Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems

Haojun Ma Hammad Ahmad Aman Goel Eli Goldweber
Jean-Baptiste Jeannin Manos Kapritsos Baris Kasikci
University of Michigan
{mahaojun, hammada, amangoel, edgoldwe, jeannin, manosk, barisk}@umich.edu

Abstract

Distributed systems are hard to design and implement correctly. Recent work has tried to use formal verification techniques to provide rigorous correctness guarantees. These works present a hard choice, though. One must either opt for the power of refinement-based approaches like IronFleet and Verdi, at the cost of large amounts of manual effort; or choose the more automated approach of I4, IC3PO, SWISS and DistAI which give up the ability to prove refinement and the power and scalability that come with it.

We propose an alternative approach, Sift, that combines the power of refinement with the ability to automate proofs. Sift is a two-tier methodology that uses a new technique, *refinement-guided automation*, to leverage automation in a refinement proof and a divide-and-conquer technique to split a system into more refinement layers when necessary. This combination advances the frontier of what systems can be proven correct using a high degree of automation. Contrary to what was possible before, our evaluation shows that our novel approach allows us to prove the correctness of a number of systems with little manual effort, and to extend our proofs to include not just the protocols, but also an executable distributed implementation of these systems.

1 Introduction

Recently, formal verification has emerged as a potential alternative to the traditional approach of testing. The promise of formal verification—to eliminate all bugs by construction—is particularly attractive for distributed systems, which are notoriously hard to design and implement correctly.

Despite recent efforts, however, formal verification of distributed systems is still not ready for real-world applications. The most powerful techniques, such as IronFleet [34] and Verdi [63], rely on *refinement* proofs [1, 25, 42] to reason about complex systems and verify real implementations. Alas, the power of those techniques comes at a high cost: performing these refinement proofs manually requires large amounts of manual effort.

In an attempt to reduce the manual verification effort, the Ivy tool [56] proposes to express distributed protocols using decidable—and thus simpler to verify—reasoning [57]. The Ivy tool achieves remarkable automation, but still requires significant human effort to complete the proof. More recent approaches, like I4 [50, 51], IC3PO [27], SWISS [33] and DistAI [66], leverage model checking and SMT solvers to automate the most challenging part of proving the correctness of distributed protocols: finding an *inductive invariant*. Alas, this automation comes at the expense of expressiveness and applicability, because tools like I4 and DistAI were designed to prove properties of *monolithic* protocols which consist of a single layer. As such, they cannot prove refinement.

Refinement [1, 25, 42], however, is an essential concept in proving the correctness of real, complex systems. It allows us to prove the correctness of a system by showing that it is equivalent to a simpler, more abstract version of that system. The power of refinement comes in many forms:

Concise specification As Lamport has argued [45] and as IronFleet demonstrated, specifications should be written as simple, abstract state machines. Consider the specification of a Paxos-based State Machine Replication in IronFleet, where the goal is to prove that the entire service is linearizable. Expressing linearizability as a set of properties on the requests and responses is daunting and will likely yield a complex specification. Using refinement, the task is simple: just show that the entire service is equivalent to a single machine executing requests one at a time. Similarly, the sharded key-value store in IronFleet was simply proven equivalent to an abstract, logically centralized key-value store; i.e., a map.

Scaling to complex systems As IronFleet and Verdi demonstrated, the key to dealing with the complexity of a real system is to take a modular approach: split the proof into multiple layers and show that each layer refines the one above it. This is especially true when verifying actual implementations, as these tend to be much more complex than abstract protocols. In the absence of refinement, we are left with the task of reasoning about a single, monolithic system, whose complexity now becomes a limiting factor for both

manual and automated approaches.

Dealing with undecidability Even when one only cares about proving the correctness of the protocol, and not of the implementation, being unable to split a monolithic system into multiple layers can be a showstopper for automation. As Padon et al. demonstrated [55], some protocols may be undecidable by construction and thus not amenable to the automation of I4 and IC3PO. In these cases, one can use refinement to split the protocol into two layers, each of which is separately decidable [62].

We aim to get the best of what are currently two distinct worlds: the *power of refinement* (i.e. IronFleet-style proofs) but with only *a fraction of the manual effort* (i.e. using the automation of monolithic provers like I4, IC3PO, SWISS and DistAI). This combination allows us to not only achieve simple, concise specifications, but also to scale our proofs to more complicated distributed protocols, and even to distributed implementations.

To achieve this goal, we introduce Sift, a two-tier methodology that combines automated verification with a small amount of manual effort to push the boundary on the kinds of systems that can benefit from proof automation. Just like IronFleet before it, Sift is a *methodology*, not a tool. Its contribution is a way of structuring refinement proofs in order to leverage the automation of existing tools. Similar to how IronFleet guided developers to manually construct proofs based on the existing tools (TLA+ and Dafny), so does Sift show developers how to construct proofs that leverage the automation of more recent tools, like IC3PO and Ivy.

The first tier of Sift introduces a new technique, called *refinement-guided automation*, which leverages the automation of monolithic provers in the context of a refinement proof. At the high level, this technique enables the automation of refinement proofs between two layers by *encapsulating* the state of the upper, more abstract, layer into the state of the lower, more concrete layer. This encapsulation allows us to transform a two-layer refinement proof into a single-layer, monolithic proof that provers like I4, IC3PO, SWISS and DistAI can perform.

Leveraging automation to prove refinement is not always enough, though. Monolithic provers have their limits and thus some refinement proofs are just too complex to prove automatically. When that happens, we provide developers with an escape hatch. The second tier of the Sift methodology describes a divide-and-conquer technique for introducing intermediate layers, thus splitting a complex proof into chunks that are small enough for the prover to handle.

The Sift methodology applies refinement-guided automation within each refinement step and uses our divide-and-conquer technique to split a refinement step into smaller, more manageable steps. As a result, Sift allows us to apply, for the first time, automation to refinement-based proofs and scale to much harder problems than was previously possible. We use Sift to automate the verification of four distributed implementations, whose proof required minimal manual effort (less than five minutes, in most cases).

We further use our divide-and-conquer technique to prove the correctness of an implementation of Raft [54] and an implementation of MultiPaxos [43, 44] — a feat that was only possible before by providing a fully manual proof of correctness. Using Sift, we were able to automate most of the proof for both Raft and MultiPaxos. The manual effort required to complete the proof with Sift is not only significantly less than that of previous approaches, it is also much less reliant on having expertise in formal verification.

Overall, this paper makes the following contributions:

- We introduce *refinement-guided automation*, a technique that leverages the automation of monolithic-oriented tools to perform more complex, refinement-based proofs.
- We present a divide-and-conquer technique for splitting a complex refinement proof into smaller pieces, such that each piece is amenable to automated verification.
- We introduce Sift, a methodology that incorporates refinement-guided automation and our divide-and-conquer technique. We evaluate Sift on six distributed implementations and find that it allows us to prove their correctness in a mostly automated manner which drastically reduces the manual effort required compared to previous refinement-based approaches.

The rest of the paper is structured as follows. Section 2 discusses the tradeoff between automation and refinement. Section 3 recaps some background material, while Section 4 gives an overview of Sift. Section 5 introduces refinement-guided automation and Section 6 shows how to introduce intermediate refinement layers when needed. Section 7 evaluates the effectiveness of using Sift to automate the verification of a number of distributed implementations. Section 8 presents the limitations of Sift and discusses future work. Section 9 discusses related work and Section 10 concludes.

2 The Price of Automation

As discussed earlier, there are currently two approaches for verifying the correctness of distributed systems. The first is the powerful but manual approach of IronFleet and Verdi [34, 35, 63], where the developer uses refinement to show that a complex implementation is equivalent—through a series of layers or transformations—to an abstract specification.

The second approach is that of I4 [50], IC3PO [27], SWISS [33] and DistAI [66] which leverage the power of model-checking and SMT solving [5] to automatically prove the correctness of abstract system descriptions at the protocol level. These approaches aim to prove that a given safety property holds for the protocol at hand, by automatically identifying an *inductive invariant* that implies this safety property.

While such automation is undoubtedly a desirable property, it comes at a heavy price. In particular, I4, IC3PO, SWISS and DistAI can only perform *monolithic* proofs: they can prove

that a protocol—defined as a single layer—satisfies a given safety property. As we described in Section 1, this not only limits the type of specifications we can use, but also severely limits the scalability of the approach.

Most importantly, the scalability limitation is not an artifact of the implementation of *monolithic provers*—like I4, IC3PO, SWISS and DistAI—but rather inherent in their design. By asking the underlying solver to find an inductive invariant that supports the desired safety property, they essentially adopt an *all-or-nothing* approach: either the solver is powerful enough to find an inductive invariant or it is not. If we consider more and more complex systems, we soon reach a point where the solver is simply not powerful enough to find an inductive invariant.

In fact, a similar dichotomy presents itself when the protocol description has elements outside the *decidable* fragment of logic [47, 55]. In several of these cases, the solver struggles considerably, even when it is trivial for a human to split the problem into decidable sub-problems. Without the ability to split this monolithic proof into multiple pieces, there is no middle ground. For example, I4 simply fails when the problem lies outside the decidable fragment, even though it is still possible to use refinement to split the protocol into two layers, each of which is separately decidable [62].

In this paper, we show that there exists a middle ground between the fully manual approaches that support refinement, like IronFleet and Verdi; and the automated-but-monolithic approaches, like I4, IC3PO, SWISS and DistAI. This middle ground, enabled by our novel Sift methodology, allows for refinement-based reasoning—and thus allows us to prove the correctness of complex distributed implementations—while making heavy use of automation to drastically reduce the amount of manual effort required compared to IronFleet and Verdi.

3 Background

3.1 Multi-Layer Refinement

Sift is heavily based on the notion of refinement. We will therefore first recap the notion of refinement and how it can be used to prove the correctness of complex systems.

A system P *refines* another system Q if the observable outputs produced by any execution of Q can also be produced by some execution of P. In the case of distributed systems, the only outputs that are visible to external observers are the messages produced by these systems.

In the simplest application of refinement, the developer writes two layers: a specification and an implementation. The specification is written as a simple, logically centralized state machine. In the case of a sharded key-value store, for example, the specification is a simple map, where the only possible actions are to `put` something to the map, or to `get` something from the map [34, 35]. The developer then shows that the implementation refines the specification, thus proving the correctness of the implementation.

In more complex systems, directly proving refinement from the implementation to the specification can be difficult [34, 35, 63]. In that case, the developer must insert one or more increasingly complex layers between the implementation and specification, thus creating a multi-layer structure, where each layer must be proven to refine the one above it. We explain how to design and insert intermediate layers in Section 6.

3.2 Automated Reasoning and Monolithic Provers

Traditional verification languages [4, 46] rely on the developer to write a full proof, including a large number of manual annotations. As a result, approaches like IronFleet [34, 35] and Verdi [63] incur a high proof-to-code ratio. To reduce this manual effort, Ivy [56] uses decidable logic to guarantee completeness. With Ivy, the developer only needs to find an *inductive invariant*—an invariant which is closed (inductive) under the system transitions—and the prover can automatically identify if this inductive invariant is correct. Ivy significantly simplifies the effort of proving the correctness of distributed systems, but finding such inductive invariants is still a non-trivial task that relies on human intuition and an intimate understanding of the system at hand.

To push the automation a step further, I4 [50] leverages the regularity of distributed protocols, so that the inductive invariant can be automatically inferred from a small, finite instance. Unfortunately, such a strategy only applies to monolithic protocols, not refinement proofs. Thus, I4 doesn't scale well when the system has a large state space and complex transitions. More recent tools [27, 33, 66] have followed the direction of using finite instances to guide the verification of distributed protocols. All these tools, however, apply only to monolithic proofs and cannot support refinement. We call such tools *monolithic provers*.

3.3 IC3PO: Our Monolithic Prover of Choice

The design of Sift does not rely on the internals of the monolithic prover that it uses. The refinement-guided automation technique of Sift can leverage any tool designed for automating monolithic, single-layer proofs. In fact, we previously tried I4 as the monolithic prover in Sift, but later found that IC3PO performs better. Our experience so far shows that IC3PO also outperforms SWISS and DistAI. As new and more powerful monolithic provers become available, Sift can adopt them to perform even larger refinement steps to further reduce manual effort. The next paragraph gives a short overview of IC3PO.

IC3PO [27, 28] is a recently-developed prover that uses the synergistic relationship between symmetry and quantification to prove the safety of distributed protocols fully automatically,

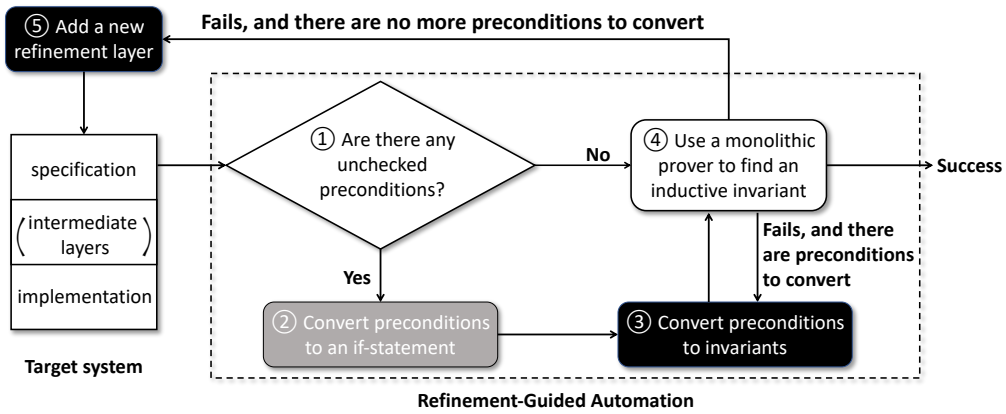

Figure 1: Summary of the Sift methodology. White boxes are fully automated, gray boxes indicate a trivial syntax change, and black boxes denote manual effort.

by inferring compact inductive invariants with both *universal* and *existential* quantifiers. At its core, IC3PO exploits the inherent regularity present in distributed protocols to significantly scale up IC3/PDR-style verification [10, 23] over finite instances of the protocol. Starting with an initial instance size, IC3PO systematically computes quantified inductive invariants over protocol instances of increasing sizes, until protocol behaviors *saturate*, concluding with an inductive proof that works for all instances of the protocol.

4 Overview of Sift

This paper introduces Sift, a methodology that allows reasoning about complex systems while still using a large degree of automation in proofs. Sift accomplishes this by employing a small amount of manual effort, when needed, to split the system into a number of layers, where each layer can be shown to refine the layer above it.

Figure 1 shows an overview of the Sift methodology. Initially, the developer starts with an implementation of the system, along with a specification, both written in the Ivy language [56]. If one were to use the Ivy prover, they would have to provide a manual proof of refinement between the specification and implementation. Sift, instead, introduces our *encapsulation* technique to merge the two layers into a single proof that our monolithic prover can attempt to solve.

Indeed, the first step of the Sift methodology is to attempt to prove refinement directly between the implementation and specification layers. If this proof is too much for the prover to handle, the developer adds an additional layer of refinement and tries again. Each additional layer of refinement splits the proof into smaller pieces that are more amenable to automation; but, of course, this comes at the cost of some manual effort, as the developer must manually introduce the new layer.

In the next two sections, we describe the Sift methodology in more detail. Section 5 describes how we can use the automation of a monolithic prover to perform a refinement proof between two layers (steps ①-④ in Figure 1). Section 6 presents the methodology for adding intermediate layers to the refinement structure (step ⑤).

Algorithm 1 Specification of the Sharded Hash Table (SHT)

```
1  function requests(R : request) : bool
2  function replies(R : reply) : bool
3  function map(K : key) : value
4  initialization {
5    ∀R. requests(R) ← false
6    ∀R. replies(R) ← false
7    ∀K. map(K) ← 0
8  }
9  action commit(req : request, rep : reply) = {
10   require rep.type = req.type
11   require rep.src = req.src
12   require rep.key = req.key
13   require req.type = read ⇒ rep.data = map(req.key)
14   if ¬requests(req) {                ▷ require ¬requests(req)
15     if req.type = write {
16       map(req.key) ← req.data
17     };
18     requests(req) ← true;
19     replies(rep) ← true;
20   }
21 }
```

Case Study: Sharded Hash Table Throughout this paper, we use the example of a Sharded Hash Table application (SHT) [34] to illustrate the Sift methodology. SHT implements a distributed key-value store, and consists of two layers, a specification layer and an implementation layer. As shown in Algorithm 1, the specification layer describes a key-value store as a simple *map* from keys to values. It maintains two local sets (modeled as boolean-valued functions, lines 1 and 2) to keep track of which messages (requests and replies) have

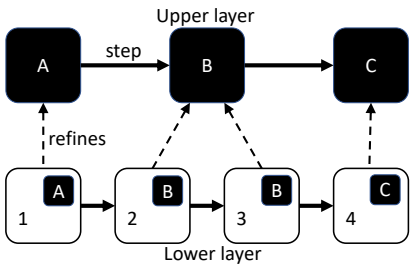

Figure 2: Encapsulation: to enable automatic refinement proofs, the state of the upper layer (A, B, C) is encapsulated inside the state of the lower layer (1, 2, 3, 4) refining it.

been sent. Initially, all keys are mapped to 0, and no messages have been sent. Requests can either be read requests or write requests. The only transition allowed by this specification is to *commit* a request *req* and its reply *rep*: i.e., perform the update (if this is a write request) and mark the corresponding messages as sent by setting $requests(req)$ and $replies(rep)$ to *true*. The specification layer consists of 32 lines of Ivy code.

In the implementation layer, every node contains a local hash table containing some subset of the total keys in the system and a delegation map. The node uses the delegation map to maintain its knowledge of where keys are stored on remote nodes. Each node can service a request using `get` and `set` actions for the keys that are locally stored, or use the delegation map to look up and forward requests to the appropriate node in the system if the requested key is not local. Nodes at the implementation layer can dynamically exchange sets of keys they are responsible for, by exchanging `delegate` messages (each carrying a key-value pair) among themselves. The implementation layer consists of 127 lines of Ivy code.

We aim to show that the implementation layer *refines* the specification, i.e., that any observable output produced by any execution of the implementation layer can also be produced by some execution of the specification. A key property is that for every key owned by a node at the implementation layer, the data matches the value stored in the specification. Additionally, every key is either owned by exactly one node in the system, or part of an in-flight `delegate` message.

5 Refinement-Guided Automation

We first explain the key high-level idea behind automating refinement proofs (step ④ in Fig. 1, Section 5.1). We then present what modifications Sift makes to the layers of a target system description to ensure that the correspondences between the layers are correctly represented before a proof is attempted (steps ①-③ in Fig. 1, Section 5.2).

Algorithm 2 Example of encapsulation in SHT

```
1   action set(req : request) = {
2     require req.type = write
3     owner ← delegation.get_owner(req.key)
4     if owner = me {
5       hash(req.k) ← req.v
6       rep ← create_reply(req)
7       call spec.commit(req, rep)
8       call network.send_reply(rep)
9     } else {
10      call network.forward_request(req, owner)
11    }
12  }
```

5.1 From Monolithic Proofs to Refinement

A key feature of Sift is that it uses the automation of monolithic provers to perform more complex, refinement-based proofs. As we explained in Section 2, monolithic (i.e. single-layer) proofs do not scale to complex systems, either due to complexity or undecidability. Yet, monolithic proofs are the only type of proof supported by these provers. The first innovation of Sift is that it converts a refinement proof between two layers into a monolithic proof which can be given as input to any monolithic prover.

We perform this transformation using our *encapsulation* technique, depicted in Figure 2. The idea of encapsulation is simple: if we want to show that a lower layer L refines an upper layer U, then we augment the state of L with the state of U. Additionally, whenever the state machine L makes a transition, the *encapsulated* U state also makes an upper-layer transition. In practice, this is expressed as a function call in Ivy, where the lower layer invokes a transition on its encapsulated state. For example, in the SHT application, the lower layer includes an encapsulated *spec* object (see Algorithm 1) and a lower-layer transition calls $spec.commit()$ if it refines the *commit* transition of the upper layer.

Encapsulating the upper-layer state into the lower layer effectively creates a single, *augmented lower layer* that can be used to reason about the relation between the upper and lower layer. Most importantly, we can now leverage traditional single-layer provers to show whether a certain property—the refinement property—holds for this augmented lower layer.

Case study: refinement proof for the SHT Algorithm 2 shows a simple example of encapsulation at the implementation layer of SHT. To perform this encapsulation, the implementation layer imports (in Ivy) the specification layer. In this example of handling a `set` request, the program checks if this node (*me*) is the owner of the key in the request (line 4). If it is the owner, the implementation layer internally makes a call (line 7) to *spec.commit* (shown in Algorithm 1). This transition corresponds to the transition from state 1 to state 2 in Figure 2: the implementation layer transitions from state 1

to state 2, while each of these states encapsulates the corresponding upper layer state, indicating a transition from state A to state B at the upper layer.

If this node is not the owner, it simply redirects the request to the owner. Such an implementation layer transition does not entail a specification layer transition and so the code does not call *spec.commit* or any specification-level function. This is usually called a "stutterring" step of the specification layer—essentially a *no-op*—and corresponds to the transition from state 2 to state 3 in Figure 2.

To prove that the implementation refines the specification, we ask our monolithic prover to prove a simple property:

$$\forall R : reply, N : node.\ \ net.replied(R,N) \implies spec.replies(R)$$

This property says that any reply R sent to any node N at the network (implementation level) can only be present if the same reply R is present at the specification level. Since replies are the only observable outputs of the system, it ensures that every output of the implementation is also an output of the specification, thus ensuring that the implementation is indeed a refinement of the specification. Note that the reply message at the implementation layer is part of the network and thus modeled as $net.replied(M,N)$.

5.2 Enforcing pre- and postconditions across layers

When calling functions from a lower layer to an upper layer, an upper-layer transition's precondition must be met. The preconditions of the callee (in the upper layer) become postconditions (assertions) for the caller (in the lower layer) to check. For example, on line 13 of SHT's specification (Algorithm 1), before committing a request, the precondition concerning the request, *req*, and the corresponding reply, *rep*, must be met:

$$req.type = read \implies rep.data = map(req.key)$$

This precondition ensures that every time a read request is committed, the data contained in the response must correspond to the data in the abstract map. Since it is the caller's responsibility to guarantee that this precondition is met before committing the request, this precondition is effectively an assertion that needs to be checked by the monolithic prover. Unfortunately, the current state-of-the-art monolithic provers do not support checking these kinds of assertions, and can only find an inductive invariant for a safety property.

If we attempt to ignore this assertion check and let the monolithic prover prove the refinement property as is, the result could be unsound—i.e., the proof may go through even if the implementation is buggy. For example, let us consider again the refinement property for SHT:

$$\forall R : reply, D : node.\ \ net.replied(R,D) \implies spec.replies(R)$$

Without precondition checks, a buggy implementation can send a bogus reply message and call *commit* at the encapsulated specification layer. This would make the refinement property trivially inductive—since the *commit* call adds the message to the *replies*—without guaranteeing that contents of that message are correct.

To avoid this problem, Sift needs to consider the assertions in function calls to maintain soundness in automated refinement proofs. In the rest of this section, we explain how we transform the assertions to either conditionals (if/else) or invariants that the monolithic prover can reason about.

5.2.1 Converting Assertions to Conditionals

A straightforward approach to model assertions in function calls is to convert the callee to an *always-enabled action* using a conditional if/else block [35]. The developer can manually rewrite an assertion P as follows: if P holds, take the transition; otherwise, do nothing. In this context, the entire if/else block is always-enabled, in that it has no preconditions and can always be taken.

For example, the original SHT specification had a precondition $\neg requests(req)$ in the specification of the *commit* action, which we convert to an if-statement (Algorithm 1, line 14). This precondition ensures that the specification can never execute the same request twice.

The benefit of this approach is that it does not rely on any understanding of the system, which makes it very easy to implement. It has, however, two downsides. First, adding an if/else block in place of a precondition makes the proof a little harder for monolithic provers, since it is harder to find an inductive invariant for a weaker problem. Second, if the if-statement refers to ghost state—i.e., proof-related state that is not compiled to an executable—such as the sets corresponding to network messages, these if-statements are not compiled directly to executable code.Therefore, if there are any assertions that refer to ghost states at the implementation layer, we cannot rely on the approach of converting assertions to conditionals. In these cases, we need to convert them to invariants, as we describe below.

5.2.2 Converting Assertions to Invariants

A second, more involved approach to the problem is to convert these assertions into invariants. Doing so requires human intuition but reduces the difficulty for the monolithic prover. For every assertion that needs to be checked, there must be an invariant to support its proof. The key idea is simple: a programmer can trace backward through a function call from the upper layer (the callee) to the lower layer (the caller) to find the enabling precondition. For the SHT precondition example above, we observe that only the node who owns the key can commit the reply. Leveraging this observation, we can construct an invariant that if node N thinks it is the owner

of key K, the local value for key K at node N, which forms the reply, must match the value in the spec:

$$\forall N : node, K : key.\ server(N).delmap(K,N) \implies server(N).hash(K) = spec.map(K)$$

where $server(N).delmap(K,N)$ indicates that from the perspective of server N, the owner of key K is N ($delmap$ stands for the delegation map). By maintaining this invariant, Sift can ensure the associated assertion will never be violated during the execution of the system. Note that the invariant is not necessarily inductive, but Sift leverages the automation of the monolithic prover to complete the proof.

Case study: converting assertions for the SHT The manual effort involved in the SHT proof requires converting seven assertions into two if-statements and five invariants. The assertion $\neg requests(r1)$ is converted from an assertion to an if-statement, as described in Section 5.2.1. On the other hand, the first three assertions (lines 10 to 12 in Algorithm 1) are already enforced by the implementation layer and do not need to be converted. We could further use the methodology described above in Section 5.2.2 to convert the fourth assertion (line 13) to an invariant, but it turns out that monolithic provers are powerful enough to complete the proof even if we simply convert it to a if-statement.

6 Introducing Intermediate Layers

We have discussed how to use automation to prove refinement between two layers. However, sometimes, the automation provided by the monolithic prover is not powerful enough to prove the desired refinement. This can happen either due to the complexity of the proof, or the presence of undecidable reasoning. When faced with such complex proofs, monolithic provers will either time out or run out of memory.

To perform such complex proofs, the solution is to introduce an intermediate layer (step ⑤ in Figure 1), thereby splitting the proof into two simpler refinement proofs: one refinement proof from the original lower layer to the intermediate layer, and another refinement proof from the intermediate layer to the original upper layer. By repeatedly using this proof-splitting technique until every refinement proof is automated, we effectively execute a *divide-and-conquer strategy* that allows us to tackle complicated refinements.

This idea is similar to IronFleet's methodology of introducing an intermediate protocol layer to simplify the proof. In IronFleet, however, the developer needed to both *write* an intermediate layer and *manually prove* it correct. By contrast, Sift uses the automation of monolithic provers to dispense with most of the latter manual effort of writing the proof, and only requires the user to write intermediate layers—a much smaller effort than coming up with manual proofs.

Thankfully for developers, introducing an additional layer is done incrementally. The new layer is essentially a variation of the layer above or below it: either a more detailed version of the layer above it or a more abstract version of the layer below it. This helps keep the manual effort needed to introduce such layers small.

In the rest of this section, we discuss the strategies that we have developed and used to introduce intermediate layers, and walk through the process on a MultiPaxos example.

6.1 Intermediate Layers for Complexity

In most cases, the biggest challenge for a monolithic prover to automatically prove a refinement is its complexity. If the system is too complex, the prover either times out or runs out of memory. When this happens, we can split the refinement proof into two simpler refinement proofs by introducing an intermediate layer. We list here a number of ways in which such a split can simplify the proof burden. This list is extracted from our experience adding intermediate layers to facilitate refinement, and is not meant to be a complete enumeration of all possible layer-splitting strategies.

Abstract Away Messages Not Needed for Safety. Some of the messages used in the implementation may only be needed for liveness or performance, but not for safety. When trying to prove safety, those messages can be abstracted away in an intermediate layer: they are removed from the intermediate layer but kept in the implementation layer—itself proven to be a refinement of the intermediate layer.

For example, in MultiPaxos the current leader needs to periodically broadcast a heartbeat message to indicate that it is still alive. This message is not needed for safety and can therefore be removed in an intermediate layer—though it is preserved in the implementation layer. The resulting intermediate layer is now simpler and thus easier to prove equivalent to the specification.

Merge Multiple Transitions into One Abstract Transition. Sometimes, the intermediate layer can take an abstract transition which is broken into multiple transitions in the low-level implementation.

For example, in MultiPaxos the learner can only receive one vote (*two_b* message) from an acceptor at a time. But what the learner really needs is a quorum of messages to learn a value. In this case we can merge multiple transitions of receiving each message separately into one abstract transition of receiving a quorum, and remove local variables for temporary results. This significantly simplifies the intermediate layer, with fewer state variables and simpler transitions.

Simplify Local State and Requirements for Transitions. Implementation layers have to take into account implementation constraints: for example, a node can only read its local

state when taking a transition; and it cannot access messages sitting in the network. But intermediate layers are essentially proof constructs and thus do not need to respect such implementation constraints.

For example, in MultiPaxos, a node needs to maintain an explicit local history of previous *two_b* votes to construct its *one_b* promise to a new leader, since a promise message depends on previous votes. In an intermediate layer however, a node can directly access all sent messages in the network, thus eliminating the need for this local history. Moreover, in an implementation a node can only read its local history, thus requiring a proof that the local history is consistent with sent messages. In the intermediate layer, since the node has access to all sent messages, it can directly check that the *one_b* promise is consistent with the vote messages, thus eliminating the need for this proof.

6.2 Intermediate Layers for Decidability

When the verifier returns an explicit decidability error, it means our refinement is not in the EPR decidable logic [47] and may take forever to check. Such an issue is typically resolved by introducing an intermediate layer and a ghost state (also known as a derived relation [55]) to hide the existential quantifier creating the undecidability [55, 62]. We apply a similar technique in Sift.

For example, in MultiPaxos an acceptor needs to send its last votes for different slots in a *one_b* message to a new leader to decide what value to propose. When a proposer becomes a leader, it needs to have a quorum of *one_b* messages, resulting in the following $\forall Round\, \exists Votes$ alternation:

$$\forall N : Node, R : Round.\ \ quorum\_of(R).contains(N) \implies \exists V : votes.\ \ one\_b(N,R,V)$$

The alternation of the $\forall$ and $\exists$ quantifiers, along with the inductive invariant, means that this proposition is outside the decidable logic of EPR. We leverage results from a followup work on Ivy [62], and introduce an intermediate layer to abstract away the payload (previous votes), thereby breaking the quantifier alternation. In this case, we only need an intermediate-layer state $joined\_round(N,R)$ to represent $\exists V.\ one\_b(N,R,V)$.

7 Evaluation

We evaluate Sift by using it to formally verify the correctness of six implementations of distributed systems: a *leader election* protocol (Section 7.1), a *distributed lock* protocol (Section 7.2), a *two-phase commit* protocol (Section 7.3), a *sharded hash table* (SHT, Section 7.4), and two consensus protocols: *Raft* (Section 7.5) and *MultiPaxos* (Section 7.6). We use Ivy to implement these systems, and extract the executable code to C++ using Ivy's built-in translator. For the more complex systems (*SHT*, *Raft* and *MultiPaxos*), we also perform a performance evaluation (Section 7.7) to demonstrate our automated approach does not impact the performance of implementations.

For all systems in our evaluation, we consider crash failures and an asynchronous network, which can arbitrarily delay, drop, or duplicate messages. Both of these can be easily implemented in Ivy. Note that since Sift (like all its predecessors that also target automation) does not support liveness proofs, it does not need to explicitly reason about crash failures—a crash results in a machine no longer taking any steps and thus has no effect on safety properties.

We find that we are able to prove these complex systems with little manual effort within a reasonable memory and time budget, using IC3PO [27] as our monolithic prover. Our verification results are in Table 1. The complexity of different systems is illustrated by the number of different types that are needed to express state transitions for a given system. For example, for the leader election protocol, there are just two types: *node* and *id*. In contrast, MultiPaxos contains 14 different types, e.g., *round*, *inst*, *value*, *time*, *node*, etc.

We now give details about the proofs of the aforementioned systems, followed with a performance evaluation (Section-7.7) of three of the more complex resulting implementations (i.e., SHT, Raft and Paxos). We ran our performance experiments on a cluster where nodes have a 16-core Intel Xeon E5-2667 v4 @3.20 GHz processor and are connected with a 10 GB Ethernet connection running Ubuntu 16.04. All our implementation and artifact can be found in GitHub [49]

7.1 Leader Election

The *leader election* protocol aims to elect a unique leader from a ring with an unbounded number of nodes with unique integer IDs [13, 50, 56]. The specification layer dictates a single action the system can take: elect a node as the leader, under the condition that no other node is already the leader. This layer contains 13 lines of Ivy code.

In the implementation layer, the nodes are totally ordered in a ring so that every node has a next node. A node n has two valid actions: (a) periodically send its ID $idn(n)$ to the next node in the ring; or (b) forward an ID i received from its predecessor if $i > idn(n)$. Once n receives its $idn(n)$, it knows that no other node in the system has a larger ID, and can now safely become the leader. The implementation layer consists of 28 lines of Ivy code.

To prove refinement between the implementation and the specification layers, we ensure that when a message stating that a leader is elected is sent in the implementation, the destination of the message should correspond to the leader node in the specification.

We perform a manual, albeit trivial, syntactic change to the specification layer to convert one precondition into an

System	Proof Effort	Refinement	# of types	Solution to Preconditions	# of Clauses in Invariant	Time (sec)	Memory (MB)
Leader Election	< 5 min	spec to impl	2	1 if-statement	6	196	1744
Distributed Lock	< 5 min	spec to impl	2	1 if-statement	8	111	425
Two-Phase Commit	< 5 min	spec to impl	4	3 if-statements	12	613	815
SHT	< 30 min	spec to impl	7	5 invariants, 2 if-statements	13	1021	856
Raft	1 person-month	spec to layer 0	6	manual			
		layer 0 to layer 1	6	15 invariants	22	787	4178
		layer 1 to impl	10	15 invariants, 1 if-statement	17	1239	2981
MultiPaxos	Previously proved	spec to layer 0	9	manual			
	3 person-weeks	layer 0 to layer 1	9	7 invariants, 2 if-statements	12	49	249
		layer 1 to layer 2	11	8 invariants, 8 if-statements	21	258	719
		layer 2 to layer 3	11	19 invariants	28	841	1935
		layer 3 to impl	14	19 invariants	25	196	398

Table 1: Summary of our six distributed systems; "spec" stands for specification, "impl" stands for implementation, and "layer *i*" represents intermediate layers. The number of different types that are needed to express the state transition illustrates the complexity of different system.

if-statement, which takes less than 5 minutes. We then simply use Sift's encapsulation technique to convert the refinement between the implementation and specification layers into a monolithic proof that is proven automatically by IC3PO.

7.2 Distributed Lock

The *distributed lock* protocol [34, 50, 56] models an unbounded number of nodes that transfer the ownership of a single lock. In this system, the ownership of a lock is associated with an ever-increasing epoch: only one node can own the lock at each epoch. This makes for a concise specification layer—12 lines of Ivy code—that only contains a lock history to indicate which node holds the lock at every epoch.

In the implementation layer, there are two possible transitions for a node: (a) transfer the lock if it holds the lock; or (b) accept the lock and jump to a higher epoch by sending a *locked* message to indicate ownership. This implementation has 35 lines of Ivy code.

The refinement property in this system is that all *locked* messages should have a corresponding node in the specification layer's lock history.

The only manual effort involved in this proof is converting one precondition to an if-statement in the specification layer, which takes less than 5 minutes. After this transformation, we can use the encapsulation technique from Sift to convert the refinement between the implementation and specification layers into a monolithic proof, and prove the *locked* message is equivalent to the lock history.

7.3 Two-Phase Commit

The *two-phase commit* protocol [31] is used by a group of nodes, known as resource managers (RMs), to coordinate the decision on whether to abort or commit a transaction. The RMs vote to either commit or abort the proposed transaction and a transaction manager (TM) node is in charge of coordinating the decision-making procedure.

The specification layer of this system uses the Transaction Commit protocol by Lamport [30, Sec. 2] translated from TLA+ [45] to Ivy. The safety property does not allow a node to commit if another node aborts. The specification contains 54 lines of Ivy code.

The implementation of this system is an Ivy translation inspired by the TLA+ specification of Two-Phase Commit [30, Sec. 3]. This layer introduces a special TM node, which coordinates all RMs. An RM can send a *Prepared* message to the TM when transiting into the *prepared* state, or unilaterally decide to abort. Upon receiving a *Prepared* message from every RM, the TM can decide to commit, broadcasting a *Commit* message to every RM node. The receipt of a *Commit* message from the TM allows an RM to decide to commit the transaction. This implementation of two-phase commit has 110 lines of code.

The refinement property between the implementation and specification ensures that all RMs commit or abort at the same time between the implementation and the specification.

After a trivial syntactic change converting preconditions to three if-statements in the specification layer, this refinement property is proven automatically.

7.4 Sharded Hash Table (SHT)

The Sharded Hash Table protocol was previously introduced as a running example in Section 4. Its specification is a simple key-value map processing read and write requests. We can automatically prove the refinement from the implementation to the specification, after converting preconditions to five invariants and two if-statements to guide IC3PO, as detailed in Section 5.2. Compared to IronKV (IronFleet's implementation of SHT), we simplify the delegate messages by transferring one key at a time. Transferring intervals of keys would require a loop iterating over keys and a loop invariant [59, 65], which cannot be found automatically by IC3PO.

The network interface for SHT is more complex than that of other systems. In particular, SHT's network interface requires that messages are not delivered twice, so that requests can only be committed once and only one node at a time can own a key. As this is not part of refinement, we leverage an existing proof [53] for these requirements.

7.5 Raft

Raft [54] implements a shared log among nodes, which can be used to implement a fault-tolerant distributed service. The log is maintained as a set of (index, value) pairs.

Raft is a *term*-based protocol. In each term, a node can be elected as the leader, append values to the log, and replicate its log to other nodes by sending an *append* message. For safety, each node maintains its own log and only votes for a leader whose log is not earlier than its own. When the leader receives reply messages for its *append* message from a majority of nodes , the leader can consider all previous log entries committed. This strategy ensures that all future leaders contain the committed log.

At the specification layer, Raft can commit a prefix to an index in the leader's log and ensure that only one value is committed at each index. The refinement property from the implementation to the specification ensures that they have the same log.

7.5.1 Intermediate Layers and Proof Effort

Our Raft implementation is similar to the previous Ivy implementation of Raft [62] with 212 lines of code. Due to undecidability, we could not refine the implementation to the specification directly. Instead, we build a first intermediate layer—layer 0—to separate the quantifier alternation (as outlined in Section 6.2). We tried to prove the refinement from specification to layer 0 automatically, but the inductive invariant contains complex quantifier alternations, which IC3PO was unable to handle. As a result, we manually prove the refinement from specification to layer 0. The refinement from spec to layer 0 took two person-weeks (including understanding the protocol). Layer 0 contains 143 lines of code.

From layer 0, the implementation is still too complex to refine directly using IC3PO. We introduce another intermediate layer, layer 1, to help IC3PO automatically prove the refinement. To write layer 1, we follow the strategies presented in Section 6, specifically by merging actions into one abstract action. In the abstract action a node can receive a quorum of messages at once, rather than receiving each of them individually in separate transitions. Layer 1 changes 57 lines from layer 0. We spent another two person-weeks to identify this intermediate layer and debug our implementation.

Overall, we were able to complete the proof of Raft in one person month, which compares favorably to the three person months needed by the original proof [62] written in Ivy. This reduction was the result of using a much higher degree of automation, by splitting the proof into layers and leveraging the power of IC3PO to prove each refinement between consecutive layers.

7.6 MultiPaxos

MultiPaxos [43, 44] is a common consensus protocol that is widely used in industry (e.g., Chubby [11], Megastore [2], and Spanner [19]). However, MultiPaxos is notoriously complex and difficult to verify.

At the specification level, *MultiPaxos* maintains an array of values; some that have been decided (i.e., agreed upon and finalized) and some that are empty. The only possible transition in the specification is to add a new decided value to this array. Similar to Raft, our refinement ensures that the implementation maintains the same values as the array in the specification.

The implementation of MultiPaxos is very similar to that of Raft but uses different strategies to ensure safety. In Raft, the leader can only be a node with the most up-to-date logs, while MultiPaxos relies on the messages from other nodes to generate an up-to-date log for the new leader.

7.6.1 Intermediate Layers and Proof Effort

Our design of the MultiPaxos protocol is inspired by previous work on expressing Paxos and MultiPaxos in the EPR decidable logic [55, 62]. Our evaluation uses the MultiPaxos implementation from [62], removing certain re-transmissions that are unnecessary for safety to simplify the refinement.

Since proving refinement directly between the implementation layer and the specification layer would introduce undecidability (see Section 6.2), we initially introduce a single intermediate layer, layer 0, to circumvent this undecidability. Moreover, as the refinement from the specification to layer 0 contains complex quantifier alternations that are too hard for IC3PO to prove automatically, we borrow the existing manual proof from Ivy. Layer 0 contains 88 lines of Ivy code.

After addressing undecidability concerns through layer 0, we found that a direct refinement from the implementation to

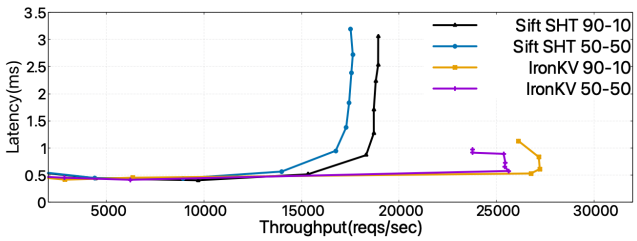

Figure 3: SHT performance

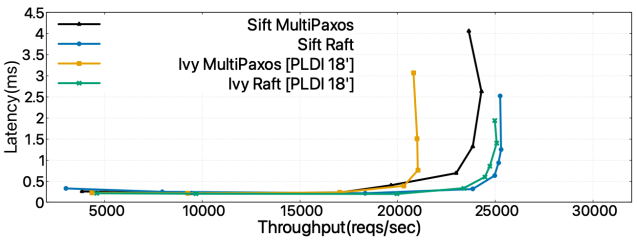

Figure 4: Raft and MultiPaxos performance

layer 0 remains infeasible for IC3PO. Using our divide-and-conquer technique, we added three additional intermediate layers to simplify this refinement. Following the strategies outlined in Section 6.1, we first added a layer 1 that abstracts away liveness messages and merges transitions to receive a quorum of messages. We augmented this by introducing a layer 2 that uses a local variable to track the current round, receives one *two_b* message, and keeps track of when a valid quorum can be formed. We then introduced a final intermediate layer that more closely resembles the implementation by using an array to track previous voted values for acceptors, and restricting a node to only receive one message during a transition.

With the addition of the four intermediate layers, Sift splits the complex refinement proof into manageable pieces, where each refinement between layers is amenable to automated verification. Producing the three intermediate layers (layers 1, 2, and 3) and converting the necessary preconditions to invariants is still a non-trivial task which takes about two person-weeks. About one third of the time is spent waiting for IC3PO to run out of time or memory, which indicates that another layer is needed (step ⑤ in Figure 1). While non-negligible, this manual effort is significantly less than the original attempt in Ivy, which was two person-months to refine layer 0 to the implementation [62].

7.7 Performance Evaluation

7.7.1 SHT Performance

We compare the throughput and latency of our verified Sift implementation of SHT with IronKV [34], as shown in Figure 3. IronKV is the closest verified implementation of a SHT that we could compare against. The SHT cluster was preloaded with 1,000 keys delegated evenly across the three nodes and serviced requests from an increasing number of clients in a closed loop. In one experiment, client processes send an even 50/50 mix of randomized GET and SET requests. We further increase the percentage of GET requests to 90%. IronKV scales about 25% better than our version of SHT. The disparity in performance between these two systems can be attributed to both unoptimized generated C++ from Ivy and design choices made in IronKV, which added extra manual proof complexity for the sake of performance purposes, such as an efficient delegation map data structure that each node maintains and consists of 833 lines of Dafny code.

7.7.2 Raft and MultiPaxos Performance

We evaluated the performance of the verified Sift implementations of Raft and MultiPaxos by varying the load of each system with an increasing number of clients submitting requests in a closed loop, as shown in Figure 4. For both systems, the experimental setup consists of three replicas on separate machines, with a fourth machine containing the client processes.

We tried to compare the performance of these systems with IronRSL, IronFleet's verified Paxos-based replicated state machine library, but the performance results of IronRSL were not reproducible for a direct comparison. By re-running the original implementation from the IronFleet paper [34], we found the performance for IronRSL to be lower than originally reported [34, Sec. 7][1]. The performance of our implementation of MultiPaxos, which does not support batching, is comparable to the results reported for IronRSL in non-batch mode [34, Fig. 13].

We do compare both Raft and MultiPaxos with the Ivy-based manually-verified implementations [62]. The performance of Raft is almost identical to the version of Raft, but we find that our MultiPaxos system exceeds the performance of MultiPaxos from that work. These results show that the automation and reduced proof effort gained by using Sift does not impact the performance of either system.

8 Limitations and Future Directions

Our experience with Sift suggests that it advances what is possible in the realm of automated verification of complex systems. For all of its successes, however, there are still more steps to be taken in this direction.

- **Automating simple transformations.** While Sift greatly increases the automation of complex refinement proofs, parts of the methodology still require manual effort that could potentially be automated, such as converting assertions to if-

[1] Even after close discussions with two of the IronFleet authors, this discrepancy was not resolved. They attributed this to possible code changes between what was originally evaluated and the currently available code.

statements and transforming to invariants through automatic computation of weakest preconditions [22].

• **Loop invariants.** Certain complex systems, such as SHT, may require loop invariants to prove optimizations that are added to enhance the performance of executable code. Loop invariants are similar to regular inductive invariants, in that both are inductive under some transitions. As described in Sections 7.4 and 7.6.1, any loop invariant in Sift must currently be written manually. In the future, we hope to add support for automatic deriviation of loop invariants in Sift, by building further on the existing literature [59, 65].

• **Leveraging multiple monolithic provers.** As shown in recent works [27, 33, 66], different monolithic provers show complementary strengths in different scenarios. Since the design of Sift is independent of the choice of monolithic prover, we plan to employ a portfolio of monolithic provers in parallel to derive refinement proofs with even higher scalability.

9 Related Work

We now provide a summary on previous efforts relevant to applying formal methods to verify distributed systems.

Automated Verification. With the advancements in automated reasoning [6, 20, 36] and abstraction techniques [3, 16, 29], automatically verifying correctness through model checking [17, 58] has significantly improved in different domains, both for hardware [9, 10, 23, 26, 61] and software [3, 7, 8, 37, 41]. However, model checking still does not scale well to large complex systems, due to state-space explosion [18, 21].

More recently, several approaches [24, 27, 33, 38, 40, 50, 66] have extended induction-based model checking [10, 23] to automatically infer inductive invariants for infinite-state distributed protocols. I4 [50] leverages the regularity of distributed protocols, combining finite model checking with unbounded reasoning in distributed protocols. IC3PO [27], described in detail in Section 3.3, incorporates invariant generalization with model checking for better scalability. SWISS [33] derives an inductive invariant by performing an exhaustive search over candidate invariants in an optimized invariant search space. DistAI [66] uses a data-driven approach and is guaranteed to find a universally-quantified inductive invariant in finite time.

All the aforementioned techniques [24, 27, 33, 38, 40, 50, 66], however, target monolithic, single-layer verification, primarily at the protocol level, and cannot scale to detailed system implementations. In contrast, our approach combines these monolithic provers with the well-founded concepts of refinement [1, 25, 42] to scale verification all the way to complex executable implementations.

Systems Verification. Much effort has gone to verifying real systems, including OS kernels [15, 32, 39, 52], file, and storage systems [12, 14, 67]. These works provide strong guarantees of correctness, but at the cost of extensive manual effort; Sift, by contrast, requires little manual proof effort while verifying systems of considerable complexity, such as MultiPaxos.

Within the realm of distributed systems, there have been attempts at manually verifying implementations of protocols [60, 64]. Ivy [56] requires the developer to iteratively refine an invariant until an inductive invariant is identified. IronFleet [34] and Verdi [63] have been used to verify practical implementations of distributed systems. In stark contrast to our work, all three approaches rely on considerable amounts of manual effort (in the order of person months) to complete a proof of correctness. Additionally, while IronFleet always uses three layers of refinement (i.e., specification, protocol, and implementation), most of the distributed systems we verify are refined directly from an implementation to a specification, with intermediate layers only added when needed to reduce the proof complexity for our monolithic provers.

More recently, Lorch et al. [48] presented Armada, a tool designed to verify concurrent programs. While Armada has some superficial similarities to Sift—namely the use of refinement and automation—it is in fact drastically different. It operates in an environment almost diametrically opposed to that of Sift: single-machine, multi-threaded code where communication happens via shared memory, as opposed to Sift's sequential execution on a distributed system where communication happens via message passing. Additionally, while Armada makes heavy use of automation to generate proofs, it still requires its users to write significant parts of the proof—hundreds of lines of code—manually.

10 Conclusion

This paper introduces Sift, a novel two-tier methodology that combines the power of refinement with the ability to automate proofs. Sift decomposes the proofs of complex distributed implementations into a number of refinement steps, each of which is amenable to automation. We use Sift to prove the correctness of six distributed implementations—including the notorious MultiPaxos—none of which had an automated proof before. Our evaluation shows that this combination of refinement and automation lets us verify complex distributed implementations with little manual effort.

Acknowledgements

We thank the anonymous reviewers for their useful feedback in improving this paper. This work was supported by the National Science Foundation under grant No 2018915, and by an Amazon Research Award.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–

284, 1991.

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[3] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *International Conference on Integrated Formal Methods*, pages 1–20. Springer, 2004.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[6] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[7] D. Beyer. Software verification: 10th comparative evaluation (sv-comp 2021). *Tools and Algorithms for the Construction and Analysis of Systems*, 12652:401, 2021.

[8] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[9] A. Biere, N. Froleyks, and M. Preiner. Hardware model checking competition (HWMCC) 2020. `http://fmv.jku.at/hwmcc20`.

[10] A. R. Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[11] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[12] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1054–1068, New York, NY, USA, 2019. Association for Computing Machinery.

[13] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[14] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[15] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 431–447, New York, NY, USA, 2016. Association for Computing Machinery.

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[17] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[18] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, Oct. 2012. USENIX Association.

[20] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[21] S. Demri, F. Laroussinie, and P. Schnoebelen. A parametric analysis of the state-explosion problem in model checking. *Journal of Computer and System Sciences*, 72(4):547–575, 2006.

[22] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[23] N. Een, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 125–134. FMCAD Inc, 2011.

[24] Y. M. Y. Feldman, J. R. Wilcox, S. Shoham, and M. Sagiv. Inferring inductive invariants from phase structures. In *Computer Aided Verification*, pages 405–425, Cham, 2019. Springer International Publishing.

[25] S. J. Garland and N. A. Lynch. Using i/o automata for developing distributed systems. *Foundations of component-based systems*, 13(285-312):5–2, 2000.

[26] A. Goel and K. Sakallah. Model checking of verilog rtl using ic3 with syntax-guided abstraction. In *NASA Formal Methods Symposium*. Springer, 2019.

[27] A. Goel and K. Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods Symposium*, pages 131–150. Springer, 2021.

[28] A. Goel and K. A. Sakallah. IC3PO: IC3 for Proving Protocol Properties. https://github.com/aman-goel/ic3po.

[29] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *International Conference on Computer Aided Verification*, pages 72–83. Springer, 1997.

[30] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[31] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[32] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[33] T. Hance, M. Heule, R. Martins, and B. Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, Apr. 2021.

[34] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[35] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, June 2017.

[36] M. Heule, M. Järvisalo, M. Suda, T. Balyo, C. Sinz, and A. Biere. The international SAT Competitions web page. http://www.satcompetition.org/.

[37] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[38] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM (JACM)*, 64(1):7, 2017.

[39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[40] J. R. Koenig, O. Padon, N. Immerman, and A. Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.

[41] D. Kroening and M. Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[42] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[43] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[44] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[45] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[46] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[47] H. R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317–353, 1980.

[48] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao. Armada: Low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 197–210, New York, NY, USA, 2020. Association for Computing Machinery.

[49] H. Ma, H. Ahmad, A. Goel, E. Goldweber, J.-B. Jeannin, M. Kapritsos, and B. Kasikci. Sift Artifact. https://github.com/GLaDOS-Michigan/Sift.

[50] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 370–384, 2019.

[51] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. Towards automatic inference of inductive invariants. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 30–36, 2019.

[52] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 293–304, New York, NY, USA, 2013. Association for Computing Machinery.

[53] K. L. McMillan. non-duplicating ordered transport service. https://github.com/microsoft/ivy/blob/master/doc/examples/sht/trans.md.

[54] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[55] O. Padon, G. Losa, M. Sagiv, and S. Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017.

[56] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. *ACM SIGPLAN Notices*, 51(6):614–630, 2016.

[57] R. Piskac, L. de Moura, and N. Bjørner. Deciding effectively propositional logic using dpll and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, Apr 2010.

[58] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[59] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana. Cln2inv: Learning loop invariants with continuous logic networks. In *International Conference on Learning Representations*, 2020.

[60] N. Schiper, V. Rahli, R. Van Renesse, M. Bickford, and R. L. Constable. Developing correctly replicated databases using formal tools. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 395–406. IEEE, 2014.

[61] B. L. Synthesis and V. Group. ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/, 2017.

[62] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–677, 2018.

[63] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices*, 50(6):357–368, 2015.

[64] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 154–165, New York, NY, USA, 2016. Association for Computing Machinery.

[65] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 106–120, New York, NY, USA, 2020. Association for Computing Machinery.

[66] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.

[67] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. Using concurrent relational logic with helpers for verifying the atomfs file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 259–274, New York, NY, USA, 2019. Association for Computing Machinery.