Why Is Random Testing Effective for Partition Tolerance Bugs?

RUPAK MAJUMDAR, Max Planck Institute for Software Systems (MPI-SWS), Germany
FILIP NIKSIC, Max Planck Institute for Software Systems (MPI-SWS), Germany

Random testing has proven to be an effective way to catch bugs in distributed systems in the presence of network partition faults. This is surprising, as the space of potentially faulty executions is enormous, and the bugs depend on a subtle interplay between sequences of operations and faults.

We provide a theoretical justification of the effectiveness of random testing in this context. First, we show a general construction, using the probabilistic method from combinatorics, that shows that whenever a random test covers a fixed coverage goal with sufficiently high probability, a small randomly-chosen set of tests achieves full coverage with high probability. In particular, we show that our construction can give test sets exponentially smaller than systematic enumeration. Second, based on an empirical study of many bugs found by random testing in production distributed systems, we introduce notions of test coverage relating to network partition faults which are effective in finding bugs. Finally, we show using combinatorial arguments that for these notions of test coverage we introduce, we can find a lower bound on the probability that a random test covers a given goal. Our general construction then explains why random testing tools achieve good coverage—and hence, find bugs—quickly.

While we formulate our results in terms of network partition faults, our construction provides a step towards rigorous analysis of random testing algorithms, and can be applicable in other scenarios.

CCS Concepts: • **Mathematics of computing** → **Combinatorics**; • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Generating random combinatorial structures*;

Additional Key Words and Phrases: distributed systems, network partition faults, random testing, probabilistic method

ACM Reference Format:
Rupak Majumdar and Filip Niksic. 2018. Why Is Random Testing Effective for Partition Tolerance Bugs?. *Proc. ACM Program. Lang.* 2, POPL, Article 46 (January 2018), 24 pages. https://doi.org/10.1145/3158134

1 INTRODUCTION

Large-scale distributed systems are difficult to build and test. On top of the non-determinism arising out of concurrent exchange of messages, these systems must account for partial failures, where components or communication can fail along the way and produce incomplete results. Fault-tolerant components are difficult to design and reason about, and usually require intricate protocols to ensure correct behavior for the global system. Even if individual components are correct, their composition may require further protocols to operate correctly under failure conditions. Thus, distributed systems are some of the most complex pieces of software. Given the critical role these

Authors' addresses: Rupak Majumdar, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, rupak@mpi-sws.org; Filip Niksic, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Str. 26, Kaiserslautern, Rheinland-Pfalz, 67663, Germany, fniksic@mpi-sws.org.

2475-1421/2018/1-ART46
https://doi.org/10.1145/3158134

systems play today, gaining high assurance of their behavior under failure conditions is a critical challenge [Lopes 2016; McCaffrey 2015].

While there is a lot of research in systematic state space exploration under faulty conditions [Alvaro et al. 2015; Deligiannis et al. 2016; Fisman et al. 2008; Gunawi et al. 2011; Kupferman et al. 2008; Leesatapornwongsa et al. 2014; McCaffrey 2015; Yang et al. 2009], practitioners mostly resort to much simpler techniques of testing with *randomly* inserted faults, and achieve remarkable success in finding bugs [Apache Hadoop 2016; Claessen et al. 2009; Izrailevsky and Tseitlin 2011; Kingsbury 2017; Scott 2016]. For example, Jepsen [Kingsbury 2017] is a framework for black-box testing of distributed systems under *partition faults*—faults that prevent portions of a system to communicate with other portions. Jepsen provides an infrastructure to set up a number of processes and exercise a system with random operations as well as randomly introduced partition faults. Using Jepsen, Kingsbury [2013] found a remarkably large number of rather subtle problems in many production distributed systems.

The success of random testing in Jepsen and similar tools presents a conundrum. On the one hand, academic wisdom asserts that random testing will be completely ineffective in finding bugs in faulty systems other than by "extremely unlikely accident": the probability that a random execution stumbles across the "right" combination of circumstances—failures, recoveries, reads, and writes—is intuitively so small that any soundness guarantee for a given coverage goal, even probabilistic, would be minuscule (many systematic testing papers start by asserting that random testing does not or should not work). On the other hand, in practice, random testing finds bugs within a small number of tests.

In this paper, we provide a theoretical understanding for the empirical success of Jepsen in exposing subtle problems. To this effect, we first introduce some test coverage notions related to network partitions and show that many bugs discovered by Jepsen can be explained in terms of these notions. For each coverage notion, we specify coverage goals, tests, and what it means for a test to cover a goal. We show that the probability of covering a single goal with a random test is not astronomically small, as might be expected; in fact, in each case the probability can be bounded from below by a sufficiently "high" constant. We then give a general combinatorial construction that uses this bound to show that a "small" set of random tests would achieve full coverage with overwhelming probability. Our construction is not specific to Jepsen: it is applicable to random testing in general, as long as there is a positive bound on the probability of covering a single goal.

Coverage notions. Most bugs in distributed systems found by Jepsen can be manifested by tests of the following nature (see Section 2 for examples). First, a small sequence of operations "sets up" the system in a special state. Then, a carefully chosen network partition separates the system into two or more blocks which cannot communicate among each other. Then, a further sequence of operations are performed, often in each of the different blocks of the partition. Finally, the partition is healed and a final set of operations are performed. This sequence (perform operations, introduce failures, perform further operations) may need to be repeated a few times at most. Depending on the nature of the network partition introduced, we introduce and study the following coverage notions.

k-Splitting. Consider a distributed application consisting of processes $a_1, \ldots, a_n$. We fix $k > 0$, and consider partitions of the network into k disjoint blocks. Intuitively, we want to test the application for all possible ways of splitting the processes into k different groups. That is, for every k processes, we would like to test what happens if these k processes cannot communicate with each other. This is formalized using the notion of *splitting coverage*. A k-partition *splits* processes $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ if these processes all end up in different blocks of the partition. The splitting coverage asks that for every choice of k processes there is a test that splits them. For example, our application could

involve two different types of quorums, and for each quorum there could be a leader (cf. Chronos in Section 2.3). A bug may occur if the two leaders are unable to communicate. Since we a priori do not know which two processes are leaders, we want tests such that for every pair (a, b) of processes, there is a test where a network failure partitions the system into two blocks such that the processes a and b cannot communicate with each other. This is an example of splitting coverage for $k = 2$. For the Jepsen bugs, values $k = 2$ and $k = 3$ sufficed for all bugs that fall under the scope of splitting coverage.

(k, l)-*Separation.* Processes in distributed applications often have a role. For example, processes may be clients, or replicas of shared state, or replicas managing consensus protocol (e.g., Zookeeper instances). Thus, a natural requirement is to ensure certain subsets of roles stay together in a split and are separated from some other subset. This is formalized using the notion of *separating coverage*: for fixed $k, l > 0$ and every pair of collections of k and l roles, separating coverage requires that the collection of k roles is separated by a partition from the collection of l roles. For example, suppose our application is a sharded system, where each shard is replicated with a replication factor $f = 3$. A bug may occur if the leading replica is separated from the two following replicas. Again, since we a priori do not know which processes form the set of replicas, and which one among them is the leader, we want tests such that every choice of one and two processes are separated by a partition. This is an example of splitting coverage for $k = 1$ and $l = 2$. Again, for bugs reported by Jepsen, values of k and l up to 3 suffice for all bugs that fall under the scope of separating coverage.

Minority isolation. In addition to separating small subsets of processes, we often wish to impose cardinality constraints. For example, it is important to cover cases in which the current leader is in the block of a partition with fewer nodes in order to force a new leader election. To study this, we introduce the notion of *minority isolation*: for each process, this coverage notion requires that the process is in the smaller block of a bipartition.

Empirically, these coverage notions capture a large class of fault tolerance bugs found by Jepsen in distributed systems. In each case, for a fixed coverage goal (a process, a set of k processes, or a pair of k and l processes) we show we can bound the probability that a random test (a bipartition or a k-partition) covers the goal (isolates, splits, or separates the processes). The bound depends on parameters k, l, but does not depend on n—the size of the system. This allows us to construct small families of tests that cover all goals with high probability by picking sufficiently many tests (we provide upper bounds) uniformly at random. Our results thus provide a theoretical justification for the effectiveness of random testing in this domain.

One caveat is the notion of a "bug." A famous result from distributed systems, called the CAP theorem [Brewer 2000; Gilbert and Lynch 2002], asserts that no distributed system can be simultaneously consistent (roughly, linearizable), available, and partition tolerant. The precise intension of CAP and guarantees provided by specific systems is a matter of considerable debate in the systems community [Brewer 2012]. For example, under partitions, a distributed database may give up availability or consistency. However, in real systems, programmers navigate a rich space of tradeoffs with many relaxed notions of availability or consistency. What constitutes a correct test oracle is usually up to the programmers of the system to decide (see, e.g., Kingsbury [2013], and discussions on his bug reports on github). In this paper, we do not specify the property used to determine whether an error has occurred: we only guarantee that all coverage goals are met. It is up to the programmer to write appropriate test oracles to check for problems.

Constructing covering families. Provided we have a positive lower bound on the probability that a random test from a space of tests meets a fixed coverage goal, we can construct a family of tests of bounded size which meets all coverage goals with high probability. Our construction uses the

probabilistic method [Alon and Spencer 2004], a technique from combinatorics that proves the *existence* of a combinatorial object without giving an explicit construction. In this technique, in order to show that a combinatorial object with certain properties exists, one argues that a randomly chosen object from a suitable probability space has the property with positive probability. Since the probability is positive, there must be at least one object—we do not know which one—with that property. If the probability of existence is sufficiently high, by repeating the construction a number of times, one can amplify the probability to make it overwhelmingly likely that a randomly sampled object has the property, even when we may not have any deterministic construction for such an object!

To see the connection to testing, assume we have a distributed system of size n, a space of coverage goals parameterized by some constants k, l assumed to be much smaller than n (e.g., coverage goals discussed above), and a space of tests (e.g., schedules of operations and partitions simulating failures in the system). A given test can cover one or more coverage goals. Our objective is to find a small *covering family*: a small set of tests that covers all the coverage goals. Here, "small" means potentially exponential in k and l, but logarithmic or linear in n. We invoke the probabilistic method to show that a small covering family exists, and that a randomly chosen small set of tests is overwhelmingly likely to be a covering family. The outline of the argument is as follows. Fix a coverage goal and suppose that a randomly chosen test satisfies the goal with probability at least $p > 0$. Then, a set of N independently chosen random tests does not cover this goal with probability at most $(1-p)^N$, and it is not a covering family with probability at most $m \cdot (1-p)^N$ (by the union bound), where $m \approx \text{poly}(n)$ is the number of coverage goals. By picking N to be $\Omega(p^{-1}\log(m))$, this probability can be made less than 1, showing that a covering family of this size exists. Moreover, for a given $\epsilon > 0$, by picking N to be $\Omega(p^{-1}(\log m - \log \epsilon))$, the probability can be made less than ϵ, showing that a family of N randomly chosen tests is a covering family with probability at least $1-\epsilon$. Thus, by running all tests from the constructed family, we are guaranteed with high probability that we have run at least one test for each coverage goal.

We summarize the main contributions of our paper.

(1) We introduce and study notions of test coverage for distributed systems with network partition faults. We show that our coverage notions can explain many different bugs found in production systems.
(2) We provide a general combinatorial construction relating the probability that a random test covers a coverage goal to the size of a random test set that is overwhelmingly likely to provide full test coverage. This result applies to any random testing methodology.
(3) Specifically, for the coverage notions introduced in Item (1), we provide explicit lower bounds on the probability using combinatorial arguments.

Together, our results imply that a small set of randomly chosen tests can already provide full coverage in distributed systems for coverage criteria empirically correlated with bugs.

While we focus on partition tolerance, our general construction provides a general technique for a rigorous treatment of random testing. We show our main construction also shows the existence of small covering families for certain random testing procedures for multi-threaded and asynchronous programs [Burckhardt et al. 2010; Chistikov et al. 2016], for feature interactions [Kuhn et al. 2010], and for VLSI circuits [Seroussi and Bshouty 1988].

2 MOTIVATING EXAMPLES

The *CAP theorem* [Brewer 2000; Gilbert and Lynch 2002] states that a distributed system running on an unreliable network cannot simultaneously satisfy **c**onsistency, **a**vailability, and **p**artition tolerance. Since the network is unreliable and network partitions do happen, at first glance this

means a system has to make a choice between consistency and availability. However, in reality the situation is far from binary—various levels of consistency and availability form a rich landscape of tradeoffs, which system developers have to navigate knowingly or unknowingly. While doing so, they inevitably encounter pitfalls.

While there are certainly intricate pitfalls that are exposed only in extremely rare combinations of events and failures, many problems in existing systems are discoverable by random testing. An evidence to this is a series of online articles by Kingsbury [2017], describing in-depth analyses of distributed systems within a testing framework called Jepsen.[1] Using Jepsen, Kingsbury has analyzed and discovered issues in a whole range of distributed systems: etcd, Postgres, Redis, Riak, MongoDB, Cassandra, Kafka, RabbitMQ, Consul, Elasticsearch, Aerospike, Zookeeper, and Chronos, to name a few. In each case, the approach is similar: a system under scrutiny is subjected to random sequences of operations under failure modes such as random network partitions. The recorded behavior of the system is then analyzed against a model to establish its correctness. The Jepsen framework provides a scripting framework to define operations, failure modes, correctness conditions, and checkers specific to a particular system. While setting up all these things for a given system need not be simple and often requires a lot of intuition and understanding of the system, in most cases, the test, once set up, uncovers subtle issues within seconds, even with random sequences of operations and partitions.

In the rest of the section, we showcase a few typical bugs found by Kingsbury and use them to motivate notions of coverage we study in Section 4.

2.1 Etcd

Etcd is a distributed key-value store.[2] It is intended to be used for storing small amounts of critial data, which a complex distributed application might need for service coordination, distributed locking, write barriers etc. Hence, etcd's foremost design goal is to provide strong consistency. To achieve this, operations are commited through consensus, for which etcd uses an implementation of the Raft algorithm [Ongaro and Ousterhout 2014].

Since consensus requires communication between nodes, it may be unachievable while the network is partitioned. Hence, strong consistency in this case comes at the expense of reduced availability. In order to improve availability, in etcd's API version 2, read operations by default do not go through consensus. Instead, a node responds to a read request by simply returning the local copy of the value. This design choice is an example of a tradeoff between consistency and availability.

But what exactly is the manifestation of this tradeoff on the consistency side? We can check this with a Jepsen test. We set up etcd on five nodes and start five clients to issue random read, write, and compare-and-swap operations on a single key. Additionally, we start a special process (called *nemesis* in Jepsen) to randomly partition the network into two blocks to simulate network failures. We record a history of execution, and analyze it for linearizability—each operation should give an appearance of being executed atomically between its invocation and completion.

Figure 1 (left) shows an inconsistent state found by Jepsen. The arrows "w i" and "r" refer to client requests to write the value i or to read the shared state, respectively (for simplicity, we omit the key), and "w ok" and "r i" denote the system responses confirming the write and returning the value read, respectively. The blue rectangle shows the Raft consensus. In the picture, time flows downward, and the values show each node's own view of the shared state. The red line marks a network partition that separates n_1 and n_4 from n_2, n_3, and n_5. A write of the value 1 after the

[1] http://jepsen.io/
[2] https://github.com/coreos/etcd

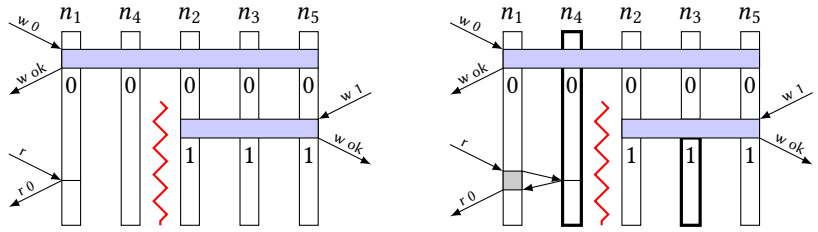

Fig. 1. Non-linearizable execution histories in etcd. On the left, the read operation is invoked in the default mode—returning the local copy of the value—and on the right it is invoked in the consistent mode—redirecting the request to the Raft leader.

partition triggers a new leader election in Raft, after which the new value is committed (by the right block). The inconsistent behavior is that after the partition, the node n_1 returns its local stale value 0 even though it is not part of the quorum.

The developers of etcd were aware of this behavior. To give users stronger consistency, the read API provides an option called *consistent*, which causes nodes to redirect read requests to the leader elected as part of the Raft consensus algorithm. This option is another example of navigating the CAP landscape: we seemingly avoid both the overhead of full consensus and the inconsistent behavior. Unfortunately, the same Jepsen test setup quickly discovers another inconsistency, shown in Fig. 1 (right). Assume that n_4 was the Raft leader to begin with. The write request w 0 is committed. At this point, the network partition separates n_1 and n_4 from the rest. Realizing that the leader is unavailable, nodes in the larger block elect n_3 as the new leader and successfully commit a new write of the value 1. Nodes n_1 and n_4 are unaware of the new leader election. Thus, n_1 forwards a read request to n_4, which returns the stale value 0. The problem is still that reads do not require consensus, so the smaller block does not realize the leader has changed.

Since the author of Jepsen reported this behavior[3], the developers of etcd have included another option for read called *quorum*. With this option, reads are commited through consensus the same way writes are. More recently, consensus for all operations has become the default behavior in the etcd API version 3.

2.2 Kafka

Kafka[4] is a distributed streaming system: it provides streams of records to which clients can publish or subscribe. To achieve scalability and fault-tolerance, records in Kafka's streams are partitioned into shards, and each shard is replicated by a set of in-sync replicas (ISR). Within an ISR, one node is designated as a leader: for leader election, Kafka uses Zookeeper[5], a distributed key-value store with strong consistency guarantees similar to etcd. The leader accepts write requests from clients and forwards them to all replicas in the ISR. When it receives acknowledgements from all replicas, it acknowledges the client. If a replica fails to acknowledge a request, the leader detects the request has timed out, and removes that node from the ISR. Remaining writes only have to be acknowledged by the healthy nodes still in the ISR. Should the leader ever fail, any replica in the ISR can take over, since all of them maintain the same history of records. In this way, Kafka can theoretically tolerate $f - 1$ failures with f replicas.

[3] https://github.com/coreos/etcd/issues/741
[4] https://kafka.apache.org/
[5] http://zookeeper.apache.org/

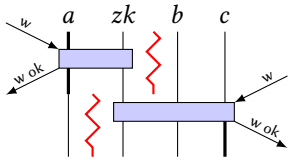

Fig. 2. A diagram of an inconsistent behavior in Kafka.

Note that with up to $f - 1$ failures Kafka provides both linearizability—all nodes in the ISR replicate records in the same order—and high availability—unresponsive nodes are automatically removed from the ISR. According to the CAP theorem, Kafka has to give up partition tolerance. The following Jepsen test, reported by Kingsbury (see Fig. 2), shows not only that Kafka gives up partition tolerance, but that in presence of network partitions a *single* node failure can cause writes committed to the system to be lost. In a system consisting of a Zookeeper node and three in-sync-replicas a, b, c, with a being the leader, nodes b and c are separated from a and the Zookeeper node. Realizing that nodes b and c are unavailable, a shrinks the ISR to just itself, and continues processing writes from the client. Node a then crashes (simulated by a partition that separates a from all other nodes), and the system enters a state called unclean leader election, in which any of the nodes b and c (whichever comes to life first) can be elected as a new leader. However, once the new leader starts processing writes, all intermediate writes processed by node a are lost.

2.3 Chronos

The third and most complex bug is from Chronos[6], a distributed and fault-tolerant job scheduler. Chronos is meant to schedule jobs with complicated dependency and periodicity specifications at the correct times. It is used in conjunction with Mesos[7]—a cluster management system that takes care of managing resources such as CPU and memory in a cluster. Both systems further depend on Zookeeper as a consistent data store.

There are three kinds of quorums involved in the system. First, Chronos has to maintain consensus over job scheduling, so Chronos nodes have a notion of leader and followers. Second, Mesos nodes are divided into "slaves," which offer resources and ultimately run jobs, and "masters," which take care of resource and job allocation. The latter also requires consensus, so Mesos masters have their own notion of leader and followers. And third, Zookeeper is a consistent data store and needs to maintain consensus over executed operations, so it also has a notion of leader and followers.

All three kinds of leaders need to be able to communicate. Chronos and Mesos need Zookeeper for their internal coordination, but also for mutual discovery. It is thus not difficult to imagine partitions among these nodes may cause problems.

Kingsbury tested the system on five nodes $n_1, \ldots, n_5$. All five serve at the same time as Chronos and Zookeeper nodes. Additionally, nodes n_2, n_3, and n_4 are Mesos masters and the remaining two are Mesos slaves. During the test, simple jobs are generated and emitted, while the nemesis randomly partitions the network into two blocks and heals it back after some time. At the end, a checker examines the history to see whether all jobs were executed at correct time.

Not suprisingly, as soon as the Chronos leader is separated from the Zookeeper leader, problems start. In this case Chronos sometimes abruptly crashes, which turns out to be an undocumented but expected behavior[8]—the programmers considered this to be a conservative way of preventing

[6] https://mesos.github.io/chronos/
[7] http://mesos.apache.org/
[8] https://github.com/mesos/chronos/issues/513

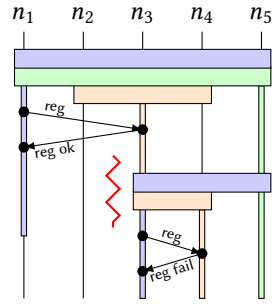

Fig. 3. A diagram of a bug in Chronos that prevents jobs from being executed.

inconsistencies while Zookeeper is unavailable. Suprisingly, however, in the same situation Chronos sometimes *does not* crash for unknown reasons.[9] This behavior is marked as a bug and was unresolved at the time this paper was written.

The case when Chronos does not crash uncovered another bug[10] that has meanwhile been fixed. The observed execution is illustrated in Fig. 3. Initially all three subsystems—Chronos, Mesos, and Zookeeper—go through leader election. Node n_1 becomes Chronos leader (blue timeline), node n_3 becomes Mesos leading master (orange timeline), and node n_5 becomes Zookeeper leader (green timeline). Chronos registers a new framework with Mesos—this is depicted by an exchange of messages "reg" and "reg ok" between n_1 and n_3—and starts scheduling jobs (omitted for simplicity). Next, a network partition separates n_1 and n_2 from n_3, n_4, and n_5. Chronos leader detects Zookeeper connection loss, but does *not* crash. After another round of elections, n_3 becomes Chronos leader, and n_4 becomes Mesos leading master. The partition is resolved, n_1 detects the Zookeeper leader and recognizes n_3 as the new Chronos leader.

At this point Chronos tries to register as a completely new framework with Mesos, instead of re-registring as the original framework. Since according to Mesos all resources in the cluster are owned by the original framework, there are no available resources for the new framework, and as a consequence no new jobs are ever started again.

2.4 Summary and Coverage Notions

In each of these examples, the test setup to manifest the misbehavior involved: setting up and running the system with a number of nodes and clients; running a random sequence of operations (reads, writes, etc.) to put the system in a specific state; introducing one or more carefully orchestrated partitions of the network; running a further sequence of operations, and optionally, further partitions of the network to demonstrate a violation. Jepsen provides an API to programmatically manage all these steps. In most Jepsen experiments, though, picking each of these steps *randomly* demonstrated the misbheaviors.

While the original Jepsen traces contain hundreds or thousands of events, the analysis of Kingsbury shows that for most bugs one requires a small number of "rounds," consisting of a small number of operations and one or two carefully chosen partitions in the system. In particular, the bugs in Chronos were exposed with a partition separating the Chronos leader from the Zookeeper leader. The inconsistency in Kafka was exposed with a partition separating node a and the Zookeeper node from nodes b and c. The inconsistencies in etcd were exposed with a partition isolating the Raft

[9] https://github.com/mesos/chronos/issues/522
[10] https://github.com/mesos/chronos/issues/520

leader in a smaller block, while the system received a write request in the larger block, followed by a read request in the smaller block.

Accordingly, we study the following coverage notions.

k-splitting Given a set of k nodes, the coverage goal is to *split* them with a network partition, that is, place each node in a separate block of the partition. The coverage problem is to split every set of k nodes with a family of partitions.

k,l-separation Given two sets of nodes of size k and l, the coverage goal is to *separate* them with a network partition, that is, place the set of k nodes in one block and the set of l nodes in another block. The coverage problem is to separate every pair of sets of size k and l with a family of partitions.

minority isolation Given a node x in a system with n nodes, the coverage goal is to *isolate* x in a *minority* block—a block containing less than $n/2$ nodes. The coverage problem is to isolate every element in a minority block with a family of partitions.

In addition to these coverage notions relating to network partitions, we require a set of *operations* to occur before or after the partition. Thus, we also study the following notion:

sequences of operations Given a sequence of k operations, the coverage problem is to observe this sequence as a contiguous subsequence of a larger sequence.

Our goal is to combine partition coverage with coverage w.r.t. short sequences of operations.

The test exposing the bugs in Chronos is an example of k-splitting with $k = 2$, but also of k,l-separation with $k = l = 1$, as the two notions overlap in this case. The test exposing the inconsistency in Kafka is an example of a sequenced k,l-separation: we need to observe a $2,2$-separating partition followed by a $1,3$-separating partition. Finally, the test exposing the inconsistencies in etcd is an example of minority isolation combined with a sequence of two operations. In each case, the separation is accompanied with short sequences of operations (e.g., reads or writes). Our results show that in each of these cases a small number of randomly generated partitions and operations achieves full coverage with overwhelming probability.

It is worth mentioning that our coverage notions are not specifically tied to the three systems we picked as examples. In fact, almost all anomalies discovered and described by Kingsbury fall under one or more of our coverage notions. There is, however, an exception: a split-brain issue in Elasticsearch[11] involving an *intersecting* partition—one in which blocks are not disjoint. In this particular issue, a node acts as a bridge between two otherwise disconnected blocks and facilitates a situation in which two nodes on the opposite sides of the partition become leaders and start processing write requests from clients. Note that an intersecting partition $\{X, Y\}$, where $Z = X \cap Y$ is nonempty, can be modeled by a partition $\{X \setminus Z, Z, Y \setminus Z\}$. Therefore, the Elasticsearch issue would fall under a straightforward generalization of k,l-separation with an additional block, and our techniques would apply. We omit this generalization for simplicity.

3 A GENERAL CONSTRUCTION

We first state and prove a general theorem on test coverage. We leave the notions of coverage goals, tests, or the notion of covering abstract—we will instantiate these notions in specific cases.

Let M be a nonempty set of *coverage goals*. Let $\mathcal{T}$ be a set of *tests*. A test $t \in \mathcal{T}$ may or may not *cover* a coverage goal. A nonempty set $\mathcal{F}$ of tests is a *covering family* for M if for each $x \in M$, there is a test $t \in \mathcal{F}$ such that t covers x.

Theorem 3.1. *Let M be a set of m coverage goals. Let $p > 0$ be a lower bound on the probability that a random test $t \in \mathcal{T}$ covers a fixed coverage goal. Given $\epsilon > 0$, let $\mathcal{F}$ be a family of tests chosen*

[11] https://github.com/elastic/elasticsearch/issues/2488

independently and uniformly at random such that $|\mathcal{F}| \geq p^{-1}(\log m - \log \epsilon)$. *Then* $\mathcal{F}$ *is a covering family with probability at least* $1 - \epsilon$. *Moreover, there exists a covering family of size* $\lceil p^{-1} \log m \rceil$ *(or 1 if* $m = 1$*).*

Proof. Consider a fixed coverage goal x. A random test does not cover x with probability at most $1 - p$. Since the tests in $\mathcal{F}$ are chosen independently, the probability that $\mathcal{F}$ does not cover x is at most $(1-p)^{|\mathcal{F}|}$. By the union bound, the probability that there exists a coverage goal not covered by $\mathcal{F}$ is at most $m(1-p)^{|\mathcal{F}|}$.

If $m > \epsilon$, using $|\mathcal{F}| \geq p^{-1}(\log m - \log \epsilon)$ and the fact that $p < -\log(1-p)$, we get

$$|\mathcal{F}| > \frac{-\log m + \log \epsilon}{\log(1-p)} = \log_{1-p}(m^{-1}\epsilon) \ .$$

Note that this trivially holds if $m \leq \epsilon$. Therefore, in both cases $m(1-p)^{|\mathcal{F}|} < \epsilon$, and the probability that $\mathcal{F}$ covers all coverage goals is at least $1 - \epsilon$. In particular, if we take $\epsilon = 1$, the probability that $\mathcal{F}$ covers all objects is positive. By the probabilistic method, there must exist a covering family of size $\lceil p^{-1} \log m \rceil$, or 1 if $m = 1$, since a covering family needs to be nonempty. □

There is always a trivial covering family of size $|M|$ (assuming each coverage goal can be covered by some test). The key observation in Theorem 3.1 is that there exist covering families of size proportional to $\log|M|$; thus, if we can show the probability p is "high," we can get an exponentially smaller covering family of tests. Moreover, a randomly chosen test set can cover all goals with high probability.

Before diving into Section 4 where we analyze network partitions, let us demonstrate how Theorem 3.1 can be used to analyze the coverage notion involving sequences of operations motivated in Section 2.4. Suppose we have $r \geq 1$ different operations, and we are generating a sequence of operations s uniformly at random. Suppose we also have a set T of target sequences of length $k \geq 1$, and we want to observe any target sequence $t \in T$ as a contiguous subsequence of s.

Theorem 3.2. *Let* $\epsilon > 0$, *let* T *be a set of sequences of operations of length* $k \geq 1$, *and let* s *be a sequence of* $n \geq 1$ *operations chosen independently and uniformly at random such that* $n \geq k - kr^k|T|^{-1}\log\epsilon$. *Then some target sequence* $t \in T$ *is a contiguous subsequence of* s *with probability at least* $1 - \epsilon$.

Proof. Split the sequence s into $\lfloor n/k \rfloor$ non-overlapping subsequences of length k. The probability that some $t \in T$ occurs among these subsequences is clearly lower than the probability that some $t \in T$ occurs in s. However, we can think of the non-overlapping sequences as $\lfloor n/k \rfloor$ sequences of length k generated independently and uniformly at random.

The probability that one of these sequences matches a sequence in T is $p = |T|/r^k$. The number of coverage goals in this case is $m = 1$, namely any target sequence $t \in T$. Since $\lfloor n/k \rfloor > n/k - 1$, we have

$$\lfloor n/k \rfloor > -r^k|T|^{-1}\log\epsilon = p^{-1}(\log m - \log \epsilon) \ .$$

Hence the result follows from Theorem 3.1. □

Example 3.3. Consider the inconsistency in etcd on Fig. 1 (left). In order to expose it, we need to observe a right combination of read and write operations during a network partition with two blocks. Note that it does not really matter how the network is partitioned, as long as we follow up with a write that changes the value in the larger block, and a read to any node in the smaller block. For simplicity, assume we only have three kinds of operations—read, write 0, and write 1—and each can be directed to any of the five nodes involved in the experiment. Thus, in total we have

15 operations, each occurring with equal probability.[12] We form an amalgamated operation by conjoining a partition with two read or write operations. Assuming partitions are balanced, this gives us a total of $r = 10 \cdot 15 \cdot 15 = 2250$ amalgamated operations. Our target operations consist of a partition followed by one of two read operations and one of three write operations. Thus the set T of target amalgamated operations has size $10 \cdot 2 \cdot 3 = 60$. Applying Theorem 3.2 with $k = 1$, we see that in a sequence of 61 amalgamated operations, we would observe a target operation—and thus expose the inconsistency—with probability at least 80%. □

4 RANDOM PARTITIONS

In this section we study the coverage notions introduced in Section 2.4 as instances of the general construction from Section 3. We start with combinatorial preliminaries about set partitions.

4.1 Combinatorial Preliminaries

Throughout this section, let $U = \{1, \ldots, n\}$ be a fixed set (a "universe") of n elements. A *partition* of U is a set of nonempty subsets of U that are pairwise disjoint and in the union give the whole U. We refer to the sets in a partition as *blocks*. If a partition has k blocks, we call it a k-partition. A *balanced partition* is a partition with blocks differing in size at most by 1.

Let us recall a few results about partitions. The number of k-partitions is given by a quantity called *Stirling number of the second kind*, denoted $\left\{ {n \atop k} \right\}$ and read "n subset k" [Graham et al. 1994]. It is not difficult to see that $\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1$ whenever $n \geq 1$. Moreover, $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$, as a 2-partition is uniquely determined by the block that does not contain the nth element, and this block needs to be nonempty. In general, Stirling numbers of the second kind satisfy the following recurrence:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\} . \tag{1}$$

Combinatorially, we can partition n elements into k blocks by partitioning the first $n-1$ elements into $k-1$ blocks and adding a singleton block consisting of the nth element, or by partitioning the first $n-1$ elements into k blocks and placing the nth element into one of these blocks in k ways.

Lemma 4.1. *For every $n \geq 1$ and k such that $1 \leq k \leq n$, we have*

$$\left\{ {n \atop k} \right\} k! \leq k^n .$$

Proof. The quantity on the right-hand side is the number of all functions from an n-element set to a k-element set, while the quantity on the left-hand side is the number of such functions that are *surjective*. To see this, note that a surjection induces a k-partition of the domain, and the induced blocks map to the codomain in one of $k!$ ways. □

For a fixed k, $\left\{ {n \atop k} \right\}$ asymptotically approaches $k^n/k!$. Intuitively, if we randomly assign n elements into k buckets and n is large, it is unlikely one of the buckets will be empty. Therefore, the difference between the left-hand side and right-hand side in Lemma 4.1 will be small.

4.2 Splitting Families

We formalize the notion of k-splitting from Section 2.4 using k-splitting families.

[12] In the real Jepsen experiment, clients were also issuing compare-and-swap operations with values ranging from 1 to 5, and the probability distribution over operations was not uniform.

Definition 4.2. Given k, let P be a k-partition of U and let $S = \{x_1, \ldots, x_k\} \subseteq U$. We say P *splits* S if $P = \{B_1, \ldots, B_k\}$ and $x_1 \in B_1, \ldots, x_k \in B_k$. We say a family $\mathcal{F}$ of k-partitions is a *k-splitting family* if for every subset $S \subseteq U$ there is a partition in $\mathcal{F}$ that splits S.

As the following theorem shows, for a given fixed set S of k elements, the probability that a random k-partition splits S is bounded from below by a constant that depends only on k.

Theorem 4.3. *Let $S \subseteq U$ be a set of k elements, and let p be the probability that a random k-partition splits S. Then $p = k^{n-k} / \left\{ {n \atop k} \right\} \geq k^{-k} k!$.*

Proof. A k-partition that splits S is uniquely determined by a map $U \setminus S \to S$ that maps $x \in U \setminus S$ to $y \in S$ if x and y are in the same block of the partition. Hence, the probability that a random k-partition splits S is $p = k^{n-k} / \left\{ {n \atop k} \right\}$. From Lemma 4.1 it follows that $p \geq k^{-k} k!$. □

Thus, splitting families are a special case of the general construction from Section 3: a coverage goal here is a subset of U of size k, a test is a partition of U with k blocks, and a covering family is a k-splitting family. We have the following corollary as an instance of Theorem 3.1.

Corollary 4.4. *Let $\epsilon > 0$ and let $\mathcal{F}$ be a family of k-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq k^{k+1}(k!)^{-1} \log n - k^k (k!)^{-1} \log \epsilon$. Then $\mathcal{F}$ is k-splitting with probability at least $1 - \epsilon$. Moreover, there exists a k-splitting family of size $\left\lceil k^{k+1}(k!)^{-1} \log n \right\rceil$.*

Proof. By Theorem 4.3, a fixed subset of U of size k is split by a random k-partition with probability $p \geq k^{-k} k!$. Moreover, the number of subsets of U of size k is $m = \binom{n}{k} \leq n^k$. Therefore, $|\mathcal{F}| \geq p^{-1}(\log m - \log \epsilon)$, and the result follows from Theorem 3.1. □

In the case of $k = 2$, the most common case in Jepsen, we can get a more precise bound.

Corollary 4.5. *Let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 2 \log_2 n - \log_2 \epsilon - 1$. Then $\mathcal{F}$ is 2-splitting with probability at least $1 - \epsilon$. Moreover, there exists a 2-splitting family of size $\lceil 2 \log_2 n - 1 \rceil$.*

Proof. We get this slightly more precise bound by performing a more precise version of the analysis from the proof of Theorem 3.1. Like there, we can bound the probability that $\mathcal{F}$ is not 2-splitting by $m(1-p)^{|\mathcal{F}|}$, with $m = \binom{n}{2}$ and $p = 2^{n-2} / \left\{ {n \atop 2} \right\}$. However, since $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$, we have $1 - p < 1/2$ whenever $n \geq 2$. Hence, $m(1-p)^{|\mathcal{F}|} < m 2^{-|\mathcal{F}|}$. On the other hand, since $m = \binom{n}{2} \leq n^2/2$, we have $|\mathcal{F}| \geq 2 \log_2 n - \log_2 \epsilon - 1 \geq \log_2(m/\epsilon)$. Hence, $m(1-p)^{-|\mathcal{F}|} < m 2^{-|\mathcal{F}|} \leq \epsilon$. □

Remark 4.6. Note that the probabilistic construction in Corollary 4.5 is sub-optimal and that there is a deterministic construction of a 2-splitting family of size $\lfloor \log_2 n \rfloor + 1$. To see this, take the binary representation of the elements in the universe. For each position $i \in \{0, \ldots, \lfloor \log_2 n \rfloor\}$, consider the partition obtained by separating all elements which have a 0 in the ith position from all elements which have a 1 in the ith position. Clearly, this set of $\lfloor \log_2 n \rfloor + 1$ partitions forms a 2-splitting family. □

Corollary 4.4 suggests that we can get k-splitting families with high probability by generating sufficiently many (logarithmically in n) k-partitions uniformly at random. But how do we generate a k-partition uniformly at random? We can do it recursively using the basic recurrence for Stirling numbers (1). The base cases are $k = 1$ (we generate a single block containing n elements) and $k = n$ (we generate n singleton blocks). Otherwise we choose between partitioning the first $n-1$ elements recursively into $k-1$ or k blocks with the following respective probabilities:

$$\left\{ {n-1 \atop k-1} \right\} \Big/ \left\{ {n \atop k} \right\} \quad \text{and} \quad k \left\{ {n-1 \atop k} \right\} \Big/ \left\{ {n \atop k} \right\} . \tag{2}$$

In the former case we add to the $k-1$ blocks a singleton block consisting of the nth element, and in the latter case we place the nth element into one of the k blocks uniformly at random.

It is not difficult to see that the described procedure indeed gives us k-partitions uniformly at random. However, the intermediate probabilities (2) involve computing Stirling numbers, which grow exponentially in n. Fortunately, due to the following theorem, it suffices to restrict ourselves to balanced partitions, which are much easier to generate uniformly at random: we generate a random permutation of the set U and split it into k balanced blocks.

Theorem 4.7. *Let $S \subseteq U$ be a set of k elements, let p be the probability that a random k-partition splits S, and let p_b be the probability that a random balanced k-partition splits S. Then $p_b \geq p$.*

In order to prove Theorem 4.7, we need an auxiliary combinatorial lemma of independent interest.[13]

Lemma 4.8. *Let $n, k \in \mathbb{N}$, and let $m = n \bmod k$. Then,*

$$k^{n-k}\binom{n}{k} \leq \left\lceil \frac{n}{k} \right\rceil^m \left\lfloor \frac{n}{k} \right\rfloor^{k-m} \left\{ {n \atop k} \right\} .$$

Proof. Let M be a binary matrix whose rows are indexed by k-partitions and columns by subsets of U of size k, and such that an entry corresponding to partition P and set S is 1 if and only if P splits S. We count the number of ones in the matrix in two different ways.

The number of ones in a column indexed by set S is the number of k-partitions that split S. As argued in the proof of Theorem 4.3, this number is k^{n-k}. Hence the total number of ones in M is $k^{n-k}\binom{n}{k}$. On the other hand, the number of ones in a row indexed by partition $P = \{B_1, \ldots, B_k\}$ is the number of sets split by P. It is not difficult to see this number is $|B_1| \cdots |B_k|$. Summing over all k-partitions, we get

$$k^{n-k}\binom{n}{k} = \sum_{P=\{B_1,\ldots,B_k\}} |B_1| \cdots |B_k| \tag{3}$$

Note that the product $B = |B_1| \cdots |B_k|$ attains its maximal value for a balanced partition. For suppose the partition is not balanced; then there are blocks B_i and B_j such that $|B_i| - |B_j| \geq 2$. From these we obtain blocks B_i' and B_j' by moving an arbitrary element from B_i to B_j. Let $B' = B/(|B_i||B_j|)$; for the new product we have:

$$\begin{aligned} B'|B_i'||B_j'| &= B'(|B_i| - 1)(|B_j| + 1) \\ &= B'(|B_i||B_j| + |B_i| - |B_j| - 1) \\ &> B'|B_i||B_j| \\ &= B \end{aligned}$$

We have thus increased the value of the product.

It is not difficult to see that a balanced partition has m blocks of size $\lceil n/k \rceil$ and $k-m$ blocks of size $\lfloor n/k \rfloor$. Hence the maximal value of the product $|B_1| \cdots |B_k|$ is $\lceil n/k \rceil^m \lfloor n/k \rfloor^{k-m}$. With this we can bound the sum in (3) and complete the proof:

$$\sum_{P=\{B_1,\ldots,B_k\}} |B_1| \cdots |B_k| \leq \left\lceil \frac{n}{k} \right\rceil^m \left\lfloor \frac{n}{k} \right\rfloor^{k-m} \left\{ {n \atop k} \right\} \qquad \square$$

[13] Independently of the authors, a weaker variant of Lemma 4.8 was posted as Problem 11957 in the February 2017 issue of the American Mathematical Monthly [Edgar et al. 2017]. The proof given here solves the problem.

Proof of Theorem 4.7. We already know that $p = k^{n-k} / \left\{ {n \atop k} \right\}$. To calculate p_b, we need to calculate the number N_S of balanced k-partitions that split S, and the number N of all balanced k-partitions; then $p_b = N_S/N$.

Let $m = n \bmod k$. As noted earlier, a balanced k-partition has m blocks of size $\lceil n/k \rceil$, and $k - m$ blocks of size $\lfloor n/k \rfloor$. In order to uniquely determine a balanced k-partition that splits S, we first choose m elements of S that are placed in larger blocks, and then for each element of S we choose a completion of its block from $U \setminus S$. Thus,

$$N_S = \binom{k}{m} \binom{n-k}{\underbrace{\lceil n/k \rceil - 1, \ldots, \lceil n/k \rceil - 1}_{m \text{ times}}, \underbrace{\lfloor n/k \rfloor - 1, \ldots, \lfloor n/k \rfloor - 1}_{k-m \text{ times}}} .$$

Similarly, in order to uniquely determine a balanced k-partition, we choose blocks of appropriate sizes, and account for the fact that neither the order of larger nor the order of smaller blocks matters. Thus,

$$N = \frac{1}{m!(k-m)!} \binom{n}{\underbrace{\lceil n/k \rceil, \ldots, \lceil n/k \rceil}_{m \text{ times}}, \underbrace{\lfloor n/k \rfloor, \ldots, \lfloor n/k \rfloor}_{k-m \text{ times}}} .$$

After expanding the multinomial coefficients and rearranging the terms, we get

$$p_b = \left\lceil \frac{n}{k} \right\rceil^m \left\lfloor \frac{n}{k} \right\rfloor^{k-m} \Big/ \binom{n}{k} .$$

Hence, the inequality $p_b \geq p$ is precisely the inequality in Lemma 4.8. □

Example 4.9. Let us get back to the Chronos example from Section 2. In order to expose the bug, we had to split the Chronos leader from the Zookeeper leader in the set of five nodes. Corollary 4.5 tells us it is possible to do this with just 4 randomly generated partitions. Moreover, by generating just 2 additional partitions, we ensure the probability of the split is at least 80%. The optimal 2-splitting family in this case contains 3 partitions, and can be constructed explicitly (not randomly!) by the construction in Remark 4.6. □

Historical note. Splitting families appear in the context of perfect hashing [Czech et al. 1997; Fredman et al. 1984; Yao 1981]. Andrew Chi-Chih Yao calls them k-separating systems, and gives an explicit construction of such systems of size $4^{k^2} (\log_2 n)^{k-1}$ [Yao 1981]. He also references personal communication with Ronald Graham for a probabilistic construction of size $e^k \sqrt{k} \log n$. Another reference to personal communication with Ronald Graham appears in Fredman et al. [1984]. It is very likely that Graham's construction is similar to the one given here.

4.3 Separating Families

We now turn to the notion of k,l-separation from Section 2.4, formalized using k,l-separating families. In this subsection we fix two positive integers k,l such that $k + l \leq n$.

Definition 4.10. Let $\mathcal{F}$ be a family of 2-partitions. We call $\mathcal{F}$ a *k,l-separating family* if for every $S = \{x_1, \ldots, x_k\} \subseteq U$ and $T = \{y_1, \ldots, y_l\}$ there is a partition $P = \{X, Y\} \in \mathcal{F}$ such that $S \subseteq X$ and $T \subseteq Y$.

As with splitting families, here we can also bound the probability that a random 2-partition separates two fixed sets of size k and l, and the bound depends only on k and l.

Theorem 4.11. *Let $S, T \subseteq U$ be sets of k and l elements, and let p be the probability that a random 2-partition separates S and T. Then $p = 2^{n-k-l} / \left\{ {n \atop 2} \right\} \geq 2^{1-k-l}$.*

Proof. A 2-partition that separates S and T is uniquely determined by a map $U \setminus (S \cup T) \to \{0,1\}$ that maps $x \in U \setminus (S \cup T)$ to 1 if and only if x is in the block of the partition that contains S. Hence, the probability that a random 2-partition separates S and T is $p = 2^{n-k-l} / \left\{ {n \atop 2} \right\}$. From the fact that $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$ it follows that $p \geq 2^{1-k-l}$. □

Thus, separating families are a special case of the general construction from Section 3. A coverage goal here is a pair (S,T) of disjoint sets $S,T \subseteq U$ such that $|S| = k$ and $|T| = l$. A test is a partition of U with two blocks, and a covering family is a k,l-separating family. We have the following corollary as an instance of Theorem 3.1.

Corollary 4.12. *Let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \geq 2^{k+l-1}(k+l)\log n - 2^{k+l-1}\log\epsilon$. Then $\mathcal{F}$ is k,l-separating with probability at least $1-\epsilon$. Moreover, there exists a k,l-separating family of size $\left\lceil 2^{k+l-1}(k+l)\log n \right\rceil$.*

Proof. By Theorem 4.11, the probability that a random 2-partition separates two subsets $S,T \subseteq U$ of size k and l is $p \geq 2^{1-k-l}$. Moreover, the number of pairs of subsets (S,T) of size k and l is $m = \binom{n}{k,l} \leq n^{k+l}$. Therefore, $|\mathcal{F}| \geq p^{-1}(\log m - \log\epsilon)$, and the result follows from Theorem 3.1. □

Example 4.13. Let us get back to the Kafka example from Section 2. By plugging $k = l = 2$ and $n = 4$ into Corollary 4.12, we see that we can separate node a and the Zookeeper node from nodes b and c with approximately 45 randomly generated partitions. We can get a more precise bound by using Theorem 3.1 directly with $p = 1/8$ and $m = \binom{4}{2,2} = 6$; this tells us separation is possible with 15 randomly generated partitions.

Of course, the separation of node a and the Zookeeper node from nodes b and c does not expose the inconsistency on its own—we need two consecutive partitions, one being 2,2-separating, and another being 1,3-separating. Suppose we alternate partitions with blocks of size 2 and 2 (first phase), and partitions with blocks of size 1 and 3 (second phase). Fix a pair of nodes (x,y)—x representing the Zookeeper node, and y representing node a. The probability of x and y ending up in the same block in the first phase is $1/3$, and the probability of y being isolated in the second phase is $1/4$, giving an overall probability of $1/12$. The number of pairs (x,y) in a set of four nodes is 12. Thus by invoking Theorem 3.1 directly, we get that we can expose the inconsistency with 30 alternations of the two phases, and we can expose it with probability at least 80% with 50 alternations. □

4.4 Minority Isolating Families

The next notion to analyze is the one of minority isolation. We formalize it using minority isolating families.

Definition 4.14. Let $\mathcal{F}$ be a family of 2-partitions. We call $\mathcal{F}$ a *minority isolating family* if for every $x \in U$ there is a partition $P = \{X,Y\} \in \mathcal{F}$ such that $x \in X$ and $|X| < |Y|$.

The analysis of minority isolating families depends on whether n is odd or even. The two cases differ slightly because if n is odd, there is a smaller block in every 2-partition, while if n is even, we can split the set into two blocks of equal size. We first analyze the case when n is odd.

Denote by p the probability that a random 2-partition isolates a fixed element x in the smaller (minority) block. By summing over the size of the block containing x, we have:

$$p = \sum_{0 \le j < \lfloor n/2 \rfloor} \binom{n-1}{j} \Big/ \left\{ n \atop 2 \right\}$$

$$1 - p = \sum_{\lfloor n/2 \rfloor \le j < n-1} \binom{n-1}{j} \Big/ \left\{ n \atop 2 \right\} = \sum_{0 < j \le \lfloor n/2 \rfloor} \binom{n-1}{j} \Big/ \left\{ n \atop 2 \right\}$$

Subtracting the first equality from the second one gives us:

$$1 - 2p = \left(\binom{n-1}{\lfloor n/2 \rfloor} - 1 \right) \Big/ \left\{ n \atop 2 \right\} .$$

Using $\left\{ n \atop 2 \right\} = 2^{n-1} - 1$, we get:

$$p = \left(2^{n-1} - \binom{n-1}{\lfloor n/2 \rfloor} \right) \Big/ (2^n - 2) .$$

Lemma 4.15. *If n is odd and $n \ge 3$, then $p \ge 1/3$.*

Proof. We show equivalently that $1 - 2p \le 1/3$, which is equivalent to showing $3\binom{n-1}{\lfloor n/2 \rfloor} \le 2^{n-1} + 2$. We show the latter inequality by induction on n.

If $n = 3$, it is easy to check that the inequality holds. Assume inductively that the inequality holds for an odd $n \ge 3$; then for $n + 2$ we have:

$$3\binom{n+1}{\lfloor (n+2)/2 \rfloor} = 3 \cdot \frac{4n}{n+1} \binom{n-1}{\lfloor n/2 \rfloor}$$

$$\le \frac{4n}{n+1} (2^{n-1} + 2) \le 2^{n+1} + 2 .$$

The last inequality boils down to $3n \le 2^n + 1$, which holds for every odd n. □

Corollary 4.16. *Let n be odd and $n \ge 3$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \ge 3(\log n - \log \epsilon)$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an odd $n \ge 3$ there exists a minority isolating family of size $\lceil 3 \log n \rceil$.*

Proof. From Lemma 4.15, the probability p that a random 2-partition isolates a fixed element in the minority block satisfies $p \ge 1/3$. Moreover, in this case coverage goals are simply elements of U, so $m = n$. The result then follows from Theorem 3.1. □

In the case of minority isolation with an odd n, balanced 2-partitions not only give us nicer random sampling, but improve the asymptotic bound on the size of minority isolating families.

Corollary 4.17. *Let n be odd and $n \ge 3$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of balanced 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \ge 2(1 + 1/(n-1))(\log n - \log \epsilon)$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an odd $n \ge 3$ there exists a minority isolating family of size $\lceil 2(1 + 1/(n-1)) \log n \rceil$.*

Proof. The probability that a random balanced 2-partition isolates a fixed element in the minority block is

$$p_b = \binom{n-1}{\lceil n/2 \rceil} \Big/ \binom{n}{\lceil n/2 \rceil} = \frac{\lfloor n/2 \rfloor}{n} = \frac{n-1}{2n} .$$

Again, in this case coverage goals are elements of U, so $m = n$ and the result follows from Theorem 3.1. □

For completeness, let us now do the analysis with an even n. In this case, for the probability p that a random 2-partition isolates a fixed element in the smaller (minority) block we have:

$$p = \sum_{0 \le j < n/2-1} \binom{n-1}{j} \Big/ \left\{ {n \atop 2} \right\}$$

$$1-p = \sum_{n/2-1 \le j < n-1} \binom{n-1}{j} \Big/ \left\{ {n \atop 2} \right\} = \sum_{0 < j \le n/2} \binom{n-1}{j} \Big/ \left\{ {n \atop 2} \right\}$$

Subtracting the first equality from the second one gives us:

$$1 - 2p = \left(\binom{n-1}{n/2-1} + \binom{n-1}{n/2} - 1 \right) \Big/ \left\{ {n \atop 2} \right\} \ .$$

Using $\left\{ {n \atop 2} \right\} = 2^{n-1} - 1$ and noting that the two binomial coefficients in the expression are equal, the equality simplifies to:

$$p = \left(2^{n-2} - \binom{n-1}{n/2} \right) \Big/ (2^{n-1} - 1) \ .$$

Lemma 4.18. *If n is even and $n \ge 4$, then $p \ge 1/7$.*

Proof. We show equivalently that $1 - 2p \le 5/7$, which is equivalent to $7\binom{n-1}{n/2} \le 5 \cdot 2^{n-2} + 1$. We show the latter inequality by induction on n.

If $n = 4$, it is easy to check that the inequality holds. Assume inductively that the inequality holds for an even $n \ge 4$; then for $n+2$ we have:

$$\begin{aligned} 7\binom{n+1}{(n+2)/2} &= 7 \cdot \frac{4(n+1)}{n+2} \binom{n-1}{n/2} \\ &\le \frac{4(n+1)}{n+2} (5 \cdot 2^{n-2} + 1) \le 5 \cdot 2^n + 1 \ . \end{aligned}$$

The last inequality boils down to $3n + 2 \le 5 \cdot 2^n$, which holds for all n. □

Corollary 4.19. *Let n be even and $n \ge 4$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \ge 7(\log n - \log \epsilon)$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an even $n \ge 4$, there exists a minority isolating family of size $\lceil 7 \log n \rceil$.*

Proof. From Lemma 4.18, the probability p that a random 2-partition isolates a fixed element in the minority block satisfies $p \ge 1/7$. Moreover, in this case coverage goals are elements of U, so $m = n$. The result then follows from Theorem 3.1. □

When n is even, balanced 2-partitions have two blocks of equal size, so we cannot use them for minority isolation. Instead, we use 2-partitions with the smaller block of size $n/2 - 1$ and the larger block of size $n/2 + 1$—we call these *semibalanced* partitions.

Corollary 4.20. *Let n be even and $n \ge 4$, let $\epsilon > 0$ and let $\mathcal{F}$ be a family of semibalanced 2-partitions chosen independently and uniformly at random such that $|\mathcal{F}| \ge 2(1+2/(n-2))(\log n - \log \epsilon)$. Then $\mathcal{F}$ is a minority isolating family with probability at least $1 - \epsilon$. Moreover, for an even $n \ge 4$, there exists a minority isolating family of size $\lceil 2(1 + 2/(n-2)) \log n \rceil$.*

Proof. The probability that a random semibalanced 2-partition isolates a fixed element in the minority block is

$$p_b = \binom{n-1}{n/2+1} \Big/ \binom{n}{n/2+1} = \frac{n/2-1}{n} = \frac{n-2}{2n} \ .$$

Again, in this case coverage goals are elements of U, so $m = n$ and the result follows from Theorem 3.1. □

Example 4.21. Minority isolation was motivated by the etcd example from Section 2. The number of nodes in the Jepsen test for etcd was 5, hence according to Corollary 4.17, with 9 randomly generated balanced partitions we will have isolated the leader in the minority block with probability at least 86%.

As in the Kafka example, the isolation here does not expose the inconsistent behavior on its own. We need to also consider read and write operations. As in Example 3.3, assume we only have three operations—read, write 0, and write 1—and each operation can be directed to any of the five nodes. Thus, in total we have 15 operations, each occurring with equal probability.

To expose the inconsistency in Fig. 1 (right), the partition first needs to isolate the leader in the minority block, which happens with probability 2/5. As in Example 3.3, this needs to be followed up with a write in the larger block, and read in the smaller block, which happens with probability $(2/15) \cdot (3/15) = 2/75$. Thus the overall probability of covering a fixed goal is 4/375. Since there are five goals (any node could be the leader), full coverage is possible with approximately 150 tests, and approximately 302 tests will ensure full coverage with probability at least 80%. □

5 OTHER APPLICATIONS

We now discuss some related work in rigorous guarantees on random testing and show how they are instances of our theorems. One might optimistically assume that "small" covering families always exist, and it is a matter of time before the appropriate probabilistic argument is found justifying a random testing procedure. The examples below show that this is not the case and that a rigorous treatment of testing can be quite subtle.

5.1 Hitting Families and PCT

In the context of testing asynchronous programs, Chistikov et al. [2016] introduce *hitting families*, a set of schedules that ensures every sequence of d events is covered by some schedule. Hitting families are another instance of our general notion of covering families: tests in this context are schedules (i.e., linearizations) of partially-ordered events, coverage goals are d-tuples of events, and a schedule covers ("hits") a d-tuple of events if it orders the events as they appear in the d-tuple, not necessarily contiguously. Chistikov et al. show a probabilistic construction of a hitting family of size $d!d \log n$ for the class of programs with n events and no dependency between them. This result is an instance of our Theorem 3.1 with $p = 1/d!$ and $m = n^d$.

Chistikov et al. go further and show *deterministic* constructions of hitting families for events organized as a tree. These constructions can give better bounds on the size of hitting families than ones we would obtain from Theorem 3.1. As an example, consider a program with events $0, 1, \ldots, n$, where events $1, \ldots, n$ are linearly ordered as $1 < 2 < \ldots < n$, and 0 is independent of all other events. There are $n + 1$ possible schedules of this program, and only one of them schedules 0 before 1, giving us a lower bound of $1/(n + 1)$ for the probability of covering a fixed pair of events. The total number of coverage goals for the program is the total number of admissible pairs of events, which is $n(n + 1)$. Therefore, Theorem 3.1 gives us existence of a 2-hitting family of size $(n + 1) \log (n(n + 1)) \approx 2n \log n$. And yet, all pairs of events can be covered with precisely two schedules: $0 < 1 < 2 < \ldots < n$ and $1 < 2 < \ldots < n < 0$. This is an example where a deterministic analysis can be asymptotically better than a probabilistic argument.

In the context of testing multi-threaded programs, Burckhardt et al. [2010] introduce a notion of *bug depth* and prove a lower bound on the probability that a test sampled randomly from a carefully constructed probability space hits a specific bug of fixed depth. They implement this technique,

called *PCT*, and use it to find several bugs in multi-threaded programs. They characterize a bug of depth d using an auxiliary notion of a *directive*, which is a set of additional dependencies in the program that ensures exposure of the bug. A directive of size d essentially corresponds to a d-tuple of events from Chistikov et al. For a fixed d, Burckhardt et al. show that a random test follows a specific directive of size d—thus exposing a corresponding bug of depth d—with probability $\frac{1}{nk^{d-1}}$, where n is the number of threads, and k is the total number of operations executed by the program.

Using the lower bound of Burckhardt et al. and directives as coverage goals, we can again apply Theorem 3.1. In a program with k operations, there are k^d directives, so there exists a covering family of size $ndk^{d-1} \log k$. Thus, as in the tree example, one cannot guarantee a "small" test suite. However, in contrast to the tree example, here it is unlikely that significantly smaller families exist, since multi-threaded programs can encode arbitrary partial orders. In fact, the same is true for hitting families: even for $d = 2$, hitting families for an arbitrary partial order can grow linearly with the number of elements [Dushnik and Miller 1941]. Thus, perhaps unsurprisingly, a "small" covering family need not exist. This example shows that one cannot in general assume that there is a covering family exponentially better than naive enumeration—some testing problems are inherently hard.

The approach of Burckhardt et al. also demonstrates that obtaining a non-trivial lower bound on the probability can require a carefully constructed probability space. A naive randomization may only provide a trivial lower bound, and thus a trivial upper bound on the number of tests. For example, a simple-minded approach which picks a thread at random at each step gives only an exponential lower bound for hitting a bug of depth d. The approach of Burckhardt et al. shows that a more sophisticated random generation strategy can be exponentially better.

5.2 Combinatorial Testing

Our results can broadly be seen as an instance of *combinatorial testing* [Colbourn 2004; Kuhn et al. 2010], the sub-field of testing that designs near-optimal test suites to cover all k-wise interactions among a large number of features. Like in our case, the key insight in combinatorial testing is that many bugs depend only on the interaction of a small number of features: in many cases, combinations of up to 6 features suffice for detecting most bugs in industrial applications [Kuhn et al. 2010].

The main objects studied in combinatorial testing are *covering arrays*. An $N \times n$ array over values from the set $\{0, \ldots, v-1\}$ is said to be *k-covering* if every $N \times k$ sub-array contains all v^k possible rows (with multiple occurences allowed). For given parameters n, k, and v, the goal is to find the smallest number of rows $N(n,k,v)$ such that there exists a k-covering array of size $N(n,k,v) \times n$. Many practical approaches for constructing covering arrays of small size have been studied, including greedy algorithms, hill-climbing algorithms, simulated annealing, and genetic algorithms [Colbourn 2004]. Non-constructive bounds on $N(n,k,v)$ are usually shown using the probabilistic method. For example, Godbole et al. [1996] show the following bound using the Lovász local lemma [Alon and Spencer 2004]:

$$N(n,k,v) \leq \frac{(k-1)\log n}{\log\left(v^k/(v^k-1)\right)}(1+o(1)) \quad .$$

For concrete values of k and v, more precise bounds are known. For example, $N(n,3,2) \leq 7.56444 \log n(1+o(1))$. This bound, attributed to Roux [Godbole et al. 1996], is proved by switching from arbitrary binary arrays to arrays with equal number of zeros and ones in each column—a technique similar to our switching from arbitrary partitions to balanced and semibalanced partitions in Sections 4.2 and 4.4.

Note that covering arrays are an instance of our covering families. A test is a row of an array—a vector of size n with components taking values in the set $\{0,\ldots,v-1\}$—and a coverage goal is a set of k positions $1 \le i_1 < i_2 < \ldots < i_k \le n$ and a vector $v \in \{0,\ldots,v-1\}^k$. A test $t \in \{0,\ldots,v-1\}^n$ covers this goal if the k components at positions $(i_1, i_2, \ldots, i_k)$ of t make up the vector v. A family $\mathcal{F}$ of vectors of size n is a covering family if it covers every coverage goal, thus it is nothing but a covering array. The probability that a random vector $t \in \{0,\ldots,v-1\}^n$ covers a coverage goal is v^{-k}. There are $\binom{n}{k}v^k$ coverage goals in all. Using Theorem 3.1, we obtain that there is a covering family of size $O(kv^k \log n)$, which is asymptotically the same as the bound by Godbole et al. This, of course, is not surprising, given that both bounds are obtained by the probabilistic method.

As a concrete example of combinatorial testing using covering arrays, consider the testing of combinational VLSI circuits [Seroussi and Bshouty 1988]. We are given a circuit with n inputs, which consists of a large number of smaller components, and each component depends on at most k of the n inputs. In VLSI testing applications, n may be much larger than k. A test consists of a Boolean vector of n bits. Informally, our coverage goal is to test every component with every possible input of k bits. Thus, our testing task precisely corresponds to finding small k-covering arrays for $v = 2$. From the discussion above, we obtain that there is a covering array of size $O(k2^k \log n)$. Moreover, by Theorem 3.1, every array of $2^k(k \log n - \log \epsilon)$ randomly generated rows is a covering array with probability at least $1 - \epsilon$. This was in fact the main result of Seroussi and Bshouty [1988]. Note that the naive bound is $O(n^k 2^k)$. In fact, Seroussi and Bshouty [1988] show a lower bound of $\Omega(2^k \log n)$, so the probabilistic method is almost optimal in this case.

6 OTHER RELATED WORK

6.1 Random Walks over Graphs

We focus on "static" notions of coverage, where showing lower bounds on probabilities are relatively easier. In random simulation of reactive systems, such as network protocols, the testing process defines a random walk over the state space of the system. If the random walk *rapidly mixes*, that is, converges to a stationary distribution in a small number of steps, random simulation fairly simulates the reachable state space in a stationary distribution. Rigorous analysis of random walks using Markov chain mixing techniques was pioneered by West [1989] for a simple class of decoupled network protocols (technically, the state space was a hypercube). Mihail and Papadimitriou [1994] used coupling techniques to prove rapid mixing for the class of symmetric dyadic flip-flops (SDFF): a concurrent system of automata, each with two states, communicating pairwise by rendezvous and where each action has a reverse action. Unfortunately, it is very hard to prove rapid mixing for most Markov chains, indeed, there are counterexamples to rapid mixing when the rather severe restrictions of SSDF are relaxed; for example, reversibility is a strong requirement [Levin et al. 2009].

It is tempting to provide a random testing version of systematic testing procedures such as context-bounded reachability. For example, could one show that a random walk on the state space defined by a multithreaded program quickly visits any k-context-bounded reachable state? Unfortunately, this involves bounding the *hitting time* for a random walk on a directed graph, i.e., the expected time for a random walk to visit a node, and we do not have a sufficiently strong lower bound on the hitting time that gives better than exponential bounds.

Thus, we believe extensions of our techniques to "dynamic" coverage may require sophisticated methods.

6.2 Deterministic Families of Tests

Probabilistic constructions demonstrate existence of covering families, and we can amplify the probability of finding a test suite. However, the soundness guarantee is "with high probability." Unfortunately, even when a small covering family can be shown to exist, an *explicit, deterministic* construction of a covering family can be a significantly harder problem. This is a recurring theme in combinatorics [Alon 2010]: for many problems, explicit deterministic constructions come much later than an existence argument using the probabilistic method. In fact, there are many combinatorial objects proved to exist using the probabilistic method for which we do not know optimal deterministic constructions [Alon and Spencer 2004]! For example, our notion of splitting families is related to perfect hash functions [Czech et al. 1997; Fredman et al. 1984; Yao 1981]. In this context, Yao [1981] gives a highly non-trivial deterministic construction of k-splitting families of size $4^{k^2}(\log_2 n)^{k-1}$, which is worse than the bound $k^{k+1}(k!)^{-1}\log n$ obtained through Corollary 4.4. Furthermore, decision problems related to minimal constructions or enumerations are usually computationally intractable (NP-hard [Seroussi and Bshouty 1988; Yannakakis 1982]); thus, it is unlikely that a simple deterministic approach can supplant the simplicity of random testing.

Even when deterministic construction of a covering family is known, it may be infeasible to execute all tests from the family. Consider Yao's construction of k-splitting families. Already for $n = 5$ and $k = 2$—the values used in Jepsen tests—the size of the family is approximately 595. Increase n to 10 and k to 3, and the size grows to 9,609,717. Compared to this, randomly constructed families have two additional benefits: they do not require a sophisticated test generation algorithm and we can always stop the construction and apply Theorem 3.1 or its instances in reverse to obtain the probability that the constructed family is covering. The probability can serve as a qualitative measure of coverage.

6.3 Systematic Approaches

One direction of research in assuring correct behavior of distributed systems is to build fully verified systems "from scratch." Despite heroic efforts in this direction [Hawblitzel et al. 2015; Lamport 1994; Wilcox et al. 2015], we are quite far from replacing existing infrastructure with fully verified deployments of comparable functionality and performance.

Systematic approaches like model checking [Baier and Katoen 2008; Fisman et al. 2008; Konnov et al. 2017; Leesatapornwongsa et al. 2014; Yang et al. 2009] and systematic fault-injection [Alvaro et al. 2015; Gunawi et al. 2011] design algorithms and heuristics that perform systematic search over behaviors which are sufficient to find all bugs. Theoretically, these approaches have an additional benefit of providing guarantees about the *absence* of bugs of a certain kind.

7 CONCLUSIONS

Random simulation is the primary mode of testing systems with large and complex state spaces across many different domains: from sequential circuits to network protocol implementations and to large-scale distributed systems. Practitioners tend to use random schedulers and random fault-injection [Apache Hadoop 2016; Claessen et al. 2009; Izrailevsky and Tseitlin 2011; Kingsbury 2017] to test their systems, sometimes even advocating doing this in production. The latter approach, dubbed Chaos Engineering and codified in the Principles of Chaos Engineering [2017], is the underlying philosophy of Netflix Simian Army [Izrailevsky and Tseitlin 2011], a collection of tools called monkeys that randomly tamper with the system in production, with engineers monitoring the effects and addressing problems as they arise.

Our results are a step towards a theoretical understanding of random testing: we show that the effectiveness of testing can be explained in certain scenarios by providing lower bounds on the

probability that a single random test covers a fixed coverage goal. For network partition tests, we introduce a set of coverage goals inspired by actual bugs in distributed systems and show lower bounds on the probability of a random test covering a goal.

ACKNOWLEDGMENTS

This research was sponsored in part by the European Research Council Grant Agreement No. 610150 (ERC Synergy Grant ImPACT (http://www.impact-erc.eu/)). We thank Madan Musuvathi and Damien Zufferey for discussions and comments.

REFERENCES

Noga Alon. 2010. *Algebraic and Probabilistic Methods in Discrete Mathematics.* Birkhäuser Basel, Basel, 455–470. https://doi.org/10.1007/978-3-0346-0425-3_1

Noga Alon and Joel H. Spencer. 2004. *The Probabilistic Method.* Wiley. https://books.google.de/books?id=6QIEjeMjBkkC

Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* ACM, 331–346. https://doi.org/10.1145/2723372.2723711

Apache Hadoop. 2016. Fault Injection Framework and Development Guide. Retrieved July 7, 2017 from http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking.* MIT Press. https://books.google.de/books?id=nDQiAQAAIAAJ

Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.* ACM, 7. https://doi.org/10.1145/343477.343502

Eric A. Brewer. 2012. CAP Twelve Years Later: How the "Rules" Have Changed. Retrieved July 7, 2017 from https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010.* ACM, 167–178. https://doi.org/10.1145/1736020.1736040

Chaos Engineering. 2017. Principles of Chaos Engineering. Retrieved July 7, 2017 from http://principlesofchaos.org/

Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. 2016. Hitting Families of Schedules for Asynchronous Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 9780. Springer, 157–176. https://doi.org/10.1007/978-3-319-41540-6_9

Koen Claessen, Michal H. Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf T. Wiger. 2009. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009.* ACM, 149–160.

Charles J. Colbourn. 2004. Combinatorial aspects of covering arrays. *Le Matematiche* 59, 1,2 (2004), 125–172. https://lematematiche.dmi.unict.it/index.php/lematematiche/article/view/166

Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. 1997. Perfect Hashing. *Theor. Comput. Sci.* 182, 1-2 (1997), 1–143. https://doi.org/10.1016/S0304-3975(96)00146-6

Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. 2016. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.* USENIX Association, 249–262. https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis

Ben Dushnik and E. W. Miller. 1941. Partially Ordered Sets. *American Journal of Mathematics* 63, 3 (1941), 600–610. http://www.jstor.org/stable/2371374

Gerald A. Edgar, Daniel H. Ullman, and Douglas B. West. 2017. Problems and Solutions. *The American Mathematical Monthly* 124, 2 (2017), 179–187. http://www.jstor.org/stable/10.4169/amer.math.monthly.124.2.179

Dana Fisman, Orna Kupferman, and Yoad Lustig. 2008. On Verifying Fault Tolerance of Distributed Protocols. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 315–331. https://doi.org/10.1007/978-3-540-78800-3_22

Michael L. Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *J. ACM* 31, 3 (1984), 538–544. https://doi.org/10.1145/828.1884

Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. https://doi.org/10.1145/564585.564601

Anant P. Godbole, Daphne E. Skipper, and Rachel A. Sunley. 1996. t-Covering arrays: Upper bounds and poisson approximations. *Combinatorics Probability and Computing* 5, 2 (12 1996), 105–117.

Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley. https://books.google.de/books?id=cjgPAQAAMAAJ

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. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011.* USENIX Association. https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* ACM, 1–17. https://doi.org/10.1145/2815400.2815428

Yury Izrailevsky and Ariel Tseitlin. 2011. The Netflix Simian Army. Retrieved July 7, 2017 from https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116

Kyle Kingsbury. 2013. Partitions for Everyone! Retrieved July 7, 2017 from https://www.infoq.com/presentations/partitioning-comparison

Kyle Kingsbury. 2013–2017. Jepsen. Retrieved July 7, 2017 from http://jepsen.io/

Igor Konnov, Helmut Veith, and Josef Widder. 2017. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Inf. Comput.* 252 (2017), 95–109. https://doi.org/10.1016/j.ic.2016.03.006

D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2010. Combinatorial Testing. In *Encyclopedia of Software Engineering*, Phillip A. Laplante (Ed.). CRC Press, 1–12.

Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. 2008. A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008.* IEEE, 1–9. https://doi.org/10.1109/FMCAD.2008.ECP.29

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923. https://doi.org/10.1145/177492.177726

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* USENIX Association, 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

David Asher Levin, Yuval Peres, and Elizabeth Lee Wilmer. 2009. *Markov Chains and Mixing Times.* American Mathematical Society. https://books.google.de/books?id=vgQnngEACAAJ

Cristina Videira Lopes. 2016. Distributed Systems Testing: The Lost World. Retrieved July 7, 2017 from http://tagide.com/blog/research/distributed-systems-testing-the-lost-world/

Caitie McCaffrey. 2015. The Verification of a Distributed System. *ACM Queue* 13, 9 (2015), 60. https://doi.org/10.1145/2857274.2889274

Milena Mihail and Christos H. Papadimitriou. 1994. On the Random Walk Method for Protocol Testing. In *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings (Lecture Notes in Computer Science)*, Vol. 818. Springer, 132–141. https://doi.org/10.1007/3-540-58179-0_49

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.* USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

Colin Scott. 2016. Technologies for Testing and Debugging Distributed Systems. Retrieved July 7, 2017 from http://colin-scott.github.io/blog/2016/03/04/technologies-for-testing-and-debugging-distributed-systems/

Gadiel Seroussi and Nader H. Bshouty. 1988. Vector sets for exhaustive testing of logic circuits. *IEEE Trans. Information Theory* 34, 3 (1988), 513–522. https://doi.org/10.1109/18.6031

Colin H. West. 1989. Protocol Validation in Complex Systems. In *SIGCOMM '89, Proceedings of the ACM Symposium on Communications Architectures & Protocols, Austin, TX, USA, September 19-22, 1989.* ACM, 303–312. https://doi.org/10.1145/75246.75276

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th*

USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA. USENIX Association, 213–228. http://www.usenix.org/events/nsdi09/tech/full_papers/yang/yang.pdf

Mihalis Yannakakis. 1982. The Complexity of the Partial Order Dimension Problem. *SIAM Journal on Algebraic Discrete Methods* 3, 3 (1982), 351–358. https://doi.org/10.1137/0603036 arXiv:https://doi.org/10.1137/0603036

Andrew Chi-Chih Yao. 1981. Should Tables Be Sorted? *J. ACM* 28, 3 (1981), 615–628. https://doi.org/10.1145/322261.322274