point within a consistent cut.

A consistent distributed state can be described concisely as the set of host states in which all associated event instances are totally ordered. We can now use this totally ordered state information to infer invariants within the distributed system.

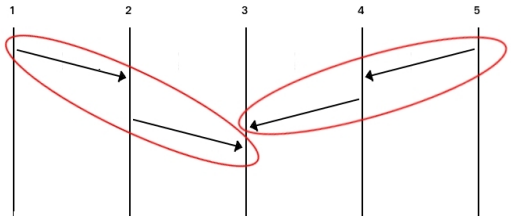

Figure 9: distributed program points in totally ordered subsets

Definition 15 (Consistent Distributed State) *Let p_σ, be the set of host states in a distributed program point p. Where S is a consistent cut, we say Σ is consistent if. $\forall p, \in S, p_\sigma \subseteq \Sigma$.*

B Ordering events with vector time

Vector time [11, 27] is a logical clock mechanism that provides a partial ordering of event instances. In a distributed system of h hosts, each host maintains an array of clocks $C = [c_0, c_1, \ldots, c_{h-1}]$, in which a clock value c_j records the local host's knowledge of (equivalently, dependence on) the local logical time at host j. We denote a timestamp's C clock value for host j as $C[j]$.

The hosts update their clocks to reflect the actual ordering of event instances in the system with the following three rules:

1. All hosts start with an initial vector clock value of $[0, \ldots, 0]$.
2. When a host i generates an event instance, it increments its own clock value (at index i) by 1, i.e $C_i[i] + +$.
3. When a host h communicates with a host h', h shares it's current clock C_h with h', and h' updates its local clock $C_{h'}$ so that $\forall i, C_{h'}[i] = \max\{C_h[i], C_{h'}[i]\}$. The host h' also updates its local clock value as in (2), since message receipt is considered an event.

C Logging node state

A program slice, is a subset of statements in a program, which either affect or are affected by some root statement. Forward slicing computes the set of statements affected by the root. Backwards slicing computes the set of statements which affect the root. Dinv extends Ottenstein's algorithm [30] for building a program dependence graph (PDG) with interprocedural analysis and builds a system dependence graph [38]. The roots of the slices in Dinv are network function calls. Dinv generates the set of variables to log by computing backwards slices from network writes and forward slices from network reads. The general algorithm for source code instrumentation is described in Algorithm 3.

Logging *network interacting variables* after each executed instruction would still produce an unmanageably large log. Dinv resolves this by logging at just select program points. Dinv includes two methods for identifying logging locations: a general-purpose automated approach and an approach based on developed-supplied annotations (Table 1).

Automated dump statement injection follows a general heuristic. The source code of the program is scanned, and annotations are injected at the entry and exit point of each function in the program. Semi-automated logging statements require the addition of annotations by either a developer, or by automated injection. Once a program has been annotated, Dinv analyzes the source code using program slicing, and replaces the annotations with logging

```
Data: Source code with dump statements S
Result: Instrumented Source Code I
I = S
dumpNode = CollectDumpAnnotationASTNodes(S)
for dump ∈ dumpNodes do
    variables = ComputeAffected(dump.line,S)
    loggingCode = NewLoggingCode(variables,
    dump.line, S)
    I += loggingCode
end
return Delta
```

Algorithm 3: Algorithm converting commented logging annotation to logging code.

code. Algorithm 3 details this process at a high level. Manual logging statements are specified explicitly by developers. They specify a list of variables as arguments, and log them. This offers fine-grained control over the instrumentation. In cases where many variables are in scope and interact with the network the majority of detected invariants are irrelevant to a desired system property and obfuscate Dinv's output. We preferable when verifying invariants on small sets of variables.

C.1 Logging functions

One factor which complicates logging variables is their scope. Conventionally variables are logged at a specific point in a programs execution, this has the advantage of capturing values at a specific moment in time, but also isolates them from being analyzed with variables which are present in another scope. To address situations where the variables should be logged at a specific point, and situations where sets of variables in different scopes should be logged together, Dinv has two distinct logging functions.

Dump writes the values of variables directly to a log when it is executed. Each unique dump statement is later merged as an independent entity, the result is invariants which reflect system invariants at arbitrary points.

Track writes variables to a key-value store and delays writing these to a log until the local vector time is incremented. By deferring and aggregating state a more complete view of a node's state can be analyzed together, at the cost of precision.

D Constructing the lattice

Algorithm 4 presents the lattice construction algorithm. Each level of the lattice is constructed sequentially. All clocks from the previous level have their values incremented by 1 for each node. If the incremented clock value

```
Data: Set of Host traces T
Result: Lattice representation of a System Trace S
i = 0 ;
Level[i].append(ZeroClock());
while !Level[i].empty() do
    i++;
    for Clock ∈ Level[i-1] do
        for Host ∈ T do
            Clock.increment(Host);
            if ValidLatticePoint(T,Clock) then
                Level[i].append(Clock)
            end
        end
    end
    S.append(Level[i])
end
```

Algorithm 4: Lattice Construction Algorithm

is valid under the happens-before relation, it is appended to the next level of the lattice.

E Constructing strongly consistent cuts

Enumerating Messages: A cut is consistent if and only if all messages sent by nodes in a cut, were also received by a node in the cut. The lattice constructed in the previous section contains the set of all possible cuts, some of which are inconsistent. If a cut is consistent the number of in flight messages at the corresponding vector time is zero. Checking this property can be done by iterating through the execution logs, and counting sending and receiving events. Because the lattice maintains the happens-before relation, If the number of sent and received messages are equal, a cut at that time is consistent. Algorithm 5 counts the number of send and receive events on each node by maintaining a delta of sent and received messages.

Extracting Cuts: The next step in the log merging process it to process the lattice by removing all points which do not correspond to consistent cuts. This is done by counting all sent and received messages at a system at each node in the lattice. The +1 per for a send, and -1 per a receive are stored as a *delta* calculated in the previous process. Figures 3A shows enumerated send and receive values in red, and Figures 3B highlights the corresponding points of the lattice which represent consistent cuts. if the sum of the *delta* for each node in a cut within the lattice is 0 the cut is consistent. This calculation run on each point in the lattice produces the complete set of consistent cuts. An abstract version of this process can be

```
Data: Set of Host traces T
Result: Enumerated index of Send and Receive Events
        Delta
for node ∈ T do
    for event ∈ node do
        if (event.WasASend()) then
            receiver, receiverEvent :=
            logs.FindReceive(event)
            Delta[node][event]++
            Delta[receiver][receiverEvent]--
        end
    end
end
return Delta
```

Algorithm 5: Sent and Received message enumeration

found in Algorithm 6.

```
Data: lattice, clocks, delta
Result: cuts
for level ∈ lattice do
    for node ∈ level do
        delta := 0; if node ∈ clocks then
            for node,clockValue ∈ node do
                delta +=
                deltaComm[node][clockValue]
            end
        end
        if delta == 0 then
            cuts.append(node)
        end
    end
end
return cuts
```

Algorithm 6: Algorithm for determining which lattice points correspond to a consistent cut

Collecting Hosts States: Next we collect the set of node states at the logical times they occurred in each consistent cut. This computation is done by searching the execution logs for node states with matching vector clock values. One challenge is the lack of a one-to-one correspondence between node states, and vector clock values. This is the consequence of either having *Dump* statements log a nodes state multiple times before incrementing its clock value, or having no dump statement executed at the given time. In such situations we choose the logged state based on the last networking operation performed by the node. In the case where the last operation was a receive, the node state corresponding to the earliest event instance is selected. Conversely if the last operation was to send a message, the node state corresponding to the last event instance is selected. If no node state is available it is not merged. In Section C.1 we introduced *Track* logging functions to mitigate this complication. *Track* which logs variables to a key value store, and only writes to a log during increments of vector time. Logging with a key value store has the advantage of aggregating node state and providing a one to one correspondence between vector time and and logged state, but has the disadvantage of missing multiple intermediate state transitions during a single vector time. The output of this process is the complete set of node states at every point during execution when all state was resident in memory.

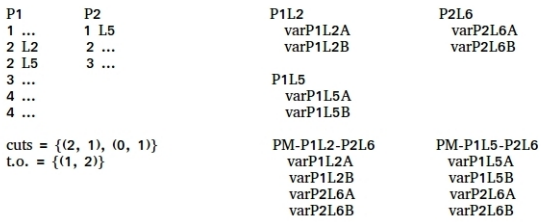

Figure 10: Retrieve consistent node states from logs

```
Data: Cuts, Log
Result: Set of consistent node states S
for Cut ∈ Cuts do
    for Host,Clock ∈ Cut do
        s += getHostStateFromLog(Host,Clock,Log)
    end
    S.append(s)
end
return S
```

Algorithm 7: Collect the Host's state, for each consistent cut

F Daikon background

Daikon is a dynamic analysis tool which automatically detects likely data invariants in sequential systems. Daikon derives data invariants by processing data traces (*dtraces*) produced during the run time of a program. A trace consists of a set of variable names, and the various values they took over the course of a programs execution. To produce a system trace Daikon injects logging statements at the beginning and end of both functions and loops, which dump the values of all variables in scope. Daikon uses a confidence measure to filter our spurious invariants.

```
1  i := 0
2  j := 1
```

Table 5: Daikon Invariants

1. $k = i + j$
2. $j > i$
3. $j = i + 1$
4. $i < 5$

```
3  k := i + j
4  for i < 5 {
5      k = i + j
6      i++
7      j++
8  }
```

Listing 1: simple program with loop invariants

```
1  i := 0
2  j := 1
3  k := i + j
4  for i < 5 {
5      logToTrace(i,j,k)
6      k = i + j
7      i++
8      j++
9      logToTrace(i,j,k)
10 }
```

Listing 2: Code instrumented to produce trace

F.1 Translation Daikon invariants to first order logic

Daikon supports unary, binary, and ternary invariants out of the box, and does not support predicate logic. The result of this is that invariant properties that hold over sets of n variables are output in pieces. For example if 4 variables $X_0 \dots X_3$ were determined to always be equal Daikon's output would appear as $X_0 == X_1, X_0 == X_2, X_0 == X_3$. In this case the 4 invariants can be combined transitively to show equality among all of them. In such cases we present the invariant as a first order logical formula. For example in this case the invariant would be denoted $\forall i, j, X_i = X_j$.

In other cases an invariant property is that in a single case a property exists. As an example in Section 5 we espouse the strong leadership principle of raft, stating that only a leader is able to issue an append entries message, and should therefore have a log greater than or equal to each of the followers. In this case we logged host state in two separate functions, one for followers, and one for the leader. The Daikon output from the execution was a separate file for each host which was a leader during execution. Detailed in Table 6 is the set of invariants used to support the property of strong leadership in etcd raft. The table supposes a cluster was composed of 3 hosts A,B,C each of which became a leader at least once. From the output it can be inferred that If a host is in the state leader, it has a log larger than that of the followers, and the followers have logs of equal size. For conciseness we represent this invariant with the following formula $Leader - LogSize >= Followers - LogSize$,$\forall followers$.

Table 6: Raft Strong Leadership Invariant

Table 7: File A-Leader B-Follower C-Follower

1. $A - State == Leader$
2. $C - State == B - State$
3. $B - State == Follower$
4. $A - LogSize >= B - LogSize$
5. $A - LogSize >= C - LogSize$
6. $B - LogSize == C - LogSize$

Table 8: File A-Follower B-Leader C-Follower

1. $B - State == Leader$
2. $A - State == C - State$
3. $A - State == Follower$
4. $B - LogSize >= A - LogSize$
5. $B - LogSize >= C - LogSize$
6. $A - LogSize == C - LogSize$

Table 9: File A-Follower B-Follower C-Leader

1. $C - State == Leader$
2. $A - State == B - State$
3. $B - State == Follower$
4. $C - LogSize >= B - LogSize$
5. $C - LogSize >= A - LogSize$
6. $B - LogSize == A - LogSize$

References

[1] AlDanial. cloc: Count lines of code. `https://github.com/AlDanial/cloc`, 2016.

[2] 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.

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

[4] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '91, pages 167–174, New York, NY, USA, 1991. ACM.

[5] coreos. A distributed init system. https://github.com/coreos/fleet, 2013.

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

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

[8] A. Das, I. Gupta, and A. Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 303–312. IEEE, 2002.

[9] Ú. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, October 2011.

[10] 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.

[11] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[12] B. Fitzpatrick. groupcache is a caching and cache-filling library, intended as a replacement for memcached in many cases. https://github.com/golang/groupcache, 2014.

[13] V. K. Garg. Maximal antichain lattice algorithms for distributed computations.

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

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

[16] 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 *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

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

[18] R. A. Jeff Overbey. Go doctor - the golang refactoring engine. http://gorefactor.org/index.html, 2014.

[19] Y. Junqueira. Kademlia/mainline dht node in go. https://github.com/nictuku/dht, 2012.

[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

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

[22] Kubernetes. Production-grade container scheduling and management. http://kubernetes.io/, 2014.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[24] 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 *Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, 2008.

[25] 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.

[26] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, New York, NY, USA, 2015. ACM.

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

[28] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS*, 2002.

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

[30] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, Apr. 1984.

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

[32] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 9. USENIX Association, 2006.

[33] RunLim. Runlim. `http://fmv.jku.at/runlim/`, 2016.

[34] 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 *NSDI*, 2011.

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

[36] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing Faulty Executions of Distributed Systems. In *NSDI*, 2016.

[37] 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.

[38] N. Walkinshaw, M. Roper, M. Wood, and N. W. M. Roper. The java system dependence graph. In *In Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 5–5, 2003.

[39] 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 *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 357–368, New York, NY, USA, 2015. ACM.

[40] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *SOSP*, 2009.

[41] M. Yabandeh, A. Anand, M. Canini, and D. Kostic. Finding Almost-Invariants in Distributed Systems. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, pages 177–182, Washington, DC, USA, 2011. IEEE Computer Society.

[42] 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 *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[43] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 629–644, Berkeley, CA, USA, 2014. USENIX Association.