ABSTRACT

Failure is always an option; in large-scale data management systems, it is practically a certainty. Fault-tolerant protocols and components are notoriously difficult to implement and debug. Worse still, choosing existing fault-tolerance mechanisms and integrating them correctly into complex systems remains an art form, and programmers have few tools to assist them.

We propose a novel approach for discovering bugs in fault-tolerant data management systems: lineage-driven fault injection. A lineage-driven fault injector reasons backwards from correct system outcomes to determine whether failures in the execution could have prevented the outcome. We present MOLLY, a prototype of lineage-driven fault injection that exploits a novel combination of data lineage techniques from the database literature and state-of-the-art satisfiability testing. If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly, in many cases using an order of magnitude fewer executions than random fault injection. Otherwise, MOLLY certifies that the code is bug-free for that configuration.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—Distributed Databases

Keywords

fault-tolerance; verification; provenance

1. INTRODUCTION

Fault tolerance is a critical feature of modern data management systems, which are often distributed to accommodate massive data sizes [2, 12, 20, 24, 28, 55, 80]. Fault-tolerant protocols—many of which, including atomic commit [33, 75], leader election [31], process pairs [34] and data replication [4, 76, 79], were pioneered by the database research community—are experiencing a renaissance in the context of these modern architectures.

With so many mechanisms from which to choose, it is tempting to take a bottom-up approach to data management system design, enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a global property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [16, 37] and bugs [59, 49].

Top-down testing approaches—which perturb and observe the behavior of complex systems—are an attractive alternative to verification of individual components. Fault injection is the dominant top-down approach in the software engineering and dependability communities. With minimal programmer investment, fault injection can quickly identify shallow bugs caused by a small number of independent faults. Unfortunately, fault injection is poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of faults (e.g., a network partition followed by a crash failure). Approaches such as Chaos Monkey [1] explore faults randomly, and hence are unlikely to find rare error conditions caused by complex combinations of failures. Worse still, fault injection techniques—regardless of their search strategy—cannot effectively guarantee coverage of the space of possible failure scenarios. Frameworks such as FATE [56] use a combination of brute-force search and heuristics to guide the enumeration of faults; such heuristic search strategies can be effective at uncovering rare failure scenarios, but, like random search, they do little to cover the space of possible executions.

An ideal top-down solution for ensuring that distributed data management systems operate correctly under fault would enrich the fault injection methodology with the best features of formal component verification. In addition to identifying bugs, a principled fault injector should provide assurances. The analysis should be sound: any generated counterexamples should correspond to meaningful fault tolerance bugs. When possible, it should also be complete: when analysis completes without finding counterexamples for a particular input and execution bound, it should guarantee that no bugs exist for that configuration, even if the space of possible executions is enormous.

To achieve these goals, we propose a novel top-down strategy for discovering bugs in distributed data management systems: lineage-driven fault injection (LDFI). LDFI is inspired by the database literature notion of data lineage [17, 25, 34, 45, 62, 89], which allows it to directly connect system outcomes to the data and messages that led to them. LDFI uses data lineage to reason backwards (from effects to causes) about whether a given correct outcome could have
failed to occur due to some combination of faults. Rather than generating faults at random (or using application-specific heuristics), a lineage-driven fault injector chooses only those failures that could have affected a known good outcome, exercising fault-tolerance code at increasing levels of complexity. Injecting faults in this targeted way allows LDFI to provide completeness guarantees like those achievable with formal methods such as model checking\cite{OLLY,CS56}, which have typically been used to verify small protocols in a bottom-up fashion. When bugs are encountered, LDFI’s top-down approach provides—in addition to a counterexample trace—fine-grained data lineage visualizations to help programmers understand the root cause of the bad outcome and consider possible remediation strategies.

We present MOLLY, an implementation of LDFI. Like fault injection, MOLLY finds bugs in large-scale, complex distributed systems quickly, in many cases using an order of magnitude fewer executions than a random fault injector. Like formal methods, MOLLY finds all of the bugs that could be triggered by failures: when a MOLLY execution completes without counterexamples it certifies that no fault-tolerance bugs exist for a given configuration. MOLLY integrates naturally with root-cause debugging by converting counterexamples into data lineage visualizations. We use MOLLY to study a collection of fault-tolerant protocols from the database and distributed systems literature, including reliable broadcast and commit protocols, as well as models of modern systems such as the Kafka reliable message queue. MOLLY quickly identifies 7 critical bugs in 14 fault-tolerant systems; for the remaining 7 systems, it provides a guarantee that no invariant violations exist up to a bounded execution depth, an assurance that state-of-the-art fault injectors cannot provide.

1.1 Example: Kafka replication

To ground and motivate our work, we consider a recently-discovered bug in the replication protocol of the Kafka\cite{OLLY} distributed message queue. In Kafka 0.80 (Beta), a Zookeeper service\cite{Zoo}—a strongly consistent metadata store—maintains and publishes a list of up-to-date replicas (the “in-sync-replicas” list or ISR), one of which is chosen as the leader, to all clients and replicas. Clients forward write requests to the leader, which forwards them to all replicas in the ISR; when the leader has received acknowledgments from all replicas, it acknowledges the client.

If replication is implemented correctly (and assuming no Byzantine failures) a system with three replicas should be able to survive one (permanent) crash failure while ensuring a “stable write” invariant: acknowledged writes will be stably stored on a non-failed replica. Kingsbury\cite{49} demonstrates a vulnerability in the replication logic by witnessing an execution in which this invariant is violated despite the fact that only one server crashes.

In brief, the execution proceeds as follows: two nodes $b$ and $c$ from a replica set $\{a, b, c\}$ are partitioned away from the leader $a$ and the Zookeeper service; as a result, they are removed from the ISR. $a$ is now the leader and sole member of the quorum. It accepts a write, acknowledges the client without any dissemination, and then crashes. The acknowledged write is lost.

The durability bug—which seems quite obvious in this post-hoc analysis—illustrates how difficult it can be to reason about the complex interactions that arise via composition of systems and multiple failures. Both Zookeeper and primary/backup replication are individually correct software components, but multiple kinds and instances of failures (message loss failure followed by node failure) result in incorrect behavior in the composition of the components. This achievable with a protocol bug (the client receives an acknowledgment only when the write is durably stored on all replicas) as it is a dynamic misconfiguration of the replication protocol, caused by a (locally correct) view change propagated by the Zookeeper service. Kingsbury used his experience and intuition to predict and ultimately witness the bug. But is it possible to encode that kind of intuition into a general-purpose tool that can identify a wide variety of bugs in fault-tolerant programs and systems?

1.2 MOLLY, a lineage-driven fault injector

Given a description of the Kafka replication protocol, we might ask a question about forward executions: starting from an initial state, could some execution falsify the invariant? This question gives us very little direction about how to search for a counterexample to the invariant. Instead, LDFI works backwards from results, asking why a given write is stable in a particular execution trace. For example, a write initiated by (and acknowledged at) the client is stable because (among other reasons) the write was forwarded to a (correct) node $b$, which now stores the write in its log. It was forwarded to $b$ by the leader node $a$, because $b$ was in a’s ISR. $b$ was in the ISR because the Zookeeper service considered $b$ to be up and forwarded the updated view membership to $a$. Zookeeper believed $b$ to be up because $a$ received timely acknowledgment messages from $b$. Most of the preceding events happened due to deterministic steps in the protocol. However, certain events (namely communication) were uncertain: in a different execution, they might not have succeeded. These are precisely the events we should explore to find the execution of interest: due to a temporary partition that prevents timely acknowledgments from $b$ and $c$, they are removed from the ISR, and the rest is history.

LDFI takes the sequence of computational steps that led to a good outcome (the outcome’s lineage) and reasons about whether some combination of failures could have prevented all “support” for the outcome. If it finds such a combination, it has discovered a schedule of interest for fault injection: based on the known outcome lineage, under this combination of faults the program might fail to produce the outcome. However, in most fault-tolerant programs multiple independent computations produce the important outcomes; in an alternate execution with failures, a different computation might produce the good outcome in another way. As a result, LDFI alternates between identifying potential counterexamples using lineage and performing concrete executions to confirm whether they correspond to true counterexamples.

Figure 1 outlines the architecture of MOLLY, an implementation of LDFI. Given a distributed program and representative inputs, MOLLY performs a forward step, obtaining a outcome by performing a failure-free concrete evaluation of the program. The hazard analysis component then performs a backward step, extracting the lineage of the outcome and converting it into a CNF formula that is passed to a SAT solver. The SAT solutions—failures that could falsify all derivations of the good outcome—are transformed into program inputs for the next forward step. MOLLY continues to execute the program over these failure inputs until it either witnesses an invariant violation (producing a counterexample) or exhausts the potential counterexamples (hence guaranteeing that none exist for the given input and failure model). To enable counterexample-driven debugging, MOLLY presents traces of buggy executions to the user as a collection of visualizations including both a Lampart diagram\cite{55} capturing communication activity in the trace and lineage graphs detailing the data dependencies that produced intermediate node states.

The remainder of the paper is organized as follows. In Section 2 we describe the system model and the key abstractions underlying
2. System model

Even in the absence of faults, distributed protocols can be buggy in a variety of ways, including sensitivity to message reordering or wall-clock timing. To maintain debugging focus and improve the efficiency of LDFI, we want to specifically analyze the effect of real-world faults on program outcomes. In this section, we describe a system model—along with its key simplifying abstractions—that underlies our approach to verifying fault-tolerant protocols. While our simplifications set aside a number of potentially confounding factors, we will see in Section 5 that Molly is nevertheless able to identify critical bugs in a variety of both classical and current protocols. In Section 7, we reflect on the implications of our focused approach, and the class of protocols that can be effectively verified using our techniques.

2.1 Synchronous execution model

A general-purpose verifier must explore not only the nondeterministic faults that can occur in a distributed execution (such as message loss and crash failures), but also nondeterminism in ordering and timing with respect to delivery of messages and scheduling of processes. Verifying the resilience of a distributed or parallel program to reordering is a challenging research problem in its own right [5, 22, 30, 74]. In that vein of research, it is common to make a strong simplifying assumption: assume that all messages are eventually delivered, and systematically explore the reordering problem [78, 82]. We argue that for a large class of fault-tolerant protocols, we can discover or rule out many practically significant bugs using a dual assumption: assume that successfully delivered messages are received in a deterministic order, and systematically explore failures. To do this, we can simply evaluate an asynchronous distributed program in a synchronous simulation. Of course any such simplification forfeits completeness in the more general model: in our case, certain bugs that could arise in an asynchronous execution may go unnoticed. We trade this weaker guarantee (which nevertheless yields useful counterexamples) for a profound reduction in the number of executions that we need to consider, as we will see in Section 5.

We discuss caveats further in Section 7.

2.1.1 Failure specifications

LDFI simulates failures that are likely to occur in large-scale distributed systems, such as (permanent) crash failures of nodes, message loss and (temporary) network partitions [11, 27, 32]. Byzantine failures are not considered, nor are crash-recovery failures, which involve both a window of message loss and the loss of ephemeral state. Verifying recovery protocols by modeling crash-recovery failures is an avenue of future work.

To ensure that verification terminates, we bound the logical time over which the simulation occurs. The notion of logical time—often used in distributed systems theory but generally elusive in practice—is well-defined in our simulations due to the synchronous execution abstraction described above. The internal parameter EOT (for end-of-time) represents a fixed logical bound on executions; the simulation then explores executions having no more than EOT global transitions (rounds of message transmission and state-changing internal events).

If the verifier explored all possible patterns of message loss up to EOT, it would always succeed in finding counterexamples for any non-trivial distributed program by simply dropping all necessary messages. However, infinite, arbitrary patterns of loss are uncommon in real distributed systems [10, 11]. More common are periods of intermittent failure or total partition, which eventually resolve (and then occur again, and so on). LDFI incorporates a notion of failure quiescence, allowing programs to attempt to recover from periods of lost connectivity. In addition to the EOT parameter, a second internal parameter indicates the end of finite failures (EFF) in a run, or the logical time at which message loss ceases. Molly ensures that EFF < EOT to give the program time to recover from message losses. If EFF = 0, Molly does not explore message loss—this models a fail-stop environment, in which processes may only fail by crashing.

A failure specification (Fspec) consists of three integer parameters (EOT, EFF, Crashes): the first two are as above, and the third specifies the maximum number of crash failures that should be explored. For example, a failure specification of (6, 4, 1) indicates that executions of up to 6 global transitions should be explored, with message loss permitted only from times 1-4, and zero or one node crashes. A crashed node behaves in the obvious way, ceasing to send messages or make internal transitions. A set of failures is admissible if it respects the given Fspec: there are no message omissions after EFF and no more than Crash crash failures.

In normal operation, Molly sets the Fspec parameters automatically by performing a sweep. First, EFF is set to 0 and EOT is increased until nontrivially correct executions are produced. Then EFF is increased until either an invariant violation is produced— in which case EOT is again increased to permit the protocol to recover—or until EFF = EOT − 1, in which case both are increased by 1. This process continues until a user-supplied wall clock bound has elapsed, and Molly reports either the minimal parameter settings necessary to produce a counterexample, or (in the case of bug-free programs) the maximum parameter settings explored within the time bound. In some cases (e.g., validating

executions in which no messages are sent are typically vacuously correct with respect to invariants.)
protocols in a fail-stop model), users will choose to override the sweep and set the parameters manually.

2.2 Language

LDIF places certain requirements on the systems and languages under test. The backwards step requires clearly identified program outputs, along with fine-grained data lineage that captures details about the uncertain communication steps contributing to the outcome. The forward step requires that programs be executable with a runtime that supports interposition on the communication mechanism, to allow the simulator to drive the execution by controlling message loss and delivery timing. Languages like Erlang [9] and Akka [68] are attractive candidates because of their explicit communication, while aspect-oriented programming could be used with a language such as Java to facilitate both trace extraction and communication interposition. Backwards slicing techniques [66] could be used to recover fine-grained lineage from the executions of programs written in a functional language.

For the MOLLY prototype, we chose to use Dedalus [7], a declarative, rule-based, executable logic language based on Datalog. Datalog-based distributed programming languages have generated considerable interest both in the theoretical and practical [11,12] and systems [37,46,56] research communities; protocol implementations in these languages often resemble pseudocode specifications. Dedalus satisfies all of the requirements outlined above. Data lineage can be extracted from logic program executions via simple, well-understood program rewrites [50]. More importantly, Dedalus (and similar languages) abstract away the distinction between events, persistent state and communication (everything is just data and relationships among data elements) and make it simple to identify redundancy of computation and data in its various forms (as we will see in Section 3). A synchronous semantics for Dedalus, consistent with the synchronous execution assumption described in Section 2.1, was proposed by Interlandi et al. [42].

All state in Dedalus is captured in relations; computation is expressed via rules that describe how relations change over time. Dedalus programs are intended to be executed in a distributed fashion, such that relations are partitioned on their first attribute (their location specifier). Figure 2 shows a simple broadcast program written in Dedalus. Line 1 is a deductive rule, and has the same intuitive meaning as a Datalog rule: it says that if some 2-tuple exists in bcast, then it also exists in log. Lines 2-3 are inductive rules, describing a relation between a particular state and its successor state. Line 2 for example, says that if some 2-tuple exists in node at some time \( t \), then that tuple also exists in node at time \( t + 1 \) (and by induction, forever after). Both are local rules, describing computations that individual nodes can perform given their internal state and the current set of events or messages. By contrast, the rule on line 4 is a distributed rule, indicating an uncertain derivation across process boundaries. It expresses a simple multicast as a join between a stream of events (bcast) and the persistent relation node. Note that the conclusion of the rule—a log tuple—exists (assuming that a failure does not occur) at a different time (strictly later) than its premises, as well as at a different place (the address represented by the variable Node2).

2.3 Correctness properties

A program is fault-tolerant (with respect to a particular Fspec) if and only if its correctness assertions hold for all possible combinations of admissible failures. Distributed invariants are commonly expressed as implications of the form \( \text{pre} \rightarrow \text{post} \); an invariant violation is witnessed by an execution in which the precondition holds but the postcondition does not. For example, the agreement invariant for a consensus protocol states that if an agent decides a value, \( \text{then} \) all agents decide that value. The Kafka stable write invariant described in Section 1.1 states that if a write is acknowledged, then it exists on a correct (non-crashed) replica.

To support this pattern, MOLLY automatically defines two built-in meta-outcomes of programmer-defined arity, called \( \text{pre} \) and \( \text{post} \). MOLLY users may express correctness assertions by defining these special relations—representing abstract program outcomes—as “views” over program state. If meta-outcomes are not defined, all persistent relations are treated as outcomes. This is acceptable for some simple protocols, such as naive reliable delivery (as we will see in Section 3), for more complex protocols, meta-outcomes should be used in order to mask unnecessary details. For example, a delivery protocol that suppresses redundant retries via message acknowledgments requires a meta-outcome that masks the exact number of ACKs and exposes only the contents of the log relation. Similarly, a consensus protocol is intended to reach some decision, though the decision it reaches may be different under different failures, so its meta-outcome should abstract away the particular decision.

Figure 3 shows a meta-outcome for reliable delivery, capturing the basic agreement requirement: if a correct node delivers a message, then all correct nodes receive it. Line 1 defines a missing log entry as one that exists on some node but is absent from another. Line 2 defines the precondition: a log entry exists on a correct (non-crashed) node. If \( \text{pre} \) does not hold, then the execution is vacuously correct. Line 3 defines the postcondition: no node is missing the log entry. If \( \text{pre} \) holds but \( \text{post} \) does not, the invariant is violated and the lineage of these meta-outcomes can be presented to the user as a counterexample.

To run MOLLY, a user must provide a program along with concrete inputs, and indicate which relations define the program’s outcomes—by default, using \( \text{pre} \) and \( \text{post} \) as defined above. We now turn to an overview of how MOLLY automates the rest of the bug-finding process.

```prolog
1 log(Node, Pload) :- bcast(Node, Pload);
2 node(Node, Neighbor) :- node(Node, Neighbor);
3 log(Node, Pload) :- log(Node, Pload);
4 log(Node, Pload) :- bcast(Node1, Pload), node(Node, Neighbor);
5 log(Node1, Pload) :- bcast(Node1, Pload), node(Node1, Node2);
```

Figure 2: simple-deliv, a Dedalus program implementing a best-effort broadcast.

```prolog
1 missing_log(A, Pl) :- log(X, Pl), notin missing_log(_, Pl) ;
2 pre(X, Pl) :- log(X, Pl), notin crash(_, X, _) ;
3 post(X, Pl) :- log(X, Pl), notin missing_log(_, Pl) ;
```

Figure 3: A correctness specification for reliable broadcast. Correctness specifications define relations \( \text{pre} \) and \( \text{post} \); intuitively, invariants are always expressed as implications: \( \text{pre} \rightarrow \text{post} \). An execution is incorrect if \( \text{pre} \) holds but \( \text{post} \) does not, and is vacuously correct if \( \text{pre} \) does not hold. In reliable broadcast, the precondition states that a (correct) process has a log entry; the postcondition states that all correct processes have a log entry.
3. USING LINEAGE TO REASON ABOUT FAULT-TOLERANCE

What a faint-heart! We must work outward from the middle of the maze. We will start with something simple.
– Thomasinana, in Arcadia [77].

One of the difficult truths of large-scale services is that if something can go wrong, eventually it will. Hence a reliable fault tolerance solution needs to account for unlikely events. A useful lens for efficiently identifying events (likely or otherwise) that could cause trouble for a fault-tolerant program is to view protocol implementation as a game between a programmer and an adversary.

In this section, we describe the LDFI approach as a repeated game (a match) in which an adversary tries to “beat” a protocol developer under a given system model. Of course, the end goal is for the developer to harden their protocol until it “can’t lose”—at which point the final protocol can truly be called fault tolerant under the model.

We show that a winning strategy for both the adversary (played by MOLLY) and the programmer is to to use data lineage to reason about the redundancy of support (or lack thereof) for program outcomes.

To play a match, the programmer and the adversary must agree upon a correctness specification, inputs for the program and a failure model (for example, the adversary agrees to crash no more than one node, and to drop messages only up until some fixed time). In each round of the match the programmer submits a distributed program; the adversary, after running and observing the execution, is allowed to choose a collection of faults (from the agreed-upon set) to occur in the next execution. The program is run again under the faults, and both contestants observe the execution. If the correctness specification is violated, the programmer loses the round (but may play again); otherwise, the adversary is allowed to choose another set of failures and the round continues. If the adversary runs out of moves, the programmer wins the round.

3.1 A match: reliable broadcast protocols

For this match, the contestants agree upon reliable broadcast as the protocol to test, with the correctness specification shown in Figure 3. Input is a single record bcast(A, data}@1, along with a fully-connected node relation for the agents \{A, B, C\}—i.e., Node A attempts to broadcast the payload “data” to nodes B and C. The adversary agrees to inject message loss failures no later than logical time 2, and to crash at most one server (EFF=2, Crashes=1).

In Figures 4, 6 and 7, we represent the lineage of the outcomes (the final contents of log) for the failure-free execution of the program. (played by MOLLY) and the programmer is to to use data lineage to reason about the redundancy of support (or lack thereof) for program outcomes.

To play a match, the programmer and the adversary must agree upon a correctness specification, inputs for the program and a failure model (for example, the adversary agrees to crash no more than one node, and to drop messages only up until some fixed time). In each round of the match the programmer submits a distributed program; the adversary, after running and observing the execution, is allowed to choose a collection of faults (from the agreed-upon set) to occur in the next execution. The program is run again under the faults, and both contestants observe the execution. If the correctness specification is violated, the programmer loses the round (but may play again); otherwise, the adversary is allowed to choose another set of failures and the round continues. If the adversary runs out of moves, the programmer wins the round.

3.1.1 Round 1: naive broadcast

The programmer’s first move is to submit the naive broadcast program simple-deliv presented in Figure 2. Figure 4 shows the lineage of three outcomes (the contents of the log relation on all nodes at time 4) for the failure-free execution of the program.

The adversary can use this representation of outcome lineage to guide its choice of failures to inject. To falsify an outcome (say log(B, data)) the adversary can simply “cut” the dotted line— that is, drop the message \(A\) sent to \(B\) at time 1. In the next execution, the adversary injects this single failure and the property expressed in Figure 3 is violated. The adversary wins the first round.

3.1.2 Round 2: retrying broadcast

The adversary was defeated, but she too can learn something from the lineage graph and the counterexample in Figure 4b. The adversary won Round 1 easily because simple-deliv has no redundancy: the good outcome of interest is supported by a single message. A fault-tolerant program must provide redundant ways to achieve a good outcome; in the context of this game, one of those “ways” must be out of the reach of the adversary. The programmer makes an incremental improvement to simple-deliv by adding a rule for bcast that converts it from an ephemeral event true at one logical time to a persistent relation that drives retransmissions:

\[ \text{bcast}(N, P) @\text{next} := \text{bcast}(N, P); \]

Instead of making a single attempt to deliver each message, this program (henceforth called retry-deliv) makes an unbounded number of attempts. Intuitively, this alteration has made the reliable delivery protocol robust to a certain class of nondeterministic failures—namely, message omissions—by ensuring that messages exhibit redundancy in time.

Figure 4b shows the outcome lineage for an execution of retry-deliv. This time, while the adversary has more difficulty choosing a move, the winning strategy once again involves reasoning directly about outcome lineage. Since \(A\) makes an unbounded number of attempts, the adversary can simply “cut” the dotted line—thereby eliminating the need to send the log message to \(B\) and \(C\), no finite pattern of omissions can falsify either outcome. The weakness of the retry-deliv algorithm is its asymmetry: the responsibility for redundant behaviors falls on \(A\) alone—this is easy to see in Figure 4b, in which all transmissions originate at node \(A\). The adversary, perceiving this weakness, might first attempt to immediately crash \(A\). If it did so, this would result in a vacuous counterexample, since the delivery invariant is only violated if some but not all agents successfully delivery the message. In Section 3.3.3 we’ll see how MOLLY avoids exploring such vacuously correct executions. However, causing \(A\) to crash after a successful transmission to one node (say \(C\)) but not the other is sufficient to falsify one outcome (log(B, data)). Exploring this potential counterexample via a concrete execution reveals the “true” counterexample shown in Figure 5. The adversary wins again.

3.1.3 Round 3: redundant broadcast

Reviewing the counterexample and the lineage of the failure-free execution, the programmer can see how the adversary won. The problem is that retry-deliv exhibits redundancy in time but not in space. Each broadcast attempt is independent of the failure of other attempts, but dependent on the fact that \(A\) remains operational. She improves the protocol further by adding another line:

\[ \text{bcast}(N, P) @\text{next} := \text{log}(N, P); \]

Now every node that has a log entry assumes responsibility for broadcasting it to all other nodes. As Figure 6 reveals, the behavior of all nodes is now symmetrical, and the outcomes have redundant failures.

These are finite executions, so the number of attempts is actually bounded by the EOT (4 in these figures) in any given execution. However, since the adversary has agreed to not drop messages, the program is guaranteed to make more attempts than there are failures.
support both in space (every correct node relays messages) and time (every node makes an unbounded number of attempts). The adversary has no moves and forfeits; at last, the programmer has won a round.

3.1.4 Round 4: finite redundant broadcast

In all of the rounds so far, the adversary was able to either choose a winning move or decide it has no moves to make by considering a single concrete trace of a failure-free simulation. This is because the naive variants that the programmer supplied—which exhibit infinite behaviors—reveal all of their potential redundancy in the failure-free case. A practical delivery protocol should only continue broadcasting a message while its delivery status is unknown; a common strategy to avoid unnecessary retransmissions is to stop broadcasting a message while its delivery status is unknown. In Round 4, the programmer provides the protocol **ack-deliv** shown in Figure 5 in which each agent retries only until it receives an ACK.

The failure-free run of **ack-deliv** (Figure 5) exhibits redundancy in space (all sites relay) but not in time (each site relays a finite number of times and ceases before EOT when acknowledgments are received). The adversary perceives that it can falsify the outcome \( \log(B, \text{data}) \) by dropping the message \( A \) sent to \( B \) at time 1, and either the message \( A \) sent to \( C \) at time 1 or the message \( C \) sent to \( B \) at time 2 (symbolically, \( O(A,B,1) \land (O(A,C,1) \lor O(C,B,2)) \)). It chooses this set of failures to inject, but in the subsequent run the failures trigger additional broadcast attempts—which occur when ACKs are not received—and provide additional support for the outcome. The adversary gets as many chances as it likes, but each time at each round the adversary “cuts” some edges and injects the corresponding failures; in the subsequent run new edges appear. Eventually it gives up, when the agreed-upon failure model permits no more moves. The programmer wins again.

3.1.5 Round 5: “classic” broadcast

For the final round, the programmer submits a well-known reliable broadcast protocol originally specified Birman et al. [61]:

\[
(\text{At a site receiving message } m) \rightarrow 
\begin{align*}
&\text{if message } m \text{ has not been received already} \\
&\text{send a copy of } m \text{ to all other sites} \\
&\text{deliver } m
\end{align*}
\]

This protocol is correct in the fail-stop model, in which processes can fail by crashing but messages are not lost. The programmer has committed a common error: deploying a “correct” component in an environment that does not match the assumptions of its correctness argument.

The classic broadcast protocol exhibits redundancy in space but not time; in a failure-free execution, it has infinite behaviors like the protocols submitted in Rounds 2-3, but this redundancy is vulnerable to message loss. The adversary, observing the lineage graph in Figure 2, immediately finds a winning move: drop messages from \( A \) to \( B \) at time 1, and from \( C \) to both \( A \) and \( B \) at time 2.

3.2 Hazard analysis

In the game presented above, both players used the lineage of program outcomes to reason about their next best move. The role of the programmer required a certain amount of intuition: given the
Process B
Process A
Process C

Figure 5: Redundant broadcast with ACKs.

Figure 6: Lineage for the redundant broadcast protocol redun-deliv, which exhibits redundancy in space and time. A redundant support (falsifier: $O(A,C,1) \lor O(C,B,2))$ is highlighted. The lineage from this single failure-free execution is sufficient to show that no counterexamples exist.

Figure 7: Lineage for the finite redundant broadcast protocol ack-deliv, which exhibits redundancy in space in this execution, but only reveals redundancy in time in runs in which ACKs are not received (i.e., when failures occur). The message diagram shows a unsuccessful fault injection attempt based on analysis of the lineage diagram: drop messages from A to B at time 1, from C to B at time 2, and then crash C. When these faults are injected, ack-deliv causes A to make an additional broadcast attempt when an ACK is not received.

4. THE MOLLY SYSTEM

In this section, we describe how we built the MOLLY prototype using off-the-shelf components including a Datalog evaluator and a SAT solver.

4.1 Program rewrites

To simulate executions of a Dedalus program, we translate it into a Datalog program that can be executed by an off-the-shelf interpreter. To model the temporal semantics of Dedalus (specifically state update and nondeterministic failure) we rewrite all program rules to reference a special clock relation. We also add additional rules to record the lineage or data provenance of the program’s outputs.

4.1.1 Clock encoding and Dedalus rewrite

Due to the synchronous execution model and finiteness assumptions presented in Section 2, we can encode the set of transitions that occur in a particular execution—some of which may fail due to the synchronous execution model and finiteness assumptions presented in Section 2, we can encode the set of transitions that occur in a particular execution—some of which may fail
to omissions or crashes—in a finite relation, and use this relation to drive the execution. We define a relation clock with attributes \((\text{From}, \text{To}, \text{SndTime})\). To model local state transitions in inductive \((\text{@next})\) rules, we ensure that clock always contains a record \((n, n, t)\), for all nodes \(n\) and times \(t < \text{EOT}\). Executions in which no faults occur also have a record \((n, m, t)\) for all pairs of nodes \(n\) and \(m\). To capture the loss of a message from node \(a\) to node \(b\) at time \(t\) (according to \(a\)’s clock), we delete the record \((a, b, t)\) from clock. To be consistent with the EFF parameter, this deletion can only occur if \(t \leq \text{EFF}\). Finally, to model a fail-stop crash of a node \(a\) at time \(t\), we simply delete records \((a, b, u)\) for all nodes \(b\) and times \(u \geq t\).

We also rewrite Dedalus rules into Datalog rules. For all relations, we add a new integer attribute \(\text{Time}\) (as the last attribute), representing the logical time at which the record exists. To rewrite the premises of a rule, we modify each rule premise to reference the \(\text{Time}\) attribute and ensure that all premises have the same \(\text{Time}\) attribute and location specifier (this models the intended temporal semantics: an agent may make conclusions from knowledge only if that knowledge is available in the same place, at the same time). To rewrite the conclusion of a rule, we consider the Dedalus temporal annotations:

**Example 1** Deductive rules have no temporal annotations and their intended semantics match that of Datalog, so they are otherwise unchanged.

\[
\text{log}(\text{Node}, \text{Funload}) \leftarrow \text{bcast}(\text{Node}, \text{Funload});
\]

\[
\text{log}(\text{Node}, \text{Funload}, \text{Time}) \leftarrow \text{bcast}(\text{Node}, \text{Funload}, \text{Time});
\]

**Example 2** Inductive rules—which capture local state transitions—are rewritten to remove the \text{@next} annotation and to compute the value of \(\text{Time}\) for the rule’s conclusion by incrementing the \(\text{SndTime}\) attribute appearing in the premises.

\[
\text{node}(\text{Node}, \text{Neighbor}, \text{From}) \leftarrow \text{node}(\text{Node}, \text{Neighbor});
\]

\[
\text{node}(\text{Node}, \text{Neighbor}, \text{SndTime}+1) \leftarrow \text{node}(\text{Node}, \text{Neighbor}, \text{SndTime}), \text{clock}(\text{Node}, \text{Node}, \text{SndTime});
\]

**Example 3** Asynchronous rules—representing uncertain communication across process boundaries—are rewritten in the same way as inductive rules. Note however that because the values of \text{To} and \text{From} are distinct, the transition represented by a matching record in \text{clock} might be a failing one (i.e., it may not exist in \text{clock}).

\[
\text{log}(\text{Node2}, \text{Funload}, \text{async}) \leftarrow \text{bcast}(\text{Node1}, \text{Funload}), \text{node}(\text{Node1}, \text{Node2});
\]

\[
\text{log}(\text{Node2}, \text{Funload}, \text{SndTime}+1) \leftarrow \text{bcast}(\text{Node1}, \text{Funload}, \text{SndTime}), \text{node}(\text{Node1}, \text{Node2}), \text{clock}(\text{Node1}, \text{Node2}, \text{SndTime});
\]

### 4.1.2 Lineage rewrite

In order to interpret the output of a concrete run and reason about fault events that could have prevented it, we record per-record data lineage capturing the computations that produced each output.

We follow the provenance-enhanced rewrite described by Kohler et al [50]. For every rule, the rewrite produces a new “firings” relation that captures bindings used in the rule’s premises.

For every rule \(r\) in the given (rewritten from Dedalus as described above) Datalog program, we create a new relation \(r_{\text{prov}}\) (called a “firings” relation) and a new rule \(r’\), such that

1. \(r’\) has the same premises as \(r\),
2. \(r’\) has \(r_{\text{prov}}\) as its conclusion, and
3. \(r_{\text{prov}}\) captures the bindings of all premise variables.

For example, given the asynchronous rule in Example 3, MOlly synthesizes a new rule:

\[
\text{log}_{\text{prov}}(\text{Node1}, \text{Node2}, \text{Funload}, \text{SndTime}) \leftarrow \text{bcast}(\text{Node1}, \text{Funload}, \text{SndTime}),
\]

\[
\text{node}(\text{Node1}, \text{Node2}, \text{SndTime}), \text{clock}(\text{Node1}, \text{Node2}, \text{SndTime});
\]

Rules with aggregation use two provenance rules, one to record variable bindings and another to perform aggregation. This prevents the capture of additional bindings from affecting the grouping attributes of the aggregation. For example:

1. \(r(X, \text{count} < Z) \leftarrow a(X, Y), b(Y, Z)\)
2. \(r_{\text{prov}}(X, \text{count} < Z) \leftarrow r(X, Y), b(Y, Z)\)
3. \(r(X, \text{count} < Z) \leftarrow \text{bcast}(X, Y, Z)\)
4. \(r_{\text{prov}}(X, \text{count} < Z) \leftarrow r_{\text{prov}}(X, Y, Z)\)

### 4.2 Proof tree extraction

We query the firings relations to produce derivation graphs for records [81]. A derivation graph is a directed bipartite graph consisting of rule nodes that correspond to rule firings, and goal nodes that correspond to records. There is an edge from each goal node to every rule firing that derived that tuple, and an edge from each rule firing to the premises (goal nodes) used by that rule. The lineage graphs in Section 3 (Figures 3a-7) are abbreviated derivation graphs, in which goal nodes are represented but rule nodes are hidden.

To construct the derivation graph for a record \(r\), we query the firings relations for rules that derive \(r\). For each matching firing, we substitute the bindings recorded in the firing relation into the rule and recursively compute the derivation graphs for each of those premises. Note that each rule node represents a firing of a rule with a particular set of inputs; it is possible for a single outcome to have multiple derivations via the same rule, each involving different premises.

Each derivation graph yields a finite forest of proof trees. Each proof tree corresponds to a separate, independent support of the tree’s root goal (i.e., its outcome). Given a proof tree, we can determine which messages were used by the proof’s rule firings; the loss of any of these messages will falsify that particular proof.

### 4.3 Solving for counterexamples

Given a forest of proof trees, a naive approach to enumerating potential counterexamples is to consider all allowable crash failure and message omissions that affect messages used by proofs. This quickly becomes intractable, since the set of fault combinations grows exponentially with EFF.

Instead, we use the proof trees to perform a SAT-guided search for failure scenarios that falsify all known proofs of our goal tuples. For each goal, we construct a SAT problem whose variables encode crash failures and message omissions and whose solutions correspond to faults that falsify all derivations in the concrete execution.

Each proof tree is encoded as a disjunction of the message omissions \(O_{\text{from-to}}\) and crash failures \(C_{\text{node-time}}\) that can individually falsify the proof. By taking the conjunction of these formulas, we express that we want solutions that falsify all derivations. For example,

\[
(O(a,c) \lor C(a,2) \lor C(a,1)) \land (O(b,c) \lor C(b,1))
\]
corresponds to a derivation graph that represents two proofs, where the first proof can be falsified by either dropping messages from $a$ to $c$ at time 2 or by $a$ crashing at some earlier time, and the second proof can be falsified by $b$ crashing or the loss of its messages sent at time 1.

If the resulting SAT problem is unsatisfiable, then there exists at least one proof that cannot be falsified by any allowable combination of message losses and crash failures—hence the program is fault-tolerant with respect to that goal! Otherwise, each SAT solution represents a potential counterexample that must be explored.

We solve a separate SAT problem for each goal tuple, and the union of the SAT solutions is the set of potential counterexamples that we must test—each potential counterexample corresponds to a “move” of the adversary in the game presented in Section 3. If the user has defined correctness properties using the built-in pre and post meta-outcomes defined in Section 3 we perform an additional optimization. Each meta-outcome is handled as above, and a potential counterexample is reported for each set of faults that falsifies a record in post unless those faults also falsify the corresponding record in pre. We need not explore such faults, as they would result in a vacuously correct outcome with respect to that property.

The algorithm described above is presented formally in the Appendix (Section B), along with proofs of the soundness and completeness of LDPI.

5. EVALUATION

In this section, we use MOLLY to study a variety of fault-tolerant protocols from the database and distributed systems literature, as well as components of modern systems. We then measure the performance of MOLLY along two axes: its efficiency in discovering bugs, and its coverage of the combinatorial space of faults for bug-free programs.

5.1 Case Study: Fault-tolerant protocols

We implemented a collection of fault-tolerant distributed programs in Dedalus, including a family of reliable delivery and atomic commitment protocols and the Kafka replication subsystem described in Section 1.1. We analyze them with MOLLY and describe the outcomes.

MOLLY automatically produces Lamport diagrams [53], like those shown in Section 3 to help visualize the message-level behavior of individual concrete executions and to enable counterexample-driven debugging. In each diagram, solid vertical lines represent individual processes; time moves from top to bottom. Messages between processes are shown as diagonal lines connecting process lines; lost messages are shown as dashed lines. Vertices represent events and are numbered to reflect global logical time; if a process crashes, its node contains the string “CRASHED.” When it comes time to debug systems to discover (and ultimately remedy) the cause of an invariant violation, MOLLY produces lineage diagrams similar to those shown in Section 3.

5.1.1 Commit Protocols

We used Dedalus to implement three commit protocol variants from the database literature, which were developed and extended over a period of about five years [15][33][75]. As we would hope, MOLLY immediately confirmed the known limitations of these protocols, and produced concrete counterexamples both for non-terminating executions and for executions in which conflicting decisions are made.

For all commit protocols, we specify two invariants as implications between pre- and postconditions:

- **Agreement:** If an agent decides to commit (respectively, abort), then all agents decide to commit (abort).
- **Termination:** If a transaction is initiated, then all agents decide either commit or abort for that transaction.

Molly automatically identified the limitations of early commit protocols that subsequent work attempted to correct. Figure 8 illustrates the well-known blocking problem associated with two-phase commit (2PC) [33]. If the coordinator process fails after preparing a transaction, the system is left in a state in which the transaction outcome is unknown but all agents are holding locks waiting for the outcome (a violation of the termination property).

The collaborative termination protocol (CTP) [15] attempts to ameliorate the blocking problem by augmenting the 2PC protocol so as to allow agents who suspect that the coordinator has failed to exchange their knowledge about the outcome. It is well-known, however, that although CTP allows more executions to terminate, it has blocking executions under the same failure assumptions as classic 2PC. MOLLY discovered a counterexample after a single forward/backward execution—due to space limitations, the diagram is omitted.

Three-phase commit [75] solves the blocking problem—under the assumption of a connected and synchronous network—by adding an additional protocol round and corresponding agent state. It uses simple timeouts as a failure detector; depending on the state a coordinator or agent is in when a timeout fires, that site can unilaterally determine the transaction outcome. Hence there are no “blocking” states.

If we relax the assumption of a connected network by allowing finite message failures, however, MOLLY discovers bad executions such as the one shown in Figure 9. In this case, message losses from the coordinator to certain agents ($a$ and $b$) cause the agents to conclude that the coordinator has failed. Since they are in the canCommit state, they decide to roll forward to commit. Meanwhile the coordinator—which has detected that agent $d$ (who originally agreed to commit) has failed—has decided to abort. This outcome is arguably worse than blocking: due to the incorrectness of the failure detector under message omissions, agents have now made conflicting decisions, violating the agreement property.

As we saw in the case of classic-deliv in Section 3 the bad execution results from deploying a protocol in an environment that

![Figure 8: A blocking execution of 2PC. Agents $a$, $b$ and $d$ successfully prepare (p) and vote (v) to commit a transaction. The coordinator then fails, and agents are left uncertain about the outcome—a violation of the termination property.](image-url)
violates the assumptions of its correctness guarantee—an all-too-common hazard in composing systems from individually-verified components.

5.1.2 Other fault-tolerant protocols

We used MOLLY to study other agreement protocols, including Paxos [54] and the bully leader election protocol [31]. As we discuss in Section 7, desirable termination properties of such protocols are difficult to verify due to their sensitivity to asynchrony. Nevertheless we are able to validate their agreement properties by demonstrating that no counterexamples are found for reasonable parameters (as noted in Figure 13).

Flux [73] is a replica synchronization protocol for streaming dataflow systems based on the process pairs fault-tolerance strategy. Flux achieves fault-tolerance by ensuring that a pair of replicas receives the same message stream without loss, duplication or reordering; at any time, should one replica fail, the other can take over. Despite its succinct specification, Flux is considered to be significantly more complicated than alternative fault-tolerance strategies for streaming systems, because of the interaction between the protocol’s granularity (tuple-at-a-time) and the various combinations of failures that can occur during operation [87]. Using MOLLY, we were able to certify that Flux is resilient to omission and crash failures up to a significant depth of execution (see Figure 13). To the best of our knowledge, this effort represents the most thorough validation of the Flux protocol to date.

5.1.3 Kafka replication bug

To reproduce the Kafka replication bug described in Section 1.1, we provide a single durability invariant:

- Durability: If a write is acknowledged at the client, then it is stored on a correct (non-crashed) replica.

MOLLY easily identified the bug. Figure 10 shows the same counterexample described by Kingsbury. A brief network partition isolates two of the replicas (b and c) from a and the Zookeeper service, and a is elected as the new leader (assuming it is not the leader already). Because a believes itself to be the sole surviving replica, it does not wait for acknowledgments before acknowledging the client. Then a crashes, violating the durability requirement.

In reproducing this durability bug, we relied heavily on MOLLY’s ability to model different components of a large-scale system at different levels of specificity. We focused first on the primary/backup replication protocol logic, which we implemented in significant detail (roughly a dozen LOC in Dedalus). Based on the intuition that the bug lay at the boundary of the replication protocol and the Zookeeper service and not in the service itself, we sketched the Zookeeper component and the client logic, ignoring details such as physical distribution (we treat the Zookeeper cluster as a single abstract node) and the underlying atomic broadcast protocol [43]. Had this model failed to identify a bug, we could subsequently have enriched the sketched specifications.

5.2 Measurements

A lineage-driven fault injector must do two things efficiently: identify bugs or provide a bounded guarantee about their absence. In this section, we measure MOLLY’s efficiency in finding counterexamples for 7 buggy programs, compared to a random fault injection approach. We then measure how quickly it covers the combinatorial space of possible faults for bug-free programs.

Figure 12 lists the buggy protocols and (minimal) parameter settings for which MOLLY found a counterexample. The table lists the protocol size (in lines of code), the number of concrete executions explored before the counterexample was found, and the elapsed time. In order to factor apart the impact of the abstractions presented in Section 2 from that of the pruning performed by
hazard analysis, we also implemented a random fault injector for MOLLY. In random mode, MOLLY chooses failure combinations at random and avoids the overhead of SAT and lineage extraction. Figure 12 also shows the average number of executions (exe) and average execution time (wall, in seconds) before the random fault injector discovered a counterexample (averages are from 25 runs).

The performance of random fault injection reveals the significance of LDFI’s abstractions in reducing the search space: for all of the (buggy) protocols we studied, random fault injection eventually uncovered a counterexample. In the case of relatively shallow failure scenarios, the random approach performed competitively with the hazard analysis-based search. In more complex examples—in particular the Kafka bug—MOLLY’s hazard analysis outperforms random fault injection by an order of magnitude.

Even more compelling than its rapid discovery of counterexamples is MOLLY’s reduction of the search space for bug-free programs. A random strategy may find certain bugs quickly, but to guarantee that no bugs exist (given a particular set of parameters) requires exhaustive enumeration of the space of executions, which is exponential both in EFF and in the number of nodes in the simulation. By contrast, MOLLY’s hazard analysis is guaranteed to discover and explore only those failures that could have invalidated an outcome. Figure 13 compares the space of possible executions (Combinations) that would need to be explored by an exhaustive strategy to the number of concrete executions (exe) performed by MOLLY (providing 100% coverage of the relevant execution space), for a number of bug-free protocol implementations. In all cases, we report the maximum parameter values reached by the sweep procedure given a 120 second time bound.

Figure 11 plots the growth in the number of concrete executions considered as the EFF is increased, for the ack-deliv and redun-deliv protocols presented in Section 3, against the upper bound (the number of possible failure combinations for that Fspec) on a logarithmic scale. It illustrates the impact of redundancy in individual executions on the pruning strategy. By revealing massive redundancy in every run, redun-deliv protocol allows MOLLY to rule out an exponentially larger set of potential counterexamples in each backward step.

6. RELATED WORK

In this section, we compare LDFI to existing techniques for testing and verifying fault-tolerant distributed systems.

Model checking \cite{30,39,48,63,64,85,86} is a widely used technique for systematically checking distributed systems for violations of correctness properties. Model checkers can provide guarantees that no bad executions are possible by exhaustively checking all program states reachable from a set of initial states. Model checking is ideally suited to specifying and exhaustively testing individual components (particularly protocols) of distributed systems. For practical distributed systems—systems that run for long periods of time, and are built from a variety of components—this state space is often too large to exhaustively explore. Some attempts to manage this complexity include abstraction refinement \cite{13,23}, running model checkers concurrently with execution to detect possible invariant violations in the local neighborhood \cite{54}, and heuristic search ordering of the state space \cite{83}. LDFI sidesteps the state explosion problem by asking a simpler, more targeted question: given a class of good outcomes, can some combination of faults prevent them? The complexity of this problem depends on the depth of the lineage of the good outcomes and on the size of the global state space.

Fault injection frameworks \cite{1,26,36,44} interpose on the execution of distributed programs to explore the consequences of multiple failures on specific outcomes. Fault injection techniques typically use either a random \cite{1,59} or heuristic \cite{26,36} strategy to explore the space of possible failures. FATE and DESTINI \cite{50} has been used to reproduce dozens of known bugs in cloud software, as well as to discover new ones. Like MOLLY, it uses Datalog as a specification language; unlike MOLLY, it uses a combination of brute force and heuristic search to explore failure combinations. MOLLY takes a more complete approach, providing assurances that no bugs exist for particular configurations and execution bounds.

LDFI focuses specifically on the effects of faults on outcomes, and is compatible with a variety of other techniques that address orthogonal issues. At a high level, MOLLY’s alternating execution strategy resembles concolic execution \cite{72}, which similarly alternates between calls to a concrete evaluator and a symbolic solver. As we discuss in Section 7, concolic testing and other symbolic execution approaches (e.g. Klee \cite{18}) are ideal for discovering bad inputs, and hence are complementary to LDFI. When verifying individual components, LDFI can be used as a complementary approach to model checkers such as Chess \cite{64}, which focus strictly on nondeterminism in interleavings. Like test generation approaches such as execution synthesis \cite{88}, MOLLY “explains” bugs, albeit at a higher level of abstraction: via data and dependencies rather than stepwise program execution. Unlike execution synthesis, MOLLY does not require a priori knowledge of bugs, but discovers them.

Like reverse data management \cite{69}, LDFI uses provenance to reason about what changes to input relations would be necessary to produce a particular change in output relations (“how-to” queries).
When the submitted Dedalus program is logically monotonic \cite{5, 58}, LDFI answers how-to queries using positive why-provenance \cite{21}, otherwise LDFI must also consider the why-not provenance \cite{40} of negated rule premises (e.g., “an acknowledgment was not received”). Systems such as Artemis \cite{38} address the why-not problem (for monotonic queries) as a special case of what-if analysis, and ask what new (perhaps partially-specified) tuples would need to be added to a database to make a missing tuple appear. First-order games \cite{51, 69} use a game-theoretic execution strategy to answer why-not queries. Wu et al. \cite{83} describe a practical approach to answering why-not queries for software-defined networks (SDNs).

LDFI can be viewed as a narrow version of the “how-to” provenance problem, restricted to considering deletions on a single distinguished input relation (the clock), in the presence of possibly non-monotonic queries. Many of the why-not provenance techniques discussed above could assist in implementing LDFI—this is a promising avenue for future work.

7. DISCUSSION AND FUTURE WORK

To conclude our discussion, we reflect on some of the limitations of the MOLLY prototype, as well as directions for future work. Our narrow focus on the fault-tolerance of distributed systems allowed us to significantly simplify the verification task, but these simplifying abstractions come with tradeoffs.

It is clearly impractical to exhaustively explore all possible inputs to a distributed system, as they are unbounded in general. We have assumed for the purposes of this discussion that the program inputs—including the execution topology—are given a priori, either by a human or by a testing framework. However, our approach is compatible with a wide variety of techniques for exploring system inputs, including software unit testing, symbolic execution \cite{19, 22} and input generation \cite{6, 23}.

LDFI assumes that the distributed protocols under test are “internally deterministic” (i.e., deterministic modulo the nondeterminism introduced by the environment). It leverages this assumption—common in many fault-tolerant system designs \cite{71}—to provide its completeness guarantee: if some execution produces a proof tree of an outcome, any subsequent execution with the same faults will also. While MOLLY can be used to find bugs in fundamentally non-deterministic protocols like anti-entropy \cite{67} or randomized consensus \cite{14}, certifying such protocols as bug-free will require additional research.

The pseudo-synchronous abstraction presented in Section \ref{sec:synchronous}—which made it possible to discover complex bugs by rapidly exploring multiple heterogeneous failures—does come at a cost. For an important class of fault-tolerant distributed algorithms (e.g., those that attempt to solve consensus), an abstraction that factors apart partial failure and asynchrony is fundamentally incomplete, because these algorithms are required to (attempt to) distinguish between delay and failure \cite{19, 29}. For example, when we verify an algorithm like Paxos \cite{54} (described in Section \ref{sec:algorithms}), the conclusion that Paxos is tolerant to a particular set of failures does not imply that Paxos terminates in all executions. Relaxing the synchronicity abstraction is an avenue of future work, but Section \ref{sec:algorithms} provides evidence that the tradeoff is worthwhile.

MOLLY automates the role of the adversary in the game presented in Section \ref{sec:games}. But what about the role of the programmer? In future work, it would be interesting to explore using the backwards reasoning approach of LDFI to assist in fault-tolerant program synthesis. Given a distributed program with a fault-tolerance bug, it seems possible to use the lineage of its failure-free run (along with a counterexample) to effectively guide the search through program transformations that provide additional redundant support of the program outcome. Similar techniques should also facilitate adapting existing fault-tolerant algorithms (like the classic-deliv protocol) to new failure assumptions.

8. CONCLUSION

Fault tolerance code is hard to test in a controlled environment, yet likely to fail catastrophically at scale unless it is fully debugged. Ad hoc approaches like random fault injection are easy to integrate with real-world code but unable to provide bullet-proof assurances. LDFI presents a middle ground between pragmatism and formalism, dictated by the importance of verifying fault tolerance in spite of the complexity of the space of faults. LDFI works with executable code, though it requires that code to be written in a language that meets the requirements outlined in Section \ref{sec:language}.

By walking this middle ground, LDFI and MOLLY offer significant benefits over the state of the art in three dimensions. First, LDFI provides radical improvements in the efficiency of fault injection by narrowing down the choice of relevant faults to inject. Second, LDFI enables MOLLY to provide useful software engineering tools, illustrating tricky fault-tolerance bugs with concrete traces complete with auto-generated visualizations of root causes (lineage diagrams) and communication visualizations (Lamport diagrams). Finally, LDFI makes it possible to formally “bless” code as being correct up to a significant depth of execution, something that is infeasible with traditional fault injection techniques.

<table>
<thead>
<tr>
<th>Program</th>
<th>Counterexample</th>
<th>LOC</th>
<th>EOT</th>
<th>EFF</th>
<th>Crashes</th>
<th>Combinations</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple-deliv</td>
<td>Figure 1a</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>$4.10 \times 10^{14}$</td>
<td>exe 4.08</td>
</tr>
<tr>
<td>retry-deliv</td>
<td>Figure 1b</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>$4.07 \times 10^{14}$</td>
<td>exe 75.24</td>
</tr>
<tr>
<td>classic-deliv</td>
<td>Figure 1c</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>$2.62 \times 10^{14}$</td>
<td>exe 116.16</td>
</tr>
<tr>
<td>2pc</td>
<td>Figure 8a</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>5.48</td>
</tr>
<tr>
<td>2pc-ctp</td>
<td>Figure 8b</td>
<td>25</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>$56 \times 10^{12}$</td>
<td>8.56</td>
</tr>
<tr>
<td>3pc</td>
<td>Figure 8c</td>
<td>24</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>$2.43 \times 10^{12}$</td>
<td>40.60</td>
</tr>
<tr>
<td>Kafka</td>
<td>Figure 10</td>
<td>18</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>$1.85 \times 10^{12}$</td>
<td>1183.12</td>
</tr>
</tbody>
</table>

Figure 12: MOLLY finds counterexamples quickly for buggy programs. For each verification task, we show the minimal parameter settings (EOT, EFF and Crashes) to produce a counterexample, alongside the number of possible combinations of failures for those parameters (Combinations), the number of concrete program executions MOLLY performed (exe) and the time elapsed in seconds (wall). We also measure the performance of random fault injection (Random), showing the average for each measurement over 25 runs.
References


state-of-the-art fault injection techniques: if a program's correct-
mented in a widely-used language: Section 2.2 discusses alterna-
tigation is achieving the benefits of LDFI for source code imple-
tools that are compatible with LDFI. A remaining area for inves-
generation and testing of interleavings—are provided by existing
identify bugs in large-scale distributed systems. It is worthwhile
A. ALTERNATIVE APPROACHES
APPENDIX
A. ALTERNATIVE APPROACHES
Figure 11 places LDFI in the feature space of existing tools that
identify bugs in large-scale distributed systems. It is worthwhile
to note that two of the key features not provided by LDFI—input
generation and testing of interleavings—are provided by existing
tools that are compatible with LDFI. A remaining area for inves-
tigation is achieving the benefits of LDFI for source code imple-
mented in a widely-used language: Section 2.2 discusses alterna-
tive approaches.
B. COMPLETENESS
LDFI provides a completeness guarantee unattainable by other
state-of-the-art fault injection techniques: if a program's correct-
ness specification can be violated by some combination of failures
(given a particular execution bound), LDFI discovers that combi-
nation; otherwise it reports that none exists. In this section, we
provide a formalization and proof for that claim.

Algorithm 1 LDFI
Require: \( P \) is a Datalog\(^+\) program produced by rewriting a Dedalus pro-
gram
Require: \( \mathcal{E} \) is an EDB including clock facts
Require: \( g \in \mathcal{P}(\mathcal{E}) \) is a goal fact
1: function LDFI(\( P, \mathcal{E}, g \))
2: \( R \leftarrow \) provenance-enhanced rewrite of \( P \)
3: \( \mathcal{E} \leftarrow \text{RGO}(\mathcal{R}, \mathcal{E}, g) \)
4: \( g \leftarrow \text{clocks}(g, \mathcal{F}) \)
5: if \( g \) is satisfiable then
6: while there are more satisfying models of \( g \) do
7: \( \mathcal{A} \leftarrow \) the next model of \( g \)
8: \( D \leftarrow \{\text{clock}(f, t, l) \in \mathcal{E}|\mathcal{A} \models O_{f,t,l} \lor (\mathcal{A} \models C_{f,t,l} \land \mathcal{F} < l)\} \)
9: if \( g \not\in \mathcal{P}(\mathcal{E} \setminus D) \) then
10: Yield \( D \)
11: end if
12: end while
13: else
14: return \( \emptyset \)
15: end if
16: end function

Section 4.1.1 describes how a (distributed) Dedalus program can
be rewritten into a fragment of Datalog\(^-\), whose execution may be
simulated by an off-the-shelf Datalog evaluator. Given two Data-
log relations \( p \) and \( q \), we write \( p \rightarrow q \) if and only if there exists a
rule \( r \) in \( P \) such that \( p \) is the relation of a subgoal in \( r \) and \( q \) is the
relation in the head (we omit the \( P \) when the context is clear). We
write \( \rightarrow \) to denote the transitive closure of \( \rightarrow \). We assume that
submitted Dedalus programs are stratifiable [81]; that is, no predic-
dates depend negatively on themselves, directly or transitively. It is
easy to see that the rewrite procedure produces stratified Datalog\(--\)
programs. In particular, the only change that rewriting makes is to
introduce the clock relation on the right-hand-side of some rules.
Since clock never appears on the left-hand-side of a rule, the rewrite
does not introduce any new transitive cyclic dependencies.

A fact is a predicate symbol all of whose arguments are con-
stants; e.g., \( \log(A, \text{“data”}) \). We write relation(f) to indicate the predicate name of a fact \( f \). For example, if \( f = \log(A, \text{“data”}) \) then relation(f) = log. We are specifi-
cally interested in the set of EDB facts \( \mathcal{C} \subseteq \{c \in \mathcal{E} | \text{relation}(c) = \text{clock}\} \) Each such fact \( c \in \mathcal{C} \) (henceforth called clock facts) is of the
form \( \text{clock}(f, \text{from}, \text{to}, \text{time}) \) and intuitively represents connectivity from computing node \( \text{from} \) to node \( \text{to} \) at time \( \text{time} \) on \( \text{from} \)'s local clock. Recall that in the Dedalus to Datalog\(--\) rewrite, every “asynchronous” rule is rewritten to include clock
as a positive subgoal. For convenience we use a named field nota-
tion (as in SQL): given a clock fact \( e \) we write \( e.\text{from} \) to indicate the
value in the first column of \( c \); similarly with \( e.\text{to} \) and \( e.\text{time} \) (the second and third columns, respectively).

Given a stratifiable Datalog\(--\) program \( P \) and an external extension (EDB) \( \mathcal{E} \) (a set of base facts comprising the program’s input), we write \( \mathcal{P}(\mathcal{E}) \) to denote the (unique) minimal model of \( P \) over \( \mathcal{E} \). The model \( \mathcal{P}(\mathcal{E}) \supseteq \mathcal{E} \) is itself a set of facts. Lineage analysis oper-
ates over a derivation graph [50,81], a bipartite rule/goal graph \( \mathcal{G} = (R \cup G, E) \), where \( G \) is a set of goal facts and \( R \) is a set of rule firings [50]. An edge \((x, y) \in \mathcal{E} \) associates either
1. a goal \( x \) with a rule \( y \) used to derive it, or
2. a rule firing \( x \) with a subgoal \( y \) that provided bindings.
We write \( \text{RGO}(\mathcal{P}, \mathcal{E}) \) to represent the rule/goal graph produced by
executing program \( P \) over input \( \mathcal{E} \).
LDFTI constructs boolean formulae and passes them to a SAT solver. Given a model \( \mathcal{M} \) returned by a solver and a formula \( \varphi \), we write \( \mathcal{M} \models \varphi \) if \( \varphi \) is true in \( \mathcal{M} \). We are concerned with the truth values of a set of propositional variables \( \{ O_{from\_to\_time} \cup C_{from\_to\_time} \} \) such that from_to are drawn from the domain of locations (the first attribute of every fact, and in particular the clock relation) and time consists of integers less than \( EOT \). Recall that \( O_{from\_to\_time} \) represents message loss from node from to node to at time time, while \( C_{from\_to\_time} \) represents a permanent crash failure of node from at time time.

We model failures in distributed systems as deletions from the EDB clock relation. By construction, such deletions affect precisely the (transitive) consequences of \( async \) Dedalus rules, and match intuition: message loss is modeled by individual deletions (a loss of connectivity between two endpoints at a particular logical time), while crash failures are modeled by batch deletions (loss of connectivity between some endpoint and all others, from a particular time onwards).

**Definition** A fault set is a set of facts \( D \equiv \{ f \in \mathcal{E} | relation(f) = clock \} \).

Given a program \( P \), an EDB \( \mathcal{E} \) and a distinguished set of “goal” relation names \( G \) (in the common case, \( G \equiv \{ post \} \)), we identify a set of goal facts \( \mathcal{F} = \{ g \in P(\mathcal{E}) | relation(g) \in G \} \). For each goal fact \( g \in \mathcal{F} \), we wish to know whether there exists a fault set which, if removed from \( \mathcal{E} \), prevents \( P \) from producing \( g \).

**Definition** A falsifier of a goal fact \( g \) is a fault set \( D \) such that \( g \in P(\mathcal{E}) \), but \( g \notin P(\mathcal{E} \setminus D) \). A falsifier is minimal if there does not exist a falsifier \( D' \) of \( g \) such that \( D' \subset D \).

LDFTI identifies potential falsifiers of program goals by inspecting the data lineage of concrete executions. We may view a lineage-driven fault injector as a function from a program, an EDB and a goal fact to a set of fault sets; we write \( LDFTI(P, \mathcal{E}, g) \) to denote the (possibly empty) set of fault sets that LDFTI has determined could falsify a goal fact \( g \) produced by applying the Datalog program \( P \) to the EDB \( \mathcal{E} \).

A lineage-driven fault injector is sound if, when \( D \in LDFTI(\mathcal{E}, \mathcal{E}, g) \), \( D \) is indeed a falsifier of \( g \). Soundness is trivially obtained by the forward/backward execution strategy: for any potential counterexamples, LDFTI performs a concrete execution (Algorithm 1, Line 7) to determine if the set of omissions constitutes a correctness violation, and outputs \( D \) only if \( g \notin P(\mathcal{E} \setminus D) \).

To prove completeness, we present the LDFTI system discussed in Section 4 formally in Algorithm 1. Most of the work of LDFTI is performed by the recursive function \( clocks \) defined in Algorithm 2, which operates over derivation graphs and returns a boolean formula whose satisfying models represent potential counterexamples. Given a node \( n \) in the graph \( \mathcal{G} \) (either a rule node or a goal node, as described above), \( clocks \) returns a formula whose satisfying models intuitively represent faults that could prevent \( n \) from being derived. If \( n \) is a clock fact (Line 4 of Algorithm 2), then \( clocks \) returns a disjunction of boolean variables representing conditions (losses and crashes) that could remove this fact from the EDB; if \( n \) is a non-clock leaf fact, \( clocks \) simply returns true (Line 7). Otherwise, \( n \) is either a rule node or a non-leaf goal. If \( n \) is a rule node, then all of its child (goal) nodes were required to cause the rule to fire; invalidating any of them falsifies the rule—hence to invalidate \( n \), we take the conjunction of the formulae that invalidate its children (Line 24). By contrast, if \( n \) is a non-leaf positive goal, then each of its (rule) children represents an alternative derivation of \( n \) via different rules—to invalidate \( n \), we must invalidate all of its alternative derivations: hence we consider the conjunction of the formulae that invalidate its children (Line 29). The last case to consider is if \( n \) is a negative goal: that is, some rule fired because (among other reasons) a particular fact did not exist (e.g., a retry was triggered because a timeout fired and there was no log of an acknowledgment message). The derivation graph \( RGG(P, \mathcal{E}) \) does not explicitly represent the reasons why a particular tuple does not exist. There are a variety of options for exploring why-not provenance, as we discuss in the related work. The MOLLY prototype currently offers three alternatives to users. The first is to ignore the provenance of negated goals—this is clearly acceptable for monotonic programs, and can be useful for quickly identifying bugs, but is incomplete. The second is similar to the approach used by Wu et al. [85] to debug software-defined networks, which uses surrogate tuples to stand in for facts that do not hold at a particular time and location. This approach seems to work well in practice. Finally, we support an optimized version of the conservative approach described in detail in the completeness proof below: we consider as possible causes of a negated goal tuple \( g \) any tuples in the model \( P(\mathcal{E}) \) from which \( relation(g) \) is reachable via an odd number of negations (based on static analysis of the program), and \( z \) whose timestamp is less than or equal to the timestamp of \( g \).

For the purposes of the proof we consider a conservative overapproximation of the set of possible causes for the nonexistence of a fact. Line 7 enumerates the set of (positive) facts \( z \) such that \( relation(n) \) is reachable from \( relation(z) \). \( clocks \) then invokes itself recursively and returns the disjunction of the falsifying formulae for all such goals \( r \). The intuition is that since we do not know the exact reason why a fact does not exist, we overapproximate the set of possible causes by considering any fact \( z \) that could reach \( n \) (based on a static analysis of the dependency relation \( \rightarrow \)) as one which, if made false, could cause \( n \) to appear and falsify a derivation that required \( n \) to be absent. The attentive reader will observe that falsifying \( z \) can only make \( n \) true if \( n \) depends negatively on \( z \)—therefore we could further constrain the set

<table>
<thead>
<tr>
<th>LDFTI</th>
<th>Model checking</th>
<th>Test generation</th>
<th>Fault injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Molly</td>
<td>MOLLY [65]</td>
<td>SPIN [86]</td>
<td>DESTINI [56]</td>
</tr>
<tr>
<td>MoDis</td>
<td>TLAe [60]</td>
<td>CrystalBall</td>
<td>TLA+</td>
</tr>
<tr>
<td>Chess</td>
<td>[62]</td>
<td>[29]</td>
<td>[84]</td>
</tr>
<tr>
<td>CrystalBall</td>
<td>[62]</td>
<td>[65]</td>
<td>[29]</td>
</tr>
<tr>
<td>MaceMC</td>
<td>[59]</td>
<td>[59]</td>
<td>[29]</td>
</tr>
<tr>
<td>SPIN</td>
<td>quick-check [22]</td>
<td>Execution synthesis [59]</td>
<td>Symbolic execution [14][22]</td>
</tr>
</tbody>
</table>
of facts enumerated in Line 17 of Algorithm 2 to include only those
z from which n is reachable via an odd number of negations. Be-
cause LDFI is sound, it is always safe to over-approximate the set
of possible falsifiers, so for simplicity of presentation we omit this
optimization from the proof.

We first establish a lemma regarding the behavior of Algorithm 2.

**Lemma B.1.** Given a program \( P \), EDB \( E \), their derivation graph \( G = \text{RGG}(P, E) \), and a goal fact \( g \in \mathcal{P}(E) \), if \( D \) is a mini-
mal falsifier of \( g \), then there exists a model \( \mathfrak{A} \) of the boolean
formula \( \text{clocks}(g, \mathcal{G}) \) such that for every \( f \in D \), either \( \mathfrak{A} \models (O_{\text{from}, f, to, ttime}) \) or \( \mathfrak{A} \models (C_{\text{from}, t}) \) for some \( t \) such that \( t \leq f.\text{time} \).

**Proof.** Proof is by induction on the structure of \( G \).

**Base case:** \( g \) is a leaf-goal. We assume the antecedent: \( D \) is a
minimal falsifier of \( g \). Consider any \( f \in D \). Because \( D \) is mini-
mal, it must be the case that without \( f \), \( g \) cannot be derived by
\( P \) over \( E \). So if \( g \) is a leaf goal, it must be the case that \( g = f \). So \( \text{clocks}(g, \mathcal{G}) = (O_{\text{from}, f, to, ttime}) \) and its satisfying models are exactly those that make true either
\((O_{\text{from}, f, to, ttime}) \) or \((C_{\text{from}, t}) \) for some \( t \) such that \( t \leq f.\text{time} \).

**Inductive case 1:** \( g \) is a rule. By the inductive hypothesis, Lemma B.1 holds for all subformulae \( \text{clocks}(g, \mathcal{G}) \) such that
\((g, g') \in E \). Line 24 returns the disjunction of the subformulæ;
since Lemma B.1 for each, it surely also holds for their disjunction
(any model of one of the subformulae is a model of the whole
disjunction).

**Inductive case 2:** \( g \) is a non-leaf goal. We consider two cases:
If \( g \) is positive, consider all subformulae \( \text{clocks}(r, \mathcal{G}) \) such that
\((g, r) \in E \)—call those subformulae \( \varphi, \psi, \ldots \). By the in-
ductive hypothesis, there exist models \( \mathfrak{A}, \mathfrak{B}, \ldots \) such that \( \mathfrak{A} \models \varphi, \mathfrak{B} \models \psi, \ldots \) and for all \( f \in D \), either \( \mathfrak{A} \models (O_{\text{from}, f, to, ttime}) \) or \( \mathfrak{A} \models (C_{\text{from}, t}) \) for some \( t \) such that \( t \leq f.\text{time} \), and similarly
for \( \mathfrak{B}, \ldots \), etc.

We must now show that there necessarily exists a model \( \mathfrak{S} \)
such that \( \mathfrak{S} \models \varphi \land \mathfrak{S} \models \psi \land \ldots \) and for all \( f \in D \), either \( \mathfrak{S} \models (O_{\text{from}, f, to, ttime}) \) or \( \mathfrak{S} \models (C_{\text{from}, t}) \).

By construction, \( \varphi, \psi, \ldots \) contain only the boolean connectives and \((\land)\) and or \((\lor)\); in particular, they do not contain negation. Hence it cannot be the case that their
conjunctive is unsatisfiable. We construct \( \mathfrak{S} \) by making true every propositional variable that is true in any of the \( \mathfrak{A}, \mathfrak{B}, \ldots \). Observe that \( \mathfrak{S} \models \varphi \), since all variables true in \( \mathfrak{S} \) are true in \( \varphi \) and \( \varphi \) does not
contain negation—similarly for \( \psi, \ldots \). Hence \( \mathfrak{S} \models \varphi \land \psi \land \ldots \).

Finally, if \( g \) is negative, then \( C \) (Line 17) is the enumeration of the
z \( \in \mathcal{G} \) such that \( g \) is (statically) reachable from \( z \) based
on \( \rightarrow_z \). Note that if \( g \) has only positive support (no predicate
\( z' \) such that \( z' \rightarrow_z g \) appears as a negative subgoal in a rule),
then no falsifiers of “not \( g \)” exist—EDB deletions cannot cause new
facts to appear except in the presence of negation—and so the
lemma holds vacuously. However, it is possible that \( g \) depends
negatively (specifically, via an odd number of negations) on some

\[ \text{Algorithm 2 Clocks algorithm} \]

**Require:** \( \mathcal{G} = (R \cup G, E) \) is a bipartite rule-goal graph.

**Require:** \( n \in (R \cup G) \)

\begin{verbatim}
1: function \text{clocks}(n, \mathcal{G})
2: if \( n \in G \) then \( \triangleright n \) is a goal
3: if \( \exists \exists (r \in R \land (n, r) \in E) \) then \( \triangleright n \) is a leaf
4: if \( \text{relation}(n) = \text{clock} \) then
5: if \( n.\text{time} < \text{EFF} \) then
6: \( \varphi \leftarrow \text{new bool: } O_{n, \text{from}, n, \text{to}, n, \text{time}} \)
7: else
8: \( \varphi \leftarrow \text{false} \)
9: end if
10: \( \psi \leftarrow \bigvee_{i=0}^{t} \text{new bool: } C_{n, \text{from}, i} \)
11: end if
12: \( \varphi \land \psi \)
13: end if
14: return \( \triangleright \) Ignore non-clock leaves
15: end if
16: if \( n \) is negative then \( \triangleright n \) was a negated goal
17: \( C \leftarrow \{z \in \text{relation}(z)^{+} \land \text{relation}(n)\} \)
18: return \( \bigvee_{z \in C} \text{clocks}(z, \mathcal{G}) \)
19: end if
20: return \( \bigwedge_{(n, r) \in E} \text{clocks}(r, \mathcal{G}) \)
21: end if
22: end if
23: end if
24: return \( \bigvee_{(n, r) \in E} \text{clocks}(g, \mathcal{G}) \)
25: end if
26: end function
\end{verbatim}