SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi*,
Jeffrey F. Lukman† and Haryadi S. Gunawi

University of Chicago *NEC Labs America †Surya University

Abstract

The last five years have seen a rise of implementation-level distributed system model checkers (dmck) for verifying the reliability of real distributed systems. Existing dmcks however rarely exercise multiple failures due to the state-space explosion problem, and thus do not address present reliability challenges of cloud systems in dealing with complex failures. To scale dmck, we introduce semantic-aware model checking (SAMC), a white-box principle that takes simple semantic information of the target system and incorporates that knowledge into state-space reduction policies. We present four novel reduction policies: local-message independence (LMI), crash-message independence (CMI), crash recovery symmetry (CRS), and reboot synchronization symmetry (RSS), which collectively alleviate redundant reorderings of messages, crashes, and reboots. SAMC is systematic; it does not use randomness or bug-specific knowledge. SAMC is simple; users write protocol-specific rules in few lines of code. SAMC is powerful; it can find deep bugs one to three orders of magnitude faster compared to state-of-the-art techniques.

1 Introduction

As more data and computation move from local to cloud settings, cloud systems[1] such as scale-out storage systems [7, 13, 18, 41], computing frameworks [12, 40], synchronization services [5, 28], and cluster management services [27, 47] have become a dominant backbone for many modern applications. Client-side software is getting thinner and more heavily relies on the capability, reliability, and availability of cloud systems. Unfortunately, such large-scale distributed systems remain difficult to get right. Guaranteeing reliability has proven to be challenging in these systems [23, 25, 51].

Software (implementation-level) model checking is one powerful method of verifying systems reliability [21, 52, 53]. The last five years have seen a rise of software model checkers targeted for distributed systems [22, 25, 43, 50, 51]; for brevity, we categorize such systems as *dmck* (distributed system model checker). Dmck works by exercising all possible sequences of events (*e.g.*, different reorderings of messages), and hereby pushing the target system into corner-case situations and unearthing hard-to-find bugs. To address the state-space explosion problem, existing dmcks adopt advanced state reduction techniques such as dynamic partial order reduction (DPOR), making them mature and highly practical for checking large-scale systems [25, 51].

Despite these early successes, existing dmcks unfortunately fall short in addressing present reliability challenges of cloud systems. In particular, large-scale cloud systems are expected to be highly reliable in dealing with complex failures, not just one instance, but *multiple* of them. However, to the best of our knowledge, *no* existing dmcks can exercise multiple failures without exploding the state space. We elaborate this issue later; for now, we discuss complex failures in cloud environments.

Cloud systems run on large clusters of unreliable commodity machines, an environment that produces a growing number and frequency of failures, including "surprising" failures [2, 26]. Therefore, it is common to see complex failure-induced bugs such as the one below.

> **ZooKeeper Bug #335:** (1) Nodes A, B, C start with latest txid #10 and elect B as leader, (2) *B crashes*, (3) Leader election re-run; C becomes leader, (4) Client writes data; A and C commit new txid-value pair {#11:X}, (5) *A crashes before* committing tx #11, (6) C loses quorum, (7) *C crashes*, (8) *A reboots* and *B reboots*, (9) A becomes leader, (10) Client updates data; A and B commit a new txid-value pair {#11:Y}, (11) *C reboots after* A's new tx commit, (12) C synchronizes with A; C notifies A of {#11:X}, (13) A replies to C the "diff" starting with tx 12 (excluding tx {#11:Y}!), (14) Violation: permanent data inconsistency as A and B have {#11:Y} and C has {#11:X}.

The bug above is what we categorize as *deep bug*. To unearth deep bugs, dmck must permute a large number

[1] These systems are often referred with different names (*e.g.*, cloud software infrastructure, datacenter operating systems). For simplicity, we use the term "cloud systems".

of events, not only network events (messages), but also *crashes* and *reboots*. Although arguably deep bugs occur with lower probabilities than "regular" bugs, deep bugs do occur in large-scale deployments and have harmful consequences (§2.3). We observe that cloud developers are prompt in fixing deep bugs (in few weeks) as they seem to believe in Murphy's law; at scale, anything that can go wrong will go wrong.

As alluded above, the core problem is that state-of-the-art dmcks [22, 25, 34, 43, 50, 51] do not incorporate failure events to their state exploration strategies. They mainly address scalability issues related to message reorderings. Although some dmcks are capable of injecting failures, usually they only exercise at most one failure. The reason is simple: exercising crash/reboot events will exacerbate the state-space explosion problem. In this regard, existing dmcks do not scale and take very long time to unearth deep bugs. This situation led us to ask: *how should we advance dmck to discover deep bugs quickly and systematically, and thereby address present reliability challenges of cloud systems in dealing with complex failures?*

In this paper, we present semantic-aware model checking (SAMC; pronounced "Sam-C"), a white-box principle that takes simple semantic information of the target system and incorporates that knowledge in state-space reduction policies. In our observation, existing dmcks treat every target system as a complete black box, and therefore many times perform message re-orderings and crash/reboot injections that lead to the same conditions that have been explored in the past. These *redundant executions* must be removed significantly to tame the state-space explosion problem. We find that simple semantic knowledge can scale dmck greatly.

The main challenge of SAMC is in defining *what* semantic knowledge can be valuable for reduction policies and *how* to extract that information from the target system. We find that useful semantic knowledge can come from *event processing semantic* (*i.e.*, how messages, crashes, and reboots are processed by the target system). To help testers extract such information from the target system, we provide *generic event processing patterns*, patterns of how messages, crashes, and reboots are processed by distributed systems in general.

With this method, we introduce four novel semantic-aware reduction policies. First, *local-message independence* (LMI) reduces re-orderings of concurrent intra-node messages. Second, *crash-message independence* (CMI) reduces re-orderings of crashes among outstanding messages. Third, *crash recovery symmetry* (CRS) skips crashes that lead to symmetrical recovery behaviors. Finally, *reboot synchronization symmetry* (RSS) skips reboots that lead to symmetrical synchronization actions. Our reduction policies are *generic*; they are applicable to many distributed systems. SAMC users (*i.e.*, testers) only need to feed the policies with short *protocol-specific rules* that describe event independence and symmetry specific to their target systems.

SAMC is purely systematic; it does not incorporate randomness or bug-specific knowledge. Our policies run on top of sound model checking foundations such as state or architectural symmetry [9, 45] and independence-based dynamic partial order reduction (DPOR) [17, 20]. Although these foundations have been around for a decade or more, its application to dmck is still limited; these foundations require testers to define *what* events are actually independent or symmetrical. With SAMC, we can define fine-grained independence and symmetry.

We have built a prototype of SAMC (SAMPRO) from scratch for a total of 10,886 lines of code. We have integrated SAMPRO to three widely popular cloud systems, ZooKeeper [28], Hadoop/Yarn [47], and Cassandra [35] (old and latest stable versions; 10 versions in total). We have run SAMPRO on 7 different protocols (leader election, atomic broadcast, cluster management, speculative execution, read/write, hinted handoff, and gossiper). The protocol-specific rules are written in only 35 LOC/protocol on average. This shows the simplicity of applying SAMC reduction policies across different systems and protocols; all the rigorous state exploration and reduction are automatically done by SAMPRO.

To show the power of SAMC, we perform an extensive evaluation of SAMC's speed in finding deep bugs. We take 12 old real-world deep bugs that require multiple crashes and reboots (some involve as high as 3 crashes and 3 reboots) and show that SAMC can find the bugs one to three orders of magnitude faster compared to state-of-the-art techniques such as black-box DPOR, random+DPOR, and pure random. We show that this speed saves tens of hours of testing time. More importantly, some deep bugs cannot be reached by non-SAMC approaches, even after 2 days; here, SAMC's speed-up factor is potentially much higher. We also found 2 new bugs in the latest version of ZooKeeper and Hadoop.

To the best of our knowledge, our work is the first solution that systematically scales dmck with the inclusion of failures. We believe none of our policies have been introduced before. Our prototype is also the first available dmck for our target systems. Overall, we show that SAMC can address deep reliability challenges of cloud systems by helping them discover deep bugs faster.

The rest of the paper is organized as follows. First, we present a background and an extended motivation (§2). Next, we present SAMC and our four reduction policies (§3). Then, we describe SAMPRO and its integration to cloud systems (§4). Finally, we close with evaluations (§5), related work (§7), and conclusion (§8).

2 Background

This section gives a quick background on dmck and related terms, followed with a detailed overview of the state of the art. Then, we present cases of deep bugs and motivate the need for dmck advancements.

2.1 DMCK Framework and Terms

As mentioned before, we define *dmck* as software model checker that checks distributed systems directly at the implementation level. Figure 1 illustrates a dmck integration to a target distributed system, a simple representation of existing dmck frameworks [25, 34, 43, 51]. The dmck inserts an interposition layer in each node of the target system with the purpose of controlling all important events (*e.g.*, network messages, timeouts) and preventing the target system to process the events until the dmck enables them. A main dmck mechanism is the permutation of events; the goal is to push the target system into all possible ordering scenarios. For example, the dmck can enforce `abcd` ordering in one execution, `bcad` in another, and so on.

We now provide an overview of basic dmck terms we use in this paper and Figure 1. Each node of the target system has a *local state* (`ls`), containing many variables. An *abstract local state* (`als`) is a subset of the local state; dmck decides which `als` is important to check. The collection of all (and abstract) local states is the *global state* (`gs`) and the *abstract global state* (`ags`) respectively. The *network state* describes all the *outstanding messages* currently intercepted by dmck (*e.g.*, `abd`). To model check a specific protocol, dmck starts a *workload driver* (which restarts the whole system, runs specific workloads, etc.). Then, dmck generates many (typically hundreds/thousands) executions; an *execution* (or a *path*) is a specific ordering of events that dmck enables (*e.g.*, `abcd`, `dbca`) from an initial state to a termination point. A *sub-path* is a subset of a path/execution. An *event* is an action by the target system that is intercepted by dmck (*e.g.*, a network message) or an action that dmck can inject (*e.g.*, a crash/reboot). Dmck enables one event at a time (*e.g.*, `enable(c)`). To permute events, dmck runs *exploration methods* such as brute-force (*e.g.*, depth first search) or random. As events are permuted, the target system enters hard-to-reach states. Dmck continuously runs state *checks* (*e.g.*, safety checks) to verify the system's correctness. To reduce the state-space explosion problem, dmck can employ *reduction policies* (*e.g.*, DPOR or symmetry). A policy is *systematic* if it does not use randomness or bug-specific knowledge. In this work, we focus on advancing systematic reduction policies.

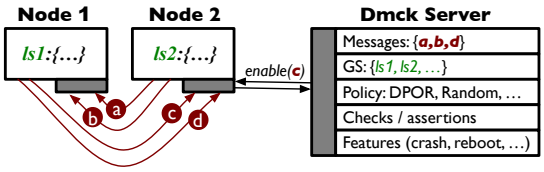

Figure 1: **DMCK.** *The figure illustrates a typical framework of a distributed system model checker (dmck).*

2.2 State-of-the-Art DMCKs

MODIST [51] is arguably one of the most powerful dmcks that comes with systematic reduction policies. MODIST has been integrated to real systems due to its exploration scalability. At the heart of MODIST is *dynamic partial order reduction (DPOR)* [17] which exploits the *independence* of events to reduce the state explosion. Independent events mean that it does not matter in what order the system execute the events, as their different orderings are considered equivalent.

To illustrate how MODIST adopts DPOR, let's use the example in Figure 1, which shows four concurrent outstanding messages `abcd` (`a` and `b` for `N1`, `c` and `d` for `N2`). A brute-force approach will try all possible combinations (`abcd`, `abdc`, `acbd`, `acdb`, `cabd`, and so on), for a total of 4! executions. Fortunately, the notion of event independence can be mapped to distributed system properties. For example, MODIST specifies this reduction policy: a message to be processed by a given node is independent of other concurrent messages destined to other nodes (based on vector clocks). Applying this policy to the example in Figure 1 implies that `a` and `b` are dependent[1] but they are independent of `c` and `d` (and vice versa). Since only dependent events need to be reordered, this reduction policy leads to only 4 executions (`ab-cd`, `ab-dc`, `ba-cd`, `ba-dc`), giving a 6x speed-up (4!/4).

Although MODIST's speed-up is significant, we find that one scalability limitation of its DPOR application is within its *black-box* approach; it only exploits general properties of distributed systems to define message independence. It does not exploit any semantic information from the target system to define more independent events. We will discuss this issue later (§3.1).

Dynamic interface reduction (DIR) [25] is the next advancement to MODIST. This work suggests that a complete dmck must re-order not only messages (global events) but also thread interleavings (local events). The reduction intuition behind DIR is that different thread interleavings often lead to the same global events (*e.g.*, a node sends the same messages regardless of how threads are interleaved in that node). DIR records local explo-

[1] In model checking, "dependent" events mean that they must be re-ordered. "Dependent" does not mean "causally dependent".

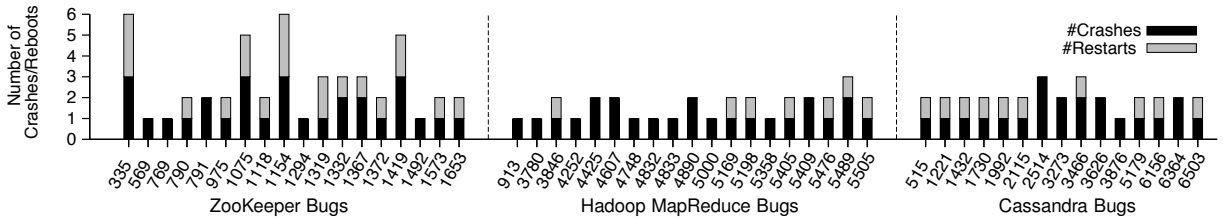

Figure 2: **Deep Bugs.** *The figure lists deep bugs from our bug study and depicts how many crashes and reboots must happen to reproduce the bugs. Failure events must happen in a specific order in a long sequence of events. These bugs came from many protocols including ZooKeeper leader election and atomic broadcast, Hadoop MapReduce speculative execution, job/task trackers, and resource/application managers, and Cassandra gossiper, anti-entropy, mutation, and hinted handoff. These bugs led to failed jobs, node unavailability, data loss, inconsistency, and corruption. They were labeled as "major", "critical", or "blocker". 12 of these bugs happened within the last one year. The median response time (i.e., time to fix) is two weeks. There are few bugs that involve 4+ reboots and 4+ crashes that we do not show here.*

ration and replays future incoming messages without the need for global exploration. In our work, SAMC focuses only on global exploration (message and failure re-orderings). We believe DIR is orthogonal to SAMC, similar to the way DIR is orthogonal to MODIST.

MODIST and DIR are examples of dmcks that employ advanced systematic reduction policies. LMC [22] is similar to DIR; it also decouples local and global exploration. dBug [43] applies DPOR similarly to MODIST. There are other dmcks such as MACEMC [34] and CrystalBall [50] that use basic exploration methods such as depth first (DFS), weight-based, and random searches.

Other than the aforementioned methods, *symmetry* is another foundational reduction policy [16, 45]. Symmetry-based methods exploit the architectural symmetry present in the target system. For example, in a ring of nodes, one can rotate the ring without affecting the behavior of the system. Symmetry is powerful, but we find no existing dmcks that adopt symmetry.

Besides dmcks, there exists sophisticated testing frameworks for distributed systems (*e.g.*, FATE [23], PREFAIL [31], SETSUDO [30], OpenStack fault-injector [32]). This set of work emphasizes the importance of multiple failures, but their major limitation is that they are not a dmck. That is, they cannot systematically control and permute non-deterministic choices such as message and failure reorderings.

2.3 Deep Bugs

To understand the unique reliability challenges faced by cloud systems, we performed a study of reliability bugs of three popular cloud systems: ZooKeeper [28], Hadoop MapReduce [47], and Cassandra [35]. We scanned through thousands of issues from their bug repositories. We then tagged complex reliability bugs that can only be caught by a dmck (*i.e.*, bugs that can occur only on specific orderings of events). We found 94 dmck-catchable bugs.[1] Our major finding is that 50% of them are deep bugs (require complex re-ordering of not only messages but also crashes and reboots).

Figure 2 lists the deep bugs found from our bug study. Many of them were induced by multiple crashes and reboots. Worse, to reproduce the bugs, crash and reboot events must happen in a specific order within a long sequence of events (*e.g.*, the example bug in §1). Deep bugs lead to harmful consequences (*e.g.*, failed jobs, node unavailability, data loss, inconsistency, corruption), but they are hard to find. We observe that since there is no dmck that helps in this regard, deep bugs are typically found in deployment (via logs) or manually, then they get fixed in few weeks, but afterwards as code changes continuously, new deep bugs tend to surface again.

2.4 Does State of the-Art Help?

We now combine our observations in the two previous sections and describe why state-of-the-art dmcks do not address present reliability challenges of cloud systems.

First, *existing systematic reduction policies often cannot find bugs quickly*. Experiences from previous dmck developments suggest that significant savings from sound reduction policies do not always imply high bug-finding effectiveness [25, 51]. To cover deep states and find bugs, many dmcks revert to non-systematic methods such as randomness or manual checkpoints. For example, MODIST combines DPOR with random walk to "jump" faster to a different area of the state space (§4.5 of [51]). DIR developers find new bugs by manually setting "interesting" checkpoints so that future state explorations happen from the checkpoints (§5.3 of [25]). In our work, although we use different target systems, we are able to reproduce the same experiences above (§5.1).

[1] Since this is a manual effort, we might miss some bugs. We also do not report "simple" bugs (*e.g.*, error-code handling) that can be caught by unit tests.

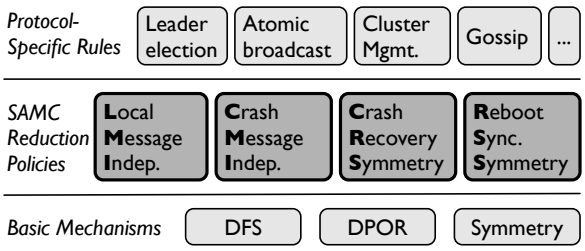

Figure 3: **SAMC Architecture.**

Second, *existing dmcks do not scale with the inclusion of failure events*. Given the first problem above, exercising multiple failures will just exacerbate the state-space explosion problem. Some frameworks that can explore multiple failures such as MACEMC [34] only do so in a random way; however, in our experience (§5.1), randomness many times cannot find deep bugs quickly. MODIST also enabled only one failure. In reality, multiple failures is a big reliability threat, and thus must be exercised.

We conclude that finding systematic (no random/checkpoint) policies that can find deep bugs is still an open dmck research problem. We believe without semantic knowledge of the target system, dmck hits a scalability wall (as also hinted by DIR authors; §8 of [25]). In addition, as crashes and reboots need to be exercised, we believe recovery semantics must be incorporated into reduction policies. All of these observations led us to SAMC, which we describe next.

3 SAMC

Semantic-aware model checking (SAMC) is a white-box model checking approach that takes semantic knowledge of how events (*e.g.*, messages, crashes, and reboots) are processed by the target system and incorporates that information in reduction policies. To show the intuition behind SAMC, we first give an example of a simple leader election protocol. Then, we present SAMC architecture and our four reduction policies.

3.1 An Example

In a simple leader election protocol, every node broadcasts its vote to reach a quorum and elect a leader. Each node begins by voting for itself (*e.g.*, `N2` broadcasts `vote=2`). Each node receives vote broadcasts from other peers and processes every vote with this simplified code segment below. As depicted in the code segment below, if an incoming vote is less than the node's current vote, it is simply discarded. If it is larger, the node changes its vote and broadcasts the new vote.

```
if (msg.vote < myVote) {discard;}
else {myVote = msg.vote; broadcast(myVote);}
```

Let's assume `N4` with `vote=4` is receiving three concurrent messages with votes `1`, `2`, and `3` from its peers. Here, a dmck with a black-box DPOR approach must perform 6 (3!) orderings (`123`, `132`, and so on). This is because a black-box DPOR does *not* know the *message processing semantic* (*i.e.*, how messages will be processed by the receiving node). Thus, a black-box DPOR must treat all of them as dependent (§2.2); they must be re-ordered for soundness. However, by knowing the processing logic above, a dmck can soundly conclude that all orderings will lead to the same state; all messages will be discarded by `N4` and its local state will not change. Thus, a semantic-aware dmck can reduce the 6 redundant executions to just 1 execution.

The example above shows a scalability limitation of a black-box dmck. Fortunately, simple semantic knowledge has a great potential in removing redundant executions. Furthermore, semantic knowledge can be incorporated on top of sound model checking foundations such as DPOR and symmetry, as we describe next.

3.2 Architecture

Figure 3 depicts the three levels of SAMC: sound exploration mechanisms, reduction policies, and protocol-specific rules. SAMC is built on top of sound model checking foundations such as DPOR [17, 20] and symmetry [9, 45]. We name these foundations as mechanisms because a dmck must specify accordingly what events are dependent/independent and symmetrical, which in SAMC will be done by the reduction policies and protocol-specific rules.

Our main contribution lies within our four novel *semantic-aware reduction policies*: local-message independence (LMI), crash-message independence (CMI), crash recovery symmetry (CRS), and reboot synchronization symmetry (RSS). To the best of our knowledge, none of these approaches have been introduced in the literature. At the heart of these policies are *generic event processing patterns* (*i.e.*, patterns of how messages, crashes, and reboots are processed by distributed systems). Our policies and patterns are simple and powerful; they can be applied to many different distributed systems. Testers can extract the patterns from their target protocols (*e.g.*, leader election, atomic broadcast) and write protocol-specific rules in few lines of code.

In the next section, we first present our four reduction policies along with the processing patterns. Later, we will discuss ways to extract the patterns from target systems (§3.4) and then show the protocol-specific rules for our target systems (§4.2).

3.3 Semantic-Aware Reduction Policies

We now present four semantic-aware reduction policies that enable us to define fine-grained event dependency/independency and symmetry beyond what black-box approaches can do.

3.3.1 Local-Message Independence (LMI)

We define *local messages* as messages that are concurrently in flight to a given node (*i.e.*, intra-node messages). As shown in Figure 4a, a black-box DPOR treats the message processing semantic inside the node as a black box, and thus must declare the incoming messages as dependent, leading to 4! permutation of `abcd`. On the other hand, with white-box knowledge, local-message independence (LMI) can define *independency relationship among local messages*. For example, illustratively in Figure 4b, given the node's local state (`ls`) and the processing semantic (embedded in the `if` statement), LMI is able to define that `a` and `b` are dependent, `c` and `d` are dependent, but the two groups are independent, which then leads to only 4 re-orderings. This reduction illustration is similar to the one in Section 2.2, but this time LMI enables DPOR application on local messages.

LMI can be easily added to a dmck. A dmck server typically has a complete view of the local states (§2.1). What is needed is the *message processing semantic*: how will a node (`N`) process an incoming message (`m`) given the node's current local state (`ls`)? The answer lies in these four simple *message processing patterns* (discard, increment, constant, and modify):

```
Discard:                Increment:
if (pd(m,ls))           if (pi(m,ls))
  (noop);                 ls++;

Constant:               Modify:
if (pc(m,ls))           if (pm(m,ls))
  ls = Const;             ls = modify(m,ls);
```

In practice, `ls` and `m` contain many fields. For simplicity, we treat them as integers. The functions with prefix `p` are boolean read-only functions (predicates) that compare an incoming message (`m`) with respect to the local state (`ls`); for example, `pd` can return true if `m<s`. The first pattern is a *discard* pattern where the message is simply discarded if `pd` is true. This pattern is prevalent in distributed systems with votes/versions; old votes/versions tend to be discarded (*e.g.*, our example in §3.1). The *increment* pattern performs an increment-by-one update if `pi` is true, which is also quite common in many protocols (*e.g.*, counting commit acknowledgements). The *constant* pattern changes the local state to a constant whenever `pc` is true. Finally, the *modify* pattern changes the local state whenever `pm` is true.

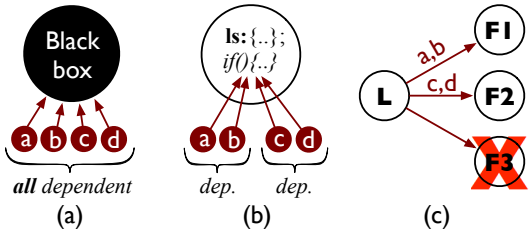

Figure 4: **LMI and CMI.** *The figures illustrate (a) a black-box approach, (b) local-message independence with white-box knowledge, and (c) crash-message independence.*

Based on these patterns, we can apply LMI in the following ways. (1) `m1` is independent of `m2` if `pd` is true on any of `m1` and `m2`. That is, if `m1` (or `m2`) will be discarded, then it does not need to be re-ordered with other messages. (2) `m1` is independent of `m2` if `pi` (or `pc`) is true on both `m1` and `m2`. That is, the re-orderings do not matter because the local state is monotonically increasing by one (or changed to the same constant). (3) `m1` and `m2` are dependent if `pm` is true on `m1` and `pd` is not true on `m2`. That is, since both messages modify the local state in unique ways, then the re-orderings can be "interesting" and hence should be exercised. All these rules are continuously evaluated before every event is enabled. If multiple cases are true, dependency has higher precedence than independency.

Overall, LMI allows dmck to smartly skip redundant re-orderings by leveraging simple patterns. The job of the tester is to find the message processing patterns from a target protocol and write *protocol-specific rules* (*i.e.*, filling in the content of the four LMI predicate functions (`pd`, `pi`, `pc`, and `pm`) specific to the target protocol). As an example, for our simple leader election protocol (§3.1), `pd` can be as simple as: return `m.vote < ls.myVote`.

3.3.2 Crash-Message Independence (CMI)

Figure 4c illustrates the motivation behind our next policy. The figure resembles an atomic broadcast protocol where a leader (`L`) sends commit messages to the followers (`Fs`). Let's assume commit messages `ab` to `F1` and `cd` to `F2` are still in flight (*i.e.*, currently outstanding in the dmck; not shown). In addition, the dmck would like to crash `F3`, which we label as a crash event `X`. The question we raise is: how should `X` be re-ordered with respect to other outstanding messages (`a`, `b`, `c`, and `d`)?

As we mentioned before, we find *no* single dmck that incorporates crash semantics into reduction policies. As an implication, in our example, the dmck must reorder `X` with respect to other outstanding messages, generating executions `Xabcd`, `aXbcd`, `abXcd`, and so on. Worse, when `abcd` are reordered, `X` will be reordered again. We find this as one major fundamental problem why existing dmcks do not scale with the inclusion of failures.

To solve this, we introduce crash-message independence (CMI) which defines *independency relationship between a to-be-injected crash and outstanding messages*. The key lies in these two crash reaction patterns (global vs. local impact) running on the surviving nodes (*e.g.*, the leader node in Figure 4c).

```
Global impact:
if (pg(X,ls))
  modify(ls);
  sendMsg();
```

```
Local impact:
if (pl(X,ls))
  modify(ls);
```

The functions with prefix `p` are predicate functions that compare the crash event `X` with respect to the surviving node's local state (*e.g.*, the leader's local state). The `pg` predicate in the *global-impact* pattern defines that the crash `X` during the local state `ls` will lead to a local state change *and* new outgoing messages (*e.g.*, to other surviving nodes). Here, no reduction can be done because the new crash-induced outgoing messages must be re-ordered with the current outstanding messages. On the other hand, reduction opportunities exist within the *local-impact* pattern, wherein the `pl` predicate specifies that the crash will just lead to a local state change but not new messages, which implies that the crash does not need to be re-ordered.

Based on the two crash impact patterns, we apply CMI in the following ways. Given a local state `ls` at node `N`, a peer failure `X`, and outstanding messages ($\texttt{m}_1 ... \texttt{m}_n$) from `N` to other surviving peers, CMI performs: (1) If `pl` is true, then `X` and $\texttt{m}_1 ... \texttt{m}_n$ are independent. (2) If `pg` is true, then `X` and $\texttt{m}_1 ... \texttt{m}_n$ are dependent. In Figure 4c for example, if `pl` is true in node `L`, then `X` does not impact outstanding messages to `F1` and `F2`, and thus `X` is independent to `abcd`; exercising `Xabcd` is sufficient.

An example of CMI application is a quorum-based write protocol. If a follower crash occurs and quorum is still established, the leader will just decrease the number of followers (local state change only). Here, for the protocol-specific rules, the tester can specify `pl` with `#follower >= majority` and `pg` with the reverse. Overall, CMI helps dmck scale with the inclusion of failures, specifically by skipping redundant re-orderings of crashes with respect to outstanding messages.

3.3.3 Crash Recovery Symmetry (CRS)

Before we discuss our next reduction policy, we emphasize again the difference between message event and crash/reboot event. Message events are generated by the target system, and thus dmck can only reduce the number of re-orderings (but it cannot reduce the events). Contrary, crash events are generated by dmck, and thus there exists opportunities to reduce the number of injected crashes. For example, in Figure 4c, in addition to crashing `F3`, the dmck can also crash `F1` and `F2` in different executions, but that might not be necessary.

To omit redundant crashes, we develop crash recovery symmetry (CRS). The intuition is that some crashes often lead to symmetrical recovery behaviors. For example, let's assume a 4-node system with node roles `FFFL`. At this state, crashing the first or second or third node perhaps lead to the same recovery since all of them are followers, and thereby injecting one follower crash could be enough. Further on, if the system enters a slightly different state, `FFLF`, crashing any of the followers might give the same result as above. However, crashing the leader in the two cases (`N4` in the first case and `N3` in the second) should perhaps be treated differently because the recovery might involve the dead leader ID. The goal of CRS is to help dmck with crash decision.

The main question in implementing CRS is: how to incorporate crash recovery semantics into dmck? Our solution is to compute *recovery abstract global state* (`rags`), a simple and concise representation of crash recovery. CRS builds `rags` with the following steps:

First, we define that two recovery actions are symmetrical if they produce the same messages and change the same local states in the same way.

Second, we extract recovery logic from the code by flattening the predicate-recovery pairs (*i.e.*, recovery-related `if` blocks). Figure 5 shows a simple example. Different recovery actions will be triggered based on which recovery predicate ($\texttt{pr}_1$, $\texttt{pr}_2$, or $\texttt{pr}_3$) is true. Each predicate depends on the local state and the information about the crashing node. Our key here is to map each predicate-recovery pair to this formal pattern:

```
if (pr_i(ls, C.ls))
   modify(rals_i);
   (and/or)
   sendMsg(rals_i);
```

Here, $\texttt{pr}_i$ is the recovery predicate for the i-th recovery action, and $\texttt{rals}_i$ is the recovery abstract local state (*i.e.*, a subset of all fields of the local state involved in recovery). That is, each recovery predicate defines what recovery abstract local state that matters (*i.e.*, $\texttt{pr}_i \rightarrow \{\texttt{rals}_i\}$). For example, in Figure 5, if $\texttt{pr}_1$ is true, then $\texttt{rals}_1$ only contains the `follower` variable; if $\texttt{pr}_3$ is true, $\texttt{rals}_3$ contains `role` and `leaderId` variables.

Third, before we crash a node, we check which $\texttt{pr}_i$ will be true on each surviving node and then obtain the $\texttt{rals}_i$. Next, we combine $\texttt{rals}_i$ of all surviving nodes and *sort* them into a recovery abstract global state (`rags`); sorting `rags` helps us exploit topological symmetry (*e.g.*, individual node IDs often do not matter).

Fourth, given a plan to crash a node, the algorithm above gives us the `rags` that represents the corresponding recovery action. We also maintain a history of `rags`

```
broadcast()  sendMsgToAll(role, leaderId);
quorumOkay()  return (follower > nodes / 2);

// pr1
if (role == L && C.role == F && quorumOkay())
  follower--;
// pr2
if (role == L && C.role == F && !quorumOkay())
  follower = 0;
  role = S;
  broadcast();
// pr3
if (role == F && C.role == L)
  leaderId = myId;
  broadcast();
```

Figure 5: **Crash Recovery in Leader Election.** *The figure shows a simplified example of crash recovery in a leader election protocol. The code runs in every node.* `C` *implies the crashing node; each node typically has a view of the states of its peers. Three predicate-recovery pairs are shown (*$\texttt{pr}_1$, $\texttt{pr}_2$, *and* $\texttt{pr}_3$*). In the first, if quorum still exists, the leader simply decrements the follower count. In the second, if quorum breaks, the leader falls back to searching mode (*`S`*). In the third, if the leader crashes, the node (as a follower) votes for itself and broadcasts the vote to elect a new leader.*

of previous injected crashes. If the `rags` already exists in the history, then the crash is skipped because it will lead to a symmetrical recovery of the past.

To recap with a concrete example, let's go back to the case of `FFFL` where we plan to enable crash(`N1`). Based on the code in Figure 5, the `rags` is {*, ∅, ∅, `#follower=3`}; * implies the crashing node, ∅ means there is no true predicate at the other two follower nodes, and `#follower=3` comes from $\texttt{rals}_1$ of $\texttt{pr}_1$ of `N4` (the leader). CRS will sort this and check the history, and assuming no hit, then crash(`N1`) will be enabled. In another execution, SAMC finds that crash(`N2`) at `FFFL` will lead to `rags`:{∅, *, ∅, `#follower=3`}, which after sorting will hit the history, and hence crash(`N2`) is skipped. If the system enters a different state `FFLF`, no follower crash will be injected, because the `rags` will be the same as above. In terms of leader crash, crashing the leader in the two cases will be treated differently because in a leader crash, $\texttt{pr}_3$ is true on followers and $\texttt{pr}_3$ involves `leaderId` which is different in the two cases.

In summary, the foundation of CRS is the computation of recovery abstract global state (`rags`) from the crash recovery logic extracted from the target system via the $\texttt{pr}_i \rightarrow \{\texttt{rals}_i\}$ pattern. We believe this extraction method is simple because CRS does *not* need to know the specifics of crash recovery; CRS just needs to know what variables are involved in recovery (*i.e.*, the `rals`) .

3.3.4 Reboot Synchronization Symmetry (RSS)

Reboots are also essential to exercise (§2.3), but if not done carefully, will introduce more scalability problems. Reboot reduction policy is needed to help dmck inject reboots "smartly". The intuition behind reboot synchronization symmetry (RSS) is similar to that of CRS. When a node reboots, it typically *synchronizes* itself with the peers. However, a reboot will not lead to a new scenario if the current state of the system is similar to the state when the node crashed. To implement RSS, we extract reboot-synchronization predicates and the corresponding actions. Since the overall approach is similar to CRS, we omit further details.

In our experience RSS is extremely powerful. For example, it allows us to find deep bugs involving multiple reboots in the ZooKeeper atomic broadcast (ZAB) protocol. RSS works efficiently here because reboots in ZAB are only interesting if the live nodes have seen new commits (*i.e.*, the dead node falls behind). In contrast, a black-box dmck without RSS initiates reboots even when the live nodes are in similar states as in before the crash, prolonging the discovery of deep bugs.

3.4 Pattern Extraction

We have presented four general, simple, and powerful semantic-aware reduction policies along with the generic event processing patterns. With this, testers can write protocol-specific rules by extracting the patterns from their target systems. Given the patterns described in previous sections, a tester must perform what we call as "extraction" phase. Here, the tester must extract the patterns from the target system and write protocol-specific rules specifically by filling in the predicates and abstractions as defined in previous sections; in Section 4.2, we will show a real extraction result (*i.e.*, real rules). Currently, the extraction phase is manual; we leave automated approaches as a future work (§6). Nevertheless, we believe manual extraction is bearable for several reasons. First, today is the era of DevOps [36] where developers are testers and vice versa; testers know the internals of their target systems. This is also largely true in cloud system development. Second, the processing patterns only cover high-level semantics; testers just fill in the predicates and abstractions but no more details. In fact, simple semantics are enough to significantly help dmck go faster to deeper states.

4 Implementation and Integration

In this section, we first describe our SAMC prototype, SAMPRO, which we built from scratch because existing

Local-Message Independence (LMI)

```
bool pd : !newVote(m, s);

bool pm : newVote(m, s);

bool newVote(m, s) :
 if (m.ep > s.ep)
   ret 1;
 else if (m.ep == s.ep)
  if (m.tx > s.tx)
   ret 1;
  else if (m.tx == s.tx &&
           m.lid > s.lid)
   ret 1;

 ret 0;
```

Crash-Message Independence (CMI)

```
bool pg (s, X) :
 if (s.rl == F && X.rl == L)
  ret 1;
 if (s.rl == L && X.rl == F
     && !quorumAfterX(s)
  ret 1;
 if (s.rl == S && X.rl == S)
  ret 1;

bool pl (s, X) :
 if (s.rl == L && X.rl == F
     && quorumAfterX(s))
  ret 1;


bool quorumAfterX(s) :
  ret ((s.fol-1) >=
        s.all/2);
```

Crash Recovery Symmetry (CRS)

```
bool pr1(s,C):
 if (s.rl == L && C.rl == F
     && quorumAfterX(s))
  ret 1;
rals1:{rl,fol,all};

bool pr2(s,C):
 if (s.rl == L && C.rl == F
     && !quorumAfterX(s))
 ret 1;
rals2: {rl,fol,lid,ep,tx,clk}

bool pr3(s,C):
 if (s.rl == F && c.rl == L)
  ret 1;
rals3: {rl,fol,lid,ep,tx,clk}

bool pr4:
 if (s.rl == S)
  ret 1;
rals4: {rl,lid,ep,tx,clk}
```

RSS

(**) See caption

Table 1: **Protocol-Specific Reduction Rules for ZLE.** *The code above shows the actual protocol-specific rules for ZLE protocol. These rules are the inputs to the four reduction policies. The rule for ZLE's RSS is not shown (it is similar to ZLE's CRS) and many variables are abbreviated (ep: epoch, tx: latest transaction ID, lid: leader ID, rl: role, fol: follower count, all: total node count, clk: logical clock, L: leading, F: following, S: searching, X/C: crashing node). LMI* `pc` *and* `pi` *predicates are not used for ZLE, but used for other protocols.*

dmcks are either proprietary [51] or only work on restricted high-level languages (*e.g.*, Mace [34]). We will then describe SAMPRO integration to three widely popular cloud systems, ZooKeeper [28], Hadoop/Yarn [47], and Cassandra [35]. Prior to SAMPRO, there was no available dmck for these systems; they are still tested via unit tests, and the test code size is bigger than the main code, but the tests are far from reaching deep bugs.

4.1 SAMPRO

SAMPRO is written in 10,886 lines of code in Java, which includes all the components mentioned in Section 2.1 and Figure 1. The detailed anatomy of dmck has been thoroughly explained in literature [22, 25, 34, 43, 51], and therefore for brevity, we will not discuss many engineering details. We will focus on SAMC-related parts.

We design SAMPRO to be highly portable; we do not modify the target code base significantly as we leverage a mature interposition technology, AspectJ, for interposing network messages and timeouts. Our interposition layer also sends local state information to the SAMPRO server. SAMPRO is also equipped with crash and reboot scripts specific to the target systems. The tester can specify a budget of the maximum number of crashes and reboots to inject per execution. SAMPRO employs basic reduction mechanisms and advanced reduction policies as described before. We deploy safety checks at the server (*e.g.*, no two leaders). If a check is violated, the trace that led to the bug is reported and can be deterministically replayed in SAMPRO. Overall, we have built all the necessary features to show the case of SAMC. Other features such as intra-node thread interleavings [25], scale-out parallelism [44], and virtual clock for network delay [51] can be integrated to SAMPRO as well.

4.2 Integration to Target Systems

In our work, the target systems are ZooKeeper, Hadoop 2.0/Yarn, and Cassandra. ZooKeeper [28] is a distributed synchronization service acting as a backbone of many distributed systems such as HBase and High-Availability HDFS. Hadoop 2.0/Yarn [47] is the current generation of Hadoop that separates cluster management and processing components. Cassandra [35] is a distributed key-value store derived from Amazon Dynamo [13].

In total, we have model checked 7 protocols: ZooKeeper leader election (ZLE) and atomic broadcast (ZAB), Hadoop cluster management (CM) and speculative execution (SE), and Cassandra read/write (RW), hinted handoff (HH) and gossiper (GS). These protocols are highly asynchronous and thus susceptible to message re-orderings and failures.

Table 1 shows a real sample of protocol-specific rules that we wrote. Rules are in general very short; we only wrote 35 lines/protocol on average. This shows the simplicity of SAMC's integration to a wide variety of distributed system protocols.

5 Evaluation

We now evaluate SAMC by presenting experimental results that answer the following questions: (1) How fast is SAMC in finding deep bugs compared to other state-of-the-art techniques? (2) Can SAMC find new deep bugs? (3) How much reduction ratio does SAMC provide?

To answer the first question, we show SAMC's effectiveness in finding old bugs. For this, we have integrated SAMPRO to old versions of our target systems that carry deep bugs: ZooKeeper v3.1.0, v3.3.2, v3.4.3, and v3.4.5, Hadoop v2.0.3 and v2.2.0, and Cassandra v1.0.1 and v1.0.6. To answer the second question, we have integrated SAMPRO to two recent stable versions: ZooKeeper v3.4.6 (released March 2014) and Hadoop v2.4.0 (released April 2014). In total, we have integrated SAMPRO to 10 versions, showing the high portability of our prototype. Overall, our extensive evaluation exercised more than 100,000 executions and used approximately 48 full machine days.

5.1 Speed in Finding Old Bugs

This section evaluates the speed of SAMC vs. state-of-the-art techniques in finding old deep bugs. In total, we have reproduced 12 old deep bugs (7 in ZooKeeper, 3 in Hadoop, and 2 in Cassandra). Figure 6 illustrates the complexity of the deep bugs that we reproduced.

Table 2 shows the result of our comparison. We compare SAMC with basic techniques (DFS and Random) and advanced state-of-the-art techniques such as black-box DPOR ("bDP") and Random+bDP ("rDP"). Black-box DPOR is the MODIST-style of DPOR (§2.2). We include Random+DPOR to mimic the way MODIST authors found bugs faster (§2.4). The table shows the number of executions to hit the bug. As a note, software model checking with the inclusion of failures takes time (back-and-forth communications between the target system and the dmck server, killing and restarting system processes multiple times, restarting the whole system from a clean state, etc.). On average, each execution runs for 40 seconds and involves a long sequence of 20-120 events including the necessary crashes and reboots to hit the bug. We do not show the result of running DFS because it never hits most of the bugs.

Based on the result in Table 2, we make several conclusions. First, with SAMC, we prove that smart systematic approaches can reach to deep bugs quickly. We do not need to revert to randomness or incorporate checkpoints. As a note, we are able to reproduce every deep bug that we picked; we did not skip any of them. (Hunting more deep bugs is possible, if needed).

MapReduce-5505: (1) A job finishes, (2) Application manager (AM) sends a "remove-app" message to Resource Manager (RM), (3) RM receives the message, (4) AM is unregistering, (5) *RM crashes* before completely processes the message, (6) AM finishes unregistering, (7) *RM reboots* and reads the old state file, (8) RM thinks the job has never started and runs the job again.

Cassandra-3395 (1) Three nodes N1-3 started and formed a ring, (2) Client writes data, (3) *N3 crashes*, (4) Client updates the data via N1; N3 misses the update, (5) *N3 reboots*, (6) N1 begins the hinted handoff process, (7) Another client reads the data with strong consistency via N1 as a coordinator, (8) N1 and N2 provide the updated value, but N3 still provides the stale value, (9) The coordinator gets "confused" and returns the stale value to the client!

Figure 6: **Complexity of Deep Bugs.** *Above are two sample deep bugs in Hadoop and Cassandra. A sample for ZooKeeper was shown in the introduction (§1). Deep bugs are complex to reproduce; crash and reboot events must happen in a specific order within a long sequence of events (there are more events behind the events we show in the bug descriptions above). To see the high degree of complexity of other old bugs that we reproduced, interested readers can click the issue numbers (hyperlinks) in Table 2.*

Second, SAMC is one to two orders of magnitude faster compared to state-of-the-art techniques. Our speed-up is up to 271x (33x on average). But most importantly, there are bugs that other techniques cannot find even after 5000 executions (around 2 days). Here, SAMC's speed-up factor is potentially much higher (labeled with "⇑"). Again, in the context of dmck (a process of hours/days), large speed-ups matter. In many cases, state-of-the-art policies such as bDP and rDP cannot reach the bugs even after very long executions. The reasons are the two problems we mentioned earlier (§2.4). Our micro-analysis (not shown) confirmed our hypothesis that non-SAMC policies frequently make redundant crash/reboot injections and event re-orderings that anyway lead to insignificant state changes.

Third, Random is truly "random". Although many previous dmcks embrace randomness in finding bugs [34, 51], when it comes to failure-induced bugs, we have a different experience. Sometimes Random is as competitive as SAMC (*e.g.*, ZK-975), but sometimes Random is much slower (*e.g.*, ZK-1419), or worse Random sometimes did not hit the bug (*e.g.*, ZK-1492, MR-5505). We find that some bugs require crashes

Issue#	Protocol	E	C	R	#Executions bDP	RND	rDP	SAMC	Speed-up of SAMC vs. bDP	RND	rDP
ZooKeeper-335	ZAB	120	3	3	↑5000	1057	↑5000	117	⇑43	9	⇑43
ZooKeeper-790	ZLE	21	1	1	14	225	82	7	2	32	12
ZooKeeper-975	ZLE	21	1	1	967	71	163	53	18	1	3
ZooKeeper-1075	ZLE	25	3	2	1081	86	250	16	68	5	16
ZooKeeper-1419	ZLE	25	3	2	924	2514	987	100	9	25	10
ZooKeeper-1492	ZLE	31	1	0	↑5000	↑5000	↑5000	576	⇑9	⇑9	⇑9
ZooKeeper-1653	ZAB	60	1	1	945	3756	3462	11	86	341	315
MapReduce-4748	SE	25	1	0	22	6	6	4	6	2	2
MapReduce-5489	CM	20	2	1	↑5000	↑5000	↑5000	53	⇑94	⇑94	⇑94
MapReduce-5505	CM	40	1	1	1212	↑5000	1210	40	30	⇑125	30
Cassandra-3395	RW+HH	25	1	1	2552	191	550	104	25	2	5
Cassandra-3626	GS	15	2	1	↑5000	↑5000	↑5000	96	⇑52	⇑52	⇑52

Table 2: **SAMC Speed in Finding Old Bugs.** *The first column shows old bug numbers in ZooKeeper, Hadoop, and Cassandra that we reproduced. The bug numbers are clickable (contain hyperlinks). The protocol column lists where the deep bugs were found; full protocol names are in §4.2. "E", "C" and "R" represent the number of events, crashes, and reboots necessary to hit the bug. The numbers in the middle four columns represent the number of executions to hit the bug across different policies. "bDP", "RND", and "rDP" stand for black-box DPOR (in* MODIST*), random, and random + black-box DPOR respectively. The SAMC column represents our reduction policies and rules. The last three columns represent the speed-ups of SAMC over the other three techniques. We stop at 5000 executions (around 2 days) if the bug cannot be found; potentially many more executions are required to hit the bug (labeled with "↑"). Thus, speed-up numbers marked with "⇑" are potentially much higher. In the experiments above, the bugs are reproduced using 3-4 nodes. We also have run DFS but do not show the result because in most cases DFS cannot hit the bugs. For model checking the SE protocol, "1C" means one straggler; we emulate a node slowdown as a failure event by modifying the progress report of the "slow" node. SE involves 20+ events but most of them are synchronized stages and cannot be re-ordered, which explains why the SE bug can be found quickly with all policies.*

and/or reboots to happen at very specific points, which is probabilistically hard to reach with randomness. With SAMC, we show that being systematic and semantic aware is consistently effective.

5.2 Ability of Finding New Bugs

The previous section was our main focus of evaluation. In addition to this, we have integrated SAMPRO to recent stable versions of ZooKeeper (v3.4.6, released March 2014) and Hadoop (v2.4.0, released April 2014). In just hours of deployment, we found 1 new ZLE bug involving 2 crashes, 2 reboots, and 52 events, and 1 new Hadoop speculative execution bug involving 2 failures and 32 events. These two new bugs are distributed data race bugs. The ZLE bug causes the ZooKeeper cluster to create two leaders at the same time. The Hadoop bug causes a speculative attempt on a job that is wrongly moved to a scheduled state, which then leads to an exception and a failed job. We can deterministically reproduce the bugs multiple times and we have reported the bugs to the developers. Currently, the bugs are still marked as major and critical, the status is still open, and the resolution is still unresolved.

We also note that in order to unearth more bugs, a dmck must have several complete features: workload generators that cover many protocols, sophisticated perturbations (*e.g.*, message re-ordering, fault injections) and detailed checks of specification violations. Further discussions can be found in our previous work [23]. Currently, SAMPRO focuses on speeding up the perturbation part. By deploying more workload generators and specification checks in SAMPRO, more deep bugs are likely to be found. As an illustration, the 94 deep bugs we mentioned in Section 2.3 originated from various protocols and violated a wide range of specifications.

5.3 Reduction Ratio

Table 3 compares the reduction ratio of SAMC over black-box DPOR (bDP) with different budgets (#crashes and #reboots). This evaluation is slightly different than the bug-finding speed evaluation in Section 5.1. Here, we measure how many executions in bDP are considered redundant based on our reduction policies and protocol-specific rules. Specifically, we run bDP for 3000 executions and run SAMC policies on the side to mark the redundant executions. The reduction ratio is then 3000 divided by the number of non-redundant executions. Table 3 shows that SAMC provides between 37x-166x execution reduction ratio in model checking ZLE and ZAB protocols across different crash/reboot budgets.

Table 3b shows that with each policy the execution reduction ratio increases when the number of crashes and

C	R	Execution Reduction Ratio in ZLE	Execution Reduction Ratio in ZAB
1	1	37	93
2	2	63	107
3	3	103	166

C	R	Execution Reduction Ratio in ZLE with All	LMI	CMI	CRS	RSS
1	1	37	18	5	4	3
2	2	63	35	6	5	5
3	3	103	37	9	9	14

Table 3: **SAMC Reduction Ratio.** *The first table shows the execution reduction ratio of SAMC over black-box DPOR (bDP) in checking ZLE and ZAB under different crash/reboot budgets. "C" and "R" are the number of crashes and reboots. The second table shows the execution reduction ratio in ZLE with individiual policies over black-box DPOR (bDP).*

reboots increases. With more crashes and reboots, the ZLE protocol generates more messages and most of them are independent, and thus the LMI policy has more opportunities to remove redundant message re-orderings. Similary, the crash and reboot symmetry policies give better benefits with more crashes and reboots. The table also shows that LMI provides the most reduction. This is because the number of message events is higher than crash and reboot events (as also depicted in Table 2).

We now discuss our reduction ratio with that of DIR [25]. As discussed earlier (§2.2), DIR records local exploration (thread interleavings) and replays future incoming messages whenever possible, reducing the work of global exploration. If the target system does not have lots of thread interleavings, DIR's reduction ratio is estimated to be between 10^1 to 10^3 (§5 of [25]). As we described earlier (§2.2), DIR is orthogonal to SAMC. Thus, the reduction ratios of SAMC and DIR are complementary; when both methods are combined, there is a potential for a higher reduction ratio. The DIR authors also hinted that domain knowledge can guide dmcks (and also help their work) to both scale and hit deep bugs (§8 of [25]). SAMC has successfully addressed such need.

Finally, we note that in evaluating SAMC, we use execution reduction ratio as a primary metric. Another classical metric to evaluate a model checker is state coverage (*e.g.*, a dmck that covers more states can be considered a more powerful dmck). However, in our observation state coverage is not a proper metric for evaluating optimization heuristics such as SAMC policies. For example, if there are three nodes ABC that have the same role (*e.g.*, follower), a naive black-box dmck will crash each node and covers three distinct states: *BC, A*C and AB*. However, with a semantic-aware approach (*e.g.*, symmetry), we know that covering one of the states is sufficient. Thus, less state coverage does not necessarily imply a less powerful dmck.

6 Discussion

In this section, we discuss SAMC's simplicity, generality and soundness. We would like to emphasize that the main goal of this paper is to show the power of SAMC in finding deep bugs both quickly and systematically, and thus we intentionally leave some subtasks (*e.g.*, automated extraction, soundness proofs) for future work.

6.1 Simplicity

In previous sections, we mentioned that policies can be written in few lines of code. Besides LOC, simplicity can be measured by how much time is required to understand a protocol implementation, extract the patterns and write the policies. This time metric is unfortunately hard to quantify. In our experience, the bulk of our time was spent in developing SAMPRO from scratch and integrating policies to dmck mechanisms (§2.1). However, the process of understanding protocols and crafting policies requires a small effort (*e.g.*, few days per protocol to the point where we feel the policies are robust). We believe that the actual developers will be able to perform this process much faster than we did as they already have deeper understandings of their code.

6.2 Generality

Our policies contain patterns that are common in distributed systems. One natural question to ask is: how much semantic knowledge should we expose to dmck? The ideal case is to expose as much information as possible as long as it is sound. Since proving soundness and extracting patterns automatically are our future work, in this paper we only propose exposing high-level processing semantics. With advanced program analysis tools that can analyze deep program logic, we believe more semantic knowledge can be exposed to dmck in a sound manner. For example, LMI can be extended to include commutative modifications. This is possible if the program analysis can verify that the individual modification does not lead to other state changes. This will perhaps be the point where symbolic execution and dmck blend in the future (§7).

Nevertheless, we find that high-level semantics are powerful enough. Beyond the three cloud systems we target in this paper, we have been integrating SAMC to MaceMC [34]. MACEMC already employs random exploration policies to model check Mace-based distributed systems such as Mace-based Chord and Pastry. To integrate SAMC, we first must re-implement DPOR in

MaceMC (existing DPOR implementation in MaceMC is proprietary [25]). Then, we have written 18 lines of LMI protocol-specific rules for Chord and attain two orders of magnitude of reduction in execution. This shows the generality of SAMC to many distributed systems.

6.3 Soundness

SAMC policies only skip re-orderings and crash/reboot events that are redundant by definition, however currently our version of SAMC is not sound; the unsound phase is the manual extraction process. For example, if the tester writes a wrong predicate definition (*e.g.*, `pd`) that is inconsistent with what the target system defines, then soundness (and correctness) is broken. Advanced program analysis tools can be developed to automate and verify this extraction process and make SAMC sound. Currently, the fact that protocol-specific rules tend to be short might also help in reducing human errors. Our prototype, SAMPRO, is no different than other testing/verification tools; full correctness requires that such tools to be free of bugs and complete in checking all specifications, which can be hard to achieve. Nevertheless, we want to bring up again the discussion in Section 2.4 that dmck's scalability and ability to find deep bugs in complex distributed systems are sometimes more important than soundness. We leave soundness proofs for future work, but we view this as a small limitation, mainly because we have successfully shown the power of SAMC.

7 Related Work

We now briefly discuss more related work on dmck and other approaches to systems verification and testing.

Formal model checking foundations such as partial order reduction [17, 20], symmetry [9, 45], and abstraction [10], were established more than a decade ago. Here, classical model checkers require system models and mainly focus on state-space reduction. Implementation-level model checkers on the other hand are expected to find real bugs in addition to being efficient.

Symbolic execution is another powerful formal method to verify systems correctness. Symbolic execution also faces an explosion problem, specifically the path explosion problem. A huge body of work has successfully addressed the problem and made symbolic execution scale to large (non-distributed) software systems [3, 6, 8, 11, 55]. Symbolic execution and model checking can formally be combined into a more powerful method [4], however this concept has not permeated the world of distributed systems; it is challenging to track symbolic values across distributed nodes.

Reliability bugs are often caused by incorrect handling of failures [23, 24]. Fault-injection testing however is challenging due to the large number of possible failures to inject. This challenge led to the development of efficient fault-injection testing frameworks. For example, AFEX [1] and LFI [39] automatically prioritize "high-impact targets" (*e.g.*, unchecked system calls). These novel frameworks target non-distributed systems and thus the techniques are different than ours.

Similarly, recent work highlights the importance of testing faults in cloud systems (*e.g.*, FATE [23], SETSUDO [30], PREFAIL [31], and OpenStack fault-injector [32]). As mentioned before (§2.2), these frameworks are not a dmck; they cannot re-order concurrent messages and failures and therefore cannot catch distributed concurrency bugs systematically.

The threat of multiple failures to systems reliability already existed since the P2P era; P2P systems are susceptible to "churn", the continuous process of node joining and departing [42]. Many dmcks such as MACEMC [34] and CrystalBall [50] evaluate their approaches on P2P systems. Interestingly, we find that they mainly re-order join messages. To our understanding, based on their publications, they did not inject and control node departures. CrystalBall authors mentioned about running churns, but only as part of their workloads, not as events that the dmck can re-order. This illustrates the non-triviality of incorporating failures in dmck.

The deep bugs we presented can be considered as concurrency bugs (in distributed nature). For non-distributed systems, there has been an abundance of innovations in detecting, avoiding, and recovering from concurrency bugs [29, 33, 38, 48]. They mainly target threads. For dmck, we believe more advancements are needed to unearth distributed concurrency bugs that still linger in cloud systems.

Finally, the journey in increasing cloud dependability is ongoing; cloud systems face other issues such as bad error handling code [54], performance failures [14], corruptions [15], and many others. Exacerbating the problem, cloud systems are becoming larger and geographically distributed [37, 46, 56]. We believe cloud systems will observe more failures and message re-orderings, and therefore our work and future dmck advancements with the inclusion of failures will play an important role in increasing the reliability of future cloud systems.

8 Conclusion

Cloud systems face complex failures and deep bugs still linger in the cloud. To address present reliability challenges, dmcks must incorporate complex failures, but existing dmcks do not scale in this regard. We strongly be-

lieve that without semantic knowledge dmck hits a scalability wall. In this paper, we show a strong case that the SAMC principle can elegantly address this scalability problem. SAMC is simple and powerful; with simple semantic knowledge, we show that dmcks can scale significantly. We presented four specific reduction policies, but beyond this, we believe our work triggers the discussion of two important research questions: *what* other semantic knowledge can scale dmck and *how* to extract white-box information from the target system? We hope (and believe) that the SAMC principle can trigger future research in this space.

9 Acknowledgments

We thank Bryan Ford, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We would also like to thank Yohanes Surya and Teddy Mantoro for their support. This material is based upon work supported by the NSF (grant Nos. CCF-1321958 and CCF-1336580). The experiments in this paper were performed in the Utah Emulab[1] [49] and NMC PRObE[2] [19] testbeds, supported under NSF grants Nos. CNS-1042537 and CNS-1042543.

References

[1] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[2] Ken Birman, Gregory Chockler, and Robbert van Renesse. Towards a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, June 2009.

[3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the 2011 EuroSys Conference (EuroSys)*, 2011.

[4] J. R. Burch, E. M. Clarke, K L. McMillan, D L. Dill, and L J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems Export. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[1] http://www.emulab.net

[2] http://www.nmc-probe.org

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[9] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *10th International Conference on Computer Aided Verification (CAV)*, 1998.

[10] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 1994.

[11] Heming Cui, Gang Hu Columbia, Jingyue Wu, and Junfeng Yang. Verifying Systems Rules Using Rule-Directed Symbolic Execution. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[14] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[15] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[16] E. Allen Emerson, Somesh Jha, and Doron Peled. Combining Partial Order and Symmetry Reductions. In *3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1997.

[17] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *The 33th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[19] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX ;login:*, 38(3), June 2013.

[20] Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. volume 1032, 1996.

[21] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997.

[22] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[23] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[24] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[25] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[26] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[28] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[29] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated Concurrency-Bug Fixing. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[30] Pallavi Joshi, Malay Ganai, Gogul Balakrishnan, Aarti Gupta, and Nadia Papakonstantinou. SETSUDO : Perturbation-based Testing Framework for Scalable Distributed Systems. In *Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[31] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[32] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. On Fault Resilience of OpenStack. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[33] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[34] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[35] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[36] Thomas A. Limoncelli and Doug Hughe. LISA '11 Theme – DevOps: New Challenges, Proven Values. *USENIX ;login: Magazine*, 36(4), August 2011.

[37] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[38] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[39] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[40] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[41] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[42] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2004.

[43] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV)*, 2010.

[44] Jiri Simsa, Randy Bryant, Garth A. Gibson, and Jason Hickey. Scalable Dynamic Partial Order Reduction. In *the 3rd International Conference on Runtime Verification (RV)*, 2012.

[45] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 2010.

[46] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[47] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[48] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[49] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[50] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed. Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[51] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[52] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[53] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[54] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[55] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 2010 EuroSys Conference (EuroSys)*, 2010.

[56] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.