Kaizen: Building a Performant Blockchain System Verified for Consensus and Integrity

Faria Kalim*, Karl Palmskog†, Jayasi Mehar‡, Adithya Murali*, Indranil Gupta* and P. Madhusudan*
*University of Illinois at Urbana-Champaign †The University of Texas at Austin ‡Facebook
*{kalim2, adithya5, indy, madhu}@illinois.edu †palmskog@acm.org ‡jayasimehar@fb.com

***Abstract*—We report on the development of a blockchain system that is significantly verified and performant, detailing the design, proof, and system development based on a process of *continuous refinement*. We instantiate this framework to build, to the best of our knowledge, the first blockchain (Kaizen) that is performant and verified to a large degree, and a cryptocurrency protocol (KznCoin) over it. We experimentally compare its performance against the stock Bitcoin implementation.**

I. Introduction

Blockchains are used to build a variety of distributed systems, e.g., applications such as cryptocurrency (Bitcoin [1] and altcoins [2]), banking, finance, automobiles, health, supply-chain, and others [3], [4], [5]. In scenarios where there is traditionally a central ledger, blockchain-based approaches provide a democratized and decentralized alternative.

At the heart of any blockchain system is a distributed consensus protocol. This protocol ensures that the ledger is sharded into blocks, and that all nodes in the system agree on the order, content, and dependency verification of the blocks. Yet, today all blockchain and cryptocurrency with large deployments remain largely unverified implementations.

Formally verifying blockchains, and more broadly distributed systems, is extremely challenging, especially due to concurrency and asynchrony. Implementations of distributed systems running in production are large pieces of code with several features, functionalities, and optimizations, making them hard to verify ex-post. Consequently, the only techniques that are viable for realizing verified systems involve designing them correct by construction.

The main contribution of this paper is to build a blockchain system, and an associated cryptocurrency system that are both performant and have a verified consensus protocol at their core. Our approach ensures both safety and liveness properties of the blockchain and cryptocurrency system. We build this system using a novel *iterative refinement process* that combines interactive theorem proving and refinement verification using automated Floyd-Hoare style techniques.

The Kaizen refinement framework: There are two main techniques for constructing correct distributed software today. The first is a set of techniques based on using interactive theorem provers, such as Coq [6], where Coq type theory experts write a high-level abstract protocol and prove correctness properties for it [7]. This protocol can then be automatically translated to equivalent code in a functional language and deployed using a shim layer to a network to obtain working *reference implementations* of the basic protocol. However, there are several drawbacks to this—it is extremely hard to work further on the reference implementation to refine it to correct imperative and performant code, and to add more features to it to meet practical requirements for building applications.

The second technique, pioneered by the IronFleet systems [8], is to use a system such as Dafny to prove a system correct with respect to its specification via automated theorem proving (using SMT solvers) guided by manual annotations.

In this paper, we develop our Kaizen approach for blockchain protocols using a synergistic blend of the two techniques above. Kaizen is a variant of the IronFleet approach. Our approach is to start with an abstract distributed protocol (a consensus protocol for blockchain in our case) and prove correctness properties using an *interactive theorem prover*. We then refine the protocol to an imperative program using a program verifier that supports more automated verification, like Dafny, and similar to IronFleet. Interactive higher-order theorem provers such as Coq are extremely powerful, capable of virtually proving/checking any proof written by humans. IronFleet showed that correctness of several distributed protocols can be coaxed into first-order systems with standard theories as supported by Dafny (using ghost code and tactics). Yet, we believe that the flexibility afforded by interactive theorem provers significantly enlarge the scope of correctness proofs. Interactive theorem provers such as Coq have been used to prove complex properties involving cryptography and game-based proofs, pseudorandomess and probability [9], [10], [11].

The idea that building verified systems need both automated SMT reasoning (for speed and less manual annotation) and expressive logics (for proving more complex theorems that cannot be easily reduced to SMT reasoning) is not new. The Everest project [12] at Microsoft Research that aims to build a verified HTTPS ecosystem uses F*, and F* has grown over the years to support interactive theorem proving in addition to automated SMT-based reasoning [13]. Similarly, the newly designed Lean proof assistant [14] also combines SMT reasoning with interactive theorem proving.

The Kaizen blockchain and KznCoin: We build a blockchain system and a cryptocurrency over it by using a refinement framework as the one described above. Our starting point is a

†The first two authors contributed equally to this work.
‡Work done while at University of Illinois at Urbana-Champaign.

recently developed formalization of the blockchain/blockforest protocol in Coq by Pirlea and Sergey [15], where the abstract distributed protocol works in an asynchronous out-of-order message-passing environment and where several properties are proved regarding validity of the reachable global states of the system as well as an eventual consistency property. The goal we set for ourselves is to take this protocol expressed in Coq and to continuously refine it to a usable performant imperative implementation of an altcoin, with proven refinement guarantees. The proven refinement ensures that our core protocol implementation inherits all the safety and liveness (eventual consistency) properties proved for the abstract protocol in Coq.

We develop our KznCoin cryptocurrency on top of the Kaizen blockchain and evaluate its performance against the stock Bitcoin implementation [16]. We argue that the blockchain can be used as a backbone for a full-fledged Bitcoin-like cryptocurrency that uses standard proof-of work measures for controlling the growth of the chain.

In our experience in this project, we found our refinement methodology to be friendly to systems developers, as it allows teams with different expertise to collaboratively build a complex verified system. Experts in interactive theorem provers can work on designing abstract distributed protocols and prove complex properties. Meanwhile, systems engineers who are not necessarily trained in formal verification or abstract specification languages, can work on refining the protocol using imperative code using automated verification techniques, and concentrate their efforts on instantiating the protocol for particular applications and making them more performant, keeping control of the design at all times.

We focus our verification on the core blockchain consensus properties under crash failures and eventual message delivery. Proving correctness under Byzantine models remains an open problem in the community (see Section IX on related work and the discussion in Section VIII).

Our work can also be seen as exploring a new point in distributed systems verification, namely *computation-intensive* distributed systems. Systems like IronFleet focused on distributed systems that were not data-intensive (key-value store, state machine library), and involved relatively low computation. A blockchain system involves relatively intense local computation which nodes execute (work done on the blockchain, such as computing hashes, validating long stretches of a growing blockchain, etc.). Consequently, a lot of of effort is devoted to refining code with imperative datastructures that speed up this computation. We also perform a final refinement that is carefully argued, but not formally verified, to increase the system's performance.

The contributions of this paper are:

- The engineering of a performant blockchain (Kaizen) and a cryptocurrency over it (KznCoin) in C#, built using a continuous refinement that combines interactive theorem proving and automated refinement. Our core system first achieves a provable refinement of a core verified blockchain consensus protocol formulated by Pirlea and Sergey [15], preserving all their safety and liveness properties. We then add at the deployment layer a network shim and carefully argued (but not formally verified) refinements to gain further performance.
- An evaluation of the cryptocurrency against a reference Bitcoin implementation on a real network that shows that the system can be used as a backbone for a full-fledged Bitcoin-like cryptocurrency that uses standard proof-of work measures for controlling the growth of the chain.

We provide artifacts and supplementary material at this URL: https://madhu.cs.illinois.edu/kaizen

II. Background

A. Blockchains and Bitcoin

A blockchain is an open, distributed ledger that can record transactions between two parties in serializable order, in a verifiable and permanent way [17], [18]. Bitcoin is a decentralized cryptocurrency that uses Nakamoto consensus [1] to maintain a blockchain of transactions across an untrusted peer-to-peer network [18]. The peers in the network generally belong to different organizations or companies so that there is no single point of authority. These peers use gossip protocols to propagate transactions and blocks further in the network.

A node in the network can transfer currency from their address to another by creating a digitally signed transaction. Any new transaction must pass a reference to a previous transaction that the node might have received from another address. This ensures that all nodes in the network can cryptographically verify the creator of the transaction.

Tamper-proof chain: Each block has a unique hash and contains the hash of the preceding block, as well as the Merkle hash of its transactions. The first block in the chain, called the genesis block, is defined in the protocol. To effectively tamper with a transaction in the middle of the chain, an adversary would have to tamper with the transaction's hash in the block. This would cause the hash of the block to change. The adversary would then have to change the hash of the block in the next block in the chain until the head. Other nodes with the same chain would be able to detect this tampering.

Proof-of-Work: Nakamoto consensus proposes using proof-of-work to periodically choose the next node that gets to add a new block to the chain. Calculating proof-of-work refers to finding the solution to a cryptographic hash function H, where the solution is defined by string y, such that given string x as input, $H(y.x)$ — the hash of the concatenation of the two — is smaller than a target value. The miner that finds the proof first sends the newly minted block to other peers in the network. Other miners add the block to the chain if they are able to verify that the proof is correct.

Forking and Consensus: The blockchain forks when two valid blocks point to the same previous block. The protocol resolves forks using a set of rules that choose the branch that has more weight in terms of mining power. Consensus in Bitcoin is achieved when all nodes have the same blocks in their local best blockchain. Miners continue to add blocks to the best branch they know of, until one branch becomes significantly longer and is considered to be the main chain. Due to forking, transactions are usually not considered fully committed into a chain, unless there are a sufficient number of blocks (usually six in Bitcoin) after their containing block.

B. *Verification paradigms*

We assume familiarity with two verification paradigms— Coq-based interactive theorem proving and Dafny-based SMT-aided mostly automated verification.

C. *System Model*

Distributed System Model: We assume a distributed system where nodes are connected via a network where any node can send messages to any other node. While the nodes or network are not Byzantine, nodes may fail by crashing. Messages may be sent asynchronously and concurrently, and may be dropped, delivered out of order, or duplicated. Our KznCoin system is run on a real cluster and injected with real Bitcoin traces.

Trusted Computing Base (TCB): Our verification work makes reasonable assumptions about underlying systems: (a) Coq proof checkers, (b) Dafny and its provers, (c) compilers, (d) translation from Coq protocol to Dafny contracts (Phase 3), (e) reasonable assumptions (e.g., hash functions being injective), (f) a network shim layer that can send and receive messages (Section VI-B).

III. The Kaizen Framework

We describe our Kaizen framework at a high level[1]. The key idea is to combine interactive theorem proving in proof assistant (based on higher-order logic), with verified refinement using Floyd-Hoare style mostly-automated reasoning.

A. *Refinement methodology based on movers:*

Our refinement is divided in three stages. Our methodology is stylized to allow sequential refinement throughout. Focusing on sequential code reduces complexity of refinement in spite of asynchronous message passing in the distributed system.

The protocol is specified using code snippets of the form:

$$R;\ C;\ SU$$

where R is a sequence of receive-messages, C is a *terminating* local computation at the node, and SU is a set of send-messages and updates of the current state of the node. We assume that these rules are executing *atomically*, without interruption.

The rule-based approach has two advantages. First, we can distribute the protocol across any set of nodes, and not worry about interleavings where the code is (in global time) interrupted by computation at other nodes. The reason is that any such computation can be *reordered* into a computation where all blocks are executed atomically. The argument follows from the *mover* arguments for reduction by Lipton [19], as receive-messages are right-movers and send-messages are left-movers, while local computations are both left- and right-movers. Consequently in any global computation where this rule executes, interleaved by actions at other nodes, we can always reorder the execution to an equivalent execution where the rule is executed atomically.

[1]Kaizen means *continuous improvement.*

Second, this approach significantly reduces proof burden for safety properties, and allows us to work with a sequential verification tool like Dafny. We can refine each step of the protocol as sequential code, and not worry about interleavings. Such refinement is not new— the receive-compute-send/update is similar to the actor model of computation [20] and similar refinement styles have been explored before, for instance in the IronFleet system [8].

Third, when we prove that the local computation *terminates* (which we do in our proofs in Dafny), even *liveness* properties are preserved by the refinement. For every infinite computation in the refined system, since it cannot be stuck in a local computation in a rule, and since rules simulate the abstract protocol, we can find an equivalent infinite execution of the abstract protocol. This immediately argues that any liveness property in the abstract system is preserved in the refined system (we need to be careful that we do not introduce new messaging or change the system model for messages). We hence inherit all the safety and liveness properties of the original blockchain protocol of Pirlea and Sergey [15]. The IronFleet system [8] uses a similar methodology for proving liveness; however, in their system, liveness properties are proved on the abstract protocol using Dafny by embedding TLA, while in ours they are proved in Coq.

B. *Kaizen refinement stages*

The first of our three refinement stages consists of leveraging a proof assistant (Coq in our case) to develop an abstract, formal protocol describing the behavior of the distributed system and proving its correctness. The second stage consists of refining the formal description to imperative code in a verification-aware language and environment (Dafny in our case). The third stage consists of refinements using a practical programming language (C# in our case) for realizing an implementation of the distributed system. Figure 1 depicts the three stages and the phases within them.

IV. Stage I: Refinements in Coq

A. *Phase 1: Protocol specification and verification*

Here, we specify and prove the protocol in Coq. We encode the state of each network node using Coq's algebraic datatypes and use pure state update functions for network message processing (like work in [21]). Correctness proofs can then be performed over a *transition system* capturing network behavior where these functions are called per step.

State and transitions can involve *parameter* datatypes and functions over which the protocol is parameterized, along with *axioms* expressing the necessary properties those datatypes and functions must fulfill for the specification to hold. In principle, properties of data and functions unrelated to the axioms do not matter in this phase.

For Kaizen, we build on work by Pirlea and Sergey [15] to define a general blockchain consensus protocol where the

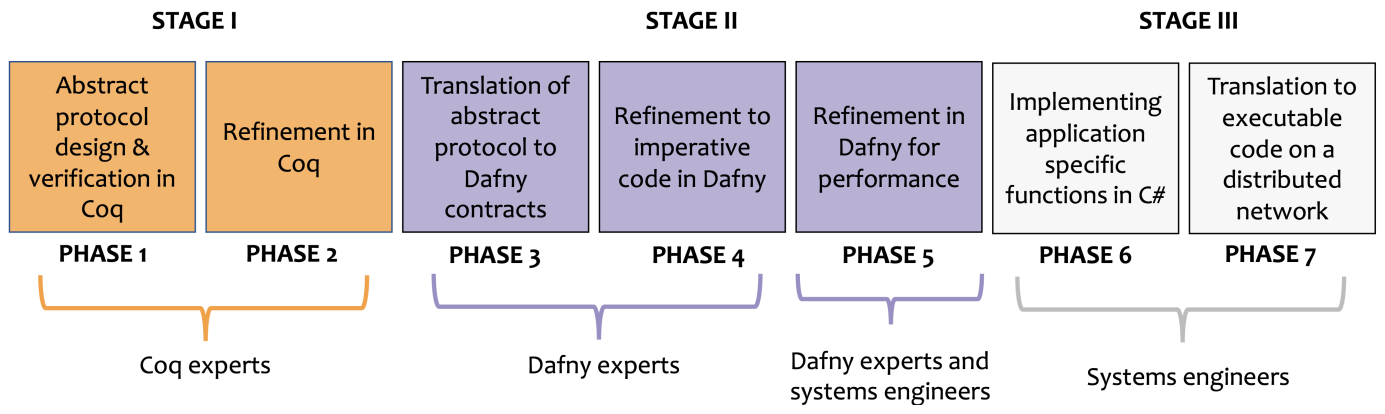
STAGE I
STAGE II
STAGE III
Abstract protocol design & verification in Coq
Refinement in Coq
Translation of abstract protocol to Dafny contracts
Refinement to imperative code in Dafny
Refinement in Dafny for performance
Implementing application specific functions in C#
Translation to executable code on a distributed network
PHASE 1
PHASE 2
PHASE 3
PHASE 4
PHASE 5
PHASE 6
PHASE 7
Coq experts
Dafny experts
Dafny experts and systems engineers
Systems engineers

Fig. 1: **The Kaizen Framework.**

key operations (e.g., transaction validation, procedures for generation of proofs of work or stake) are parameters. The main property proved in this work is eventual consistency: that in a quiescent state, all nodes have a canonical ledger. This is proved by proving a safety property that states that in a clique topology, there exists a global block forest which is a superset of the local forest of any node, which is a valid block forest that contains the *canonical ledger* (which is a ledger that is greater than or equal to any local ledger). This global block forest is in fact the one obtained by adding to the local block forest at any node all the blocks *in-flight* at the time to it. In a quiescent state, there are no in-flight messages and hence the local blockchains are in consensus. The protocol abstracts away several functions requiring only basic sanity properties on them; this includes transaction validation, blockchain validation, and the fork-chain rule.

The goal of our work is to build a blockchain system that provably inherits this eventual consistency property and the various safety properties of the above protocol. This constitutes the precise set of verified properties that we target.

The Coq encoding defines blocks as a datatype with a previous-block "pointer" hash, a list of transactions, and a proof-of-work/stake, as follows:

```
Record Block := mkB { prevBlockHash : Hash;
  txs : seq Transaction; proof : VProof }.
```

The state of a node in Kaizen is defined by its globally unique name, a list with the name of its peers, a list (pool) of transactions, and a forest of blocks, in the form of a finite map from hashes to blocks:

```
Record State := mkS { id: Address; peers: seq Address;
 forest: map Hash Block; txpool: seq Transaction }.
```

At any time, a node can obtain its longest known chain of blocks rooted in the special *genesis block*, where *longest* is defined according to a parameterized relation ("fork choice rule"), which is not concretely defined. All transactions in blocks in this chain are validated using the parameter function `txValid`. One axiom, used in our correctness proofs, states that `txValid` returns true when passed any transaction and the empty chain: $\forall$`t, txValid t [::]`. We check block validity via the function:

```
Definition valid_chain_block (bc:seq Block)
(b:Block):= VAF (proof b) bc (txs b) &&
 all [pred t | txValid t bc] (txs b)].
```

B. *Phase 2: Initial protocol refinement*

For KznCoin, we refine the abstract blockchain datatypes and functions according to the specification of Bitcoin [1], [16]. For example, representing byte vectors as lists of mathematical integers, the type of transactions, `Transaction`, is defined in Coq as follows:

```
Record TIn := mkTI { prevout_hash : seq Z;
 prevout_n : seq Z; scriptsig : seq Z;
 sequence : seq Z }.
Record TOut := mkTO { value : seq Z;
 scriptpubkey : seq Z }.
Record Transaction := mkT { version : seq Z;
 ins : seq TIn; outs : seq TOut; locktime : seq Z}.
```

Here, the definitions reflect that KznCoin transactions consist of a number of inputs (pointing to previous transactions) transferring cryptocurrency to a number of output addresses.

We also define block and transaction hashing functions, which are parameters in the abstract protocol, to use SHA-256 hashing as prescribed by Bitcoin, i.e., twice on properly concatenated vectors of bytes. We used Coq's module system to verify that the functions and data in our protocol refinement have the proper signatures, so that the correctness theorem can be reestablished [22].

Addressing Obstacles Faced: We encountered several obstacles from [15] that prevented us from capturing the Bitcoin specification. To overcome these obstacles, we made several important proof-preserving changes to the abstract protocol and related definitions in Coq; these changes were all approved by Pirlea and Sergey.

One obstacle was that the abstract protocol did not permit adding new transactions when blocks are minted. This rules out *coinbase transactions* that are added by miners to obtain a block reward. We resolved this by adjusting the function parameter `genProof`, used during block minting.

Another obstacle was that the validator acceptance function, `VAF`, which checks the validity of a proof-of-work or proof-of-stake, was not called except during block minting in the abstract protocol, while Bitcoin specifies that it has to be called on all blocks in a chain. We resolved this by adding the VAF call to the function valid_chain_block above, which is used when obtaining the longest chain from a block forest.

V. Stage II: Refinements in Dafny

In Stage II, we manually translate the data structures and functions in Stage I from Coq to Dafny, and refine the resulting definitions to performant imperative code in Dafny. Our implementation of KznCoin consists of several classes and methods whose contracts express that they conform to the behavior of some function translated from Coq. This contrasts with IronFleet's approach [8], wherein the equivalent translation is entirely within Dafny.

A. Phase 3: Protocol translation to Dafny

We manually translated all relevant datatypes, functions, and refinements from Phase 1 and Phase 2 to equivalent Dafny definitions. Due to the close similarities between Coq's programming language and Dafny's purely functional fragment, the translation was straightforward. For example, the Block and State datatypes (presented in Coq above) are defined in Dafny as:

```
datatype Block = Block(prevBlockHash:Hash,
 txs:seq<Transaction>, proof:VProof)
datatype State = Node(id:Address,
 peers:seq<Address>, forest:map<Hash,Block>,
 txpool:seq<Transaction>)
```

Preparing to Transition from Functional to Imperative Code: As an initial step, we set up a Dafny class that holds the node state and implements methods for protocol state updates:

```
class StateImpl {
  var id :Address;
  var peers :...; var forest :...; var txpool :...;
  ghost var st:State;
  predicate Valid() { ...}
  method ProcMsgImpl(from:Address, msg:Message,
    ts:Timestamp) returns (pt:seq<Packet>)
  requires Valid(); ensures Valid();
  ensures st == procMsg(old(st), from, msg, ts).0;
  ensures pt == procMsg(old(st), from, msg, ts).1;
  { ...}
  method ProcIntImpl(tr:InternalTransition,
    ts:Timestamp) returns (pt:seq<Packet>)
  requires Valid(); ensures Valid();
  ensures st == procInt(old(st), tr, ts).0;
  ensures pt == procInt(old(st), tr, ts).1;
  { ...}
}
```

The variables peers, forest, and txpool can be of any (mutable) type as long as they have a *representation* in terms of the (immutable) algebraic datatypes in the State data type. For example, peers could be an array, which has a sequence representation. The Valid() predicate then asserts that the representations of the variables are consistent with what is stored in the ghost variable st, in the style advocated by Leino [23]. In turn, this means that the contracts for ProcMsgImpl and ProcIntImpl express that these methods perform state updates consistent with the Coq functions procMsg and procInt (so the Coq correctness proofs hold for systems implemented using StateImpl objects).

In essence, with the Dafny modules and the StateImpl class, we have obtained constraints under which all further refinement must be performed, as well as the interface for unverified code to call our verified code.

B. Phase 4: Refinement to imperative code in Dafny

Next, we refine the functional code into imperative code. We start by implementing Dafny methods for each function called in procMsg and procInt. Because the code of a method corresponds closely to its specification, and we use the same algebraic datatypes, it is straightforward to do this translation from functional code into Dafny, and to prove the specification correct. We also encode stubs for methods that are deferred to C# code inside Dafny, and write the corresponding method signatures in C# code that gets linked to the code compiled from Dafny. This process yields an executable and verified (though incomplete) implementation of StateImpl.

Next, we iteratively replace functional code in the verified Dafny implementation. We (a) replace algebraic datatypes with mutable datastructures such as arrays and classes, and (b) replace functional idioms such as recursion and pattern-matching by imperative idioms such as loops and accesses to mutable fields. For example, we refined the Dafny sequences of transactions in the node state (txpool) into linked lists that also store the list size, and with ghost variables capturing the heaplet of the list which is needed for mutable datastructure reasoning. We then systematically replace all methods acting on sequences of transactions to instead act on linked lists, while preserving their specification (now in terms of the sequence representing the linked list).

C. Phase 5: Performance optimizations & Dafny refinements

Refinements here optimize performance in three ways:

Data structure refinements: A prime example of these refinements in our work involves our encoding of a *block forest* as a finite map from Hashes to Blocks, similar to the formalization in the work by Pirlea and Sergey [15]. Our initial implementation used Dafny's map type, which we found to be inefficient due to the number of accesses (e.g., by the method that returns the FCR-longest valid chain in the block forest). Instead we store hashes using a binary search tree and implement a map over it. We define this in Dafny, and prove that the operations on it—including addition, removal, lookup, and copy—satisfy the same contracts as that of the map structure. This took about 680 lines of Dafny code.

Refinements to operations on data structures: An example of such an optimization is our method to append two linked lists (defined in Phase 4 Section V-B). Its implementation added one element at a time as this made verification easy. But this is inefficient as it involved multiple passes and pointer manipulations. We replaced this with a more efficient one-pass algorithm. This involved a notion of validity of linked list *segments* and the maintenance of complex invariants.

Recursive to iterative implementations: We identified about 20 auxiliary functions that were implemented recursively and refined them to more efficient iterative versions. For example, the method to validate a list of transactions with respect to a blockchain would validate the first transaction and then recursively compute on the rest of the list. This had to be reimplemented using iteration (using array semantics of the list containing the transactions).

The first two classes above are hard. Defining data structures that are allocated on the heap often have definitions (of the validity of their representations) that are recursive and involve complex relationships between their attributes.

VI. Stage III: Refinements in C#

We build KznCoin from our verified system Kaizen by adding Bitcoin-specific functionality (Section VI-A) and connect it to other peers via a network shim layer (Section VI-B).

A. Phase 6: Application specific functions

We fill in methods that were left unspecified in the abstract Coq/Dafny protocol. Any methods that needed verification were already handled through the separation described Phase 4. Consequently, we only need to add methods in C# that do not need verification.

- **Transaction Validity (txValid):** In KznCoin, a node must validate each transaction to ensure that no transaction with the same hash exists in the chain already, the transaction's parent transactions exist, and the transaction's parent transactions are unspent to prevent double-spending attacks.
- **Validator Acceptor Function (VAF):** We found that the specification in [15] did not completely satisfy Bitcoin semantics and thus we modified it while developing KznCoin (e.g., it did not support coinbase transactions; see Phase 1). Apart from a duplicate block check, we also check that the calculated value of its cryptographic hash proof is correct.
- **Generate Proof:** KznCoin uses proof-of-work to mint blocks. Every miner solves a hash problem with difficulty based on the nonce value. We set this nonce difficulty level to a negligible value during our evaluation to measure the overhead of only the blockchain mechanisms.
- **Fork Chain Rule (FCR):** This function is used to compare the weights of two chains and establish a total order between them. The specification [15] is indifferent to how such an order is established. We follow the Bitcoin semantics to pick the chain with the highest weight.

B. Phase 7: Building a connected system & optimizations

Optimizations: Systems needs to be amenable to optimizations – this is part of their lifecycle. We improve KznCoin's performance by adding several unverified optimizations:
1. Extracting the chain from the Block Tree: The base KznCoin stores the block tree as a top-down binary search tree. This makes it time-consuming to extract all chains and compare them to the longest chain. Instead, we extract only the heaviest chain. We keep track of the weight of all leaf nodes. We extract the chain by following the leaf node with the heaviest weight all the way up to the genesis block (instead of top-down). We can see why this follows the contract.
2. Processing only new blocks for chain extraction: Chain extraction was computationally expensive since to update weights of leaf pointers, we traversed through all blocks in the block tree. Instead, we now update pointers by only traversing through newly arrived or minted blocks. This also allows KznCoin to quickly return a cached copy of the heaviest chain if no new blocks have arrived/minted.
3. Replacing binary search tree with C# dictionary : We found that the binary search tree was a bottleneck also due to its lack of a self-balancing mechanism. To address this bottleneck, we replaced it with a C# dictionary.
4. Removing unnecessary sorting: At least one assertion was superfluous and removable without affecting the proof. In particular, we removed the assertion that compared if two collections were equal and was expensive, as it required sorting the two collections for comparison.

The above optimizations were not formally verified. Though they all seem simple and correct, they are not all easy to realize strictly as refinements to the verified code. The problem of redefining all the layers starting from Phase 1 in a way that allows such performance optimizations as refinements seems hard, and is left to future work.
Building a connected system: We connect our implementation to other peers via a network shim layer that uses lightweight UDP.

A peer sends a `connect` message to announce itself to the network, and an `address` message to propagate information about newly discovered peers. The `transaction` and `block` messages announce new transactions and blocks respectively. An `inv` message is sent periodically to inform other nodes about the transactions and blocks a peer holds, represented by their hash. The `inv` can be responded with a `get-data` message, requesting the full transaction/block by sending its hash.

Due to our sequentiality assumption in our specifications (Section III), the shim layer at each node uses one thread for interactions with the KznCoin implementation. Since the network is asynchronous, we speed up message processing by using a thread-safe queue to store incoming messages.
Optimizing the Shim: Initially, messages were processed in a FIFO manner from the received queue. However, we observed (in our evaluation) that the queue sizes grew monotonically. We address this via several optimizations:
1. *Refreshing stale `inv` messages in the queue:* We avoid processing of messages that are obsolete by subsequent messages from the same sender. This is done via an up-to-date map of all unprocessed `inv` messages, indexed by sender address. When a new message is received from the same sender, we update that sender's entry (creating one if sender is absent).
2. *Prioritizing `block` messages:* Whenever we receive a `block` message, we add it to the front of the queue. This prevents other nodes from repeating effort for transactions that are already a part of the block.
3. *Filtering duplicate `get-data` messages:* Every `inv` message needs a `get-data` message response sent back to the sender. In practice, duplicate `inv`'s are received. We curtail

the overhead of processing and generating duplicate responses via a cache of all transactions and blocks already requested.

These optimizations prune messages as the protocol would do. However, these are not verified to be equivalent in the present system and are part of the Trusted Computing Base.

VII. Evaluation

A. Experimental Setup

KznCoin is not as scalable as Bitcoin yet. However, the goal of our evaluation is to prove that the Kaizen methodology has the potential to allow KznCoin to approach Bitcoin performance at some scales.

Performance Evaluation—Metrics: We compare the performance of KznCoin (with optimizations enabled) with the stock Bitcoin implementation [16] with respect to two important metrics: the time it takes all peers to globally achieve consensus, and the amount of time required to mint a block. These two metrics directly impact end users whose transactions only make it into the global ledger once blocks are minted and all nodes decide on the same chain.

Experimental Setup: We use a 30-node Emulab cluster [24]. Each node has a 2.4 GHz 64-bit 8-Core processor and a 64 GB RAM, and is connected using a 1 Gbps network. To measure performance independent of proof-of-work overhead, we set the difficulty of the hash problem to a negligible value. For fairness, we set the stock Bitcoin client to its regression testing mode [25], which uses a similar low hash problem setting.

Workload: We use a realistic workload by obtaining traces of the arrival times of 50 transactions from the "Blockchain" website [26], a software platform for digital assets that offers real time transaction data for developers to analyze the Bitcoin network. We replay the trace by announcing transactions in a round-robin fashion across peers in the network, using the same inter-arrival times as the trace.

B. Results

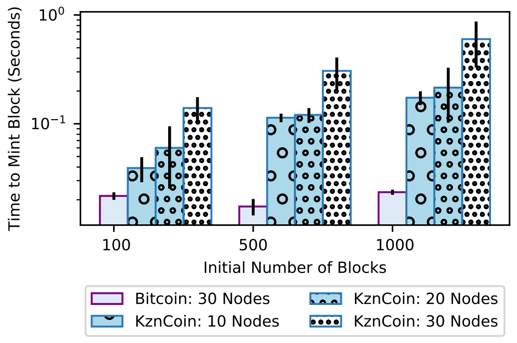

Fig. 2: **Minting Times** KznCoin vs. Bitcoin. Time required to mint a new block from new transactions.

Minting Times: Figure 2 shows the time to mint a block after sufficient transactions are received. We vary the cluster size and the size of the blockchain that the experiment is initially started with, and observe the effect on performance. KznCoin can generate a block in less than a second. KznCoin's performance degrades slightly with both increasing chain size and cluster size. For every single minted block, KznCoin extracts and validates the chain in which the block belongs from the block tree. This process becomes time-consuming as the chain size increases. Larger clusters also cause more forking, increasing overhead. In the worst case, KznCoin takes 0.59 seconds to mint blocks on a 30-node with a 1000 block chain. However, this is still practical as it is much smaller than the time to do proof of work today (about 10 minutes [27]).

Time to Achieve Consensus: Figure 3 shows the time for KznCoin to reach consensus, i.e., time from the beginning of trace injection until all nodes have consistent state. First, KznCoin's performance degrades gracefully with increasing chain size–0.58 s for 100 blocks vs. 2.1 s for 1000 blocks. Second, Bitcoin degrades less at scale because of its heavily-optimized C++ implementation.

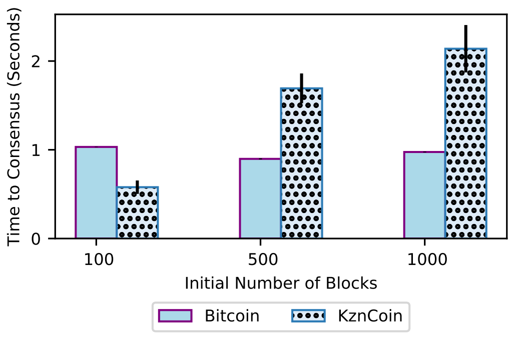

Fig. 3: **Time to Consensus:** KznCoin vs. Bitcoin.

Scalability: Figure 4 shows the effect of increasing cluster sizes and the length of the input workload on the performance of KznCoin. Each node is configured to mint a block after it has received two transactions not already present in the chain.

Figure 4 shows that for a shorter trace length, KznCoin's consensus time falls from 4.2 s to 2.1 s when cluster size increases from 20 to 30. As the trace length is equal, each node in the 20-node cluster receives more transactions to process than a node in the 30-node cluster, leading to longer convergence times. Nodes on the 30 node cluster receive less transactions and immediately mint blocks, reducing forking.

When the trace size increases to 250, consensus times are greater for the 10-node cluster than for 20-nodes since each node has more transactions to mint blocks from. Forking is higher on the 30-node cluster because nodes disseminate transactions so quickly at the experiment start that many mint new blocks with subsets of the same transactions.

Development Effort: Table I shows the lines of code in different components that were built as part of the Kaizen framework. 4 developers were involved for 4 months (all part-time). 50% (2 of the 4) developers were verification experts, and 50% were distributed systems experts.

VIII. Lessons Learnt and Discussion

Lessons we expected: Our team consists of people with expertise in Coq, Dafny, system engineering, and some have

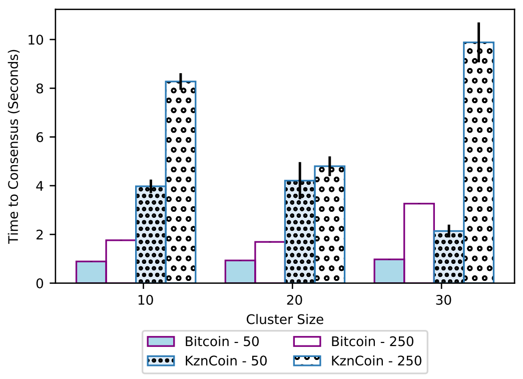

Fig. 4: **Scalability:** KznCoin vs. Bitcoin. The number in the legend indicates number of transactions in the input trace.

Component	Lines of Code
Additions to the original Coq proof	1119
Blockchain Refinement in Dafny	4937
Bitcoin Refinement in C#	915
Network Shim in C#	4159

TABLE I: **Lines of proof and code.**

mixed expertise. While the subteams required a lot of communication (and a learning curve to recognize the strictness that verified refinement imposes), we found that the teams could, for the most part, exercise their expertise in their domains.

We often encountered situations where a design change was required, typically to incorporate features (like coinbase transactions) or incorporate a different approach for performance. We were able to identify the appropriate abstraction to backtrack and make changes that would allow the required refinement. At the same time, making this iterative design process more agile, nimble, and fast, remains an open direction.

Lessons we did not expect: Going into development, we assumed that communication would be the aspect to optimize, not local computation. However, it turned out that computation was more important to optimize, primarily because of computing validity of blockforests.

Future Work: Our work builds on the work by Pirlea and Sergey [15], where properties are proven about the abstract protocol assuming only honest agents. Implementing a blockchain/bitcoin system with fully formally proven correctness guarantees for consensus and integrity including under Byzantine conditions, that also matches performance of stock Blockchain (possibly using concurrency) remains an important open problem [28], [29], [30]. It requires assumptions involving proof-of-work complexity and network speed, and formalizing this for our protocol in Coq is an interesting future direction. We believe Kaizen is the right framework to achieve this, but doing so would require modeling the protocol even at the first stage by keeping an eye on performance, anticipating the refined system that will be built, before proving properties of the protocol and refining it.

IX. Related Work

There is a lot of work on formalizing and analyzing consensus protocols for blockchains such as [28], [30]. The work by Pass, *et al.* shows that consensus mechanisms satisfy strong forms of consistency and liveness in an asynchronous network with adversarial delays that are *a-priori* bounded.

There is also work on other Byzantine Agreement protocols such as Algorand [31] and on proof-of-stake consensus protocols such as Ouroboros [29]. Such protocols can be implemented using the Kaizen framework as well.

Many recent lines of work are in systems verification. seL4 [32] pioneered a functionally correct, general-purpose OS kernel. However, it took 22 person-years to produce 8700 lines of verified C code. Ironclad Apps [33] allows users to securely transmit data to other machines with the guarantee that instructions executed there adhere to a formal behavioral specification. ExpressOS [34] proves key security invariants for an OS architecture, using Dafny and abstract interpretation of Code Contracts. IronFleet [8] proposes a methodology like ours that slices a system into (Dafny) layers to make verification of practical distributed system implementations feasible. They utilize their proposed methodology to build verified implementations of Paxos and a key-value store. The Everest project [12] uses F* [35] to develop a fully verified and usable replacement for HTTPS. The Ivy tool [36] leverages constrained but decidable theories in a framework for verifying safety properties of distributed systems. Systematic model-checking approaches as well as integration testing techniques have also been explored [37], [38].

Verdi [7] is a framework for implementing and verifying distributed systems in Coq. Verdi is used to verify a (non-Byzantine) fault tolerant key-value store using Raft consensus [21]. Rahli et al. [39] present a Coq framework for verifying safety of Byzantine fault-tolerant distributed systems (PBFT), based on an event-based reasoning approach similar to EventML, with extraction to OCaml. Sergey et al. [40] use distributed separation logic to verify a two-phase commit protocol in Coq. Authors of [41] build a total order broadcast protocol based on Paxos, specified in EventML [42] and proved it correct using NuPRL [43]. In contrast to Kaizen where we generate imperative code, all these approaches consider the compiled extracted functional code the final product.

CompCert [44] is a compiler for a subset of C verified using Coq. VST [45] builds on CompCert to define Verifiable C, a program logic for C. In principle, the approach can be used to refine functional to imperative C code completely inside Coq, but takes substantially more effort than Kaizen.

Ensemble [46] is one of the earliest distributed systems for group communication. It layers simple micro protocols to produce distributed systems, and performs cross-protocol optimizations. It is written in a functional language, manually translated to IO automata and verified using NuPRL. Kaizen separates out skills and required fewer man-months.

X. Acknowledgements

This work was supported in part by grants NSF CNS 1409416 and NSF CCF-1527395, and in part by gifts from Microsoft and Facebook.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] "Handbook of digital currency," D. L. K. Chuen, Ed. San Diego: Academic Press, 2015.

[3] I. Eyal, "Blockchain technology: Transforming libertarian cryptocurrency dreams to finance and banking realities," *Computer*, vol. 50, no. 9, pp. 38–49, 2017.

[4] P. Treleaven, R. G. Brown, and D. Yang, "Blockchain technology in finance," *Computer*, vol. 50, no. 9, pp. 14–17, 2017.

[5] A. Dorri, M. Steger, S. S. Kanhere, and R. Jurdak, "Blockchain: A distributed solution to automotive security and privacy," *IEEE Communications Magazine*, vol. 55, no. 12, pp. 119–125, 2017.

[6] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Berlin, Heidelberg: Springer, 2004.

[7] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems," in *Conference on Programming Language Design and Implementation*. ACM, 2015, pp. 357–368.

[8] 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 *Symposium on Operating Systems Principles*. ACM, 2015, pp. 1–17.

[9] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel, "Verified correctness and security of mbedTLS HMAC-DRBG," in *Conference on Computer and Communications Security*, 2017, pp. 2007–2020.

[10] A. W. Appel, "Verification of a cryptographic primitive: SHA-256," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 7:1–7:31, 2015.

[11] R. Affeldt, D. Nowak, and K. Yamada, "Certifying assembly with formal security proofs: The case of BBS," *Science of Computer Programming*, vol. 77, no. 10, pp. 1058 – 1074, 2012, aVoCS'09.

[12] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoué, "Everest: Towards a verified, drop-in replacement of HTTPS," in *Summit on Advances in Programming Languages*, 2017.

[13] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *Symposium on Principles of Programming Languages*. ACM, 2016, pp. 256–270.

[14] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean theorem prover (system description)," in *International Conference on Automated Deduction*, 2015, pp. 378–388.

[15] G. Pirlea and I. Sergey, "Mechanising blockchain consensus," in *Conference on Certified Programs and Proofs*. ACM, 2018, pp. 78–90.

[16] "Bitcoin," https://github.com/bitcoin/bitcoin, 2011, last visited: August 17, 2019.

[17] M. Iansiti and K. R. Lakhani, "The Truth About Blockchain," https://hbr.org/2017/01/the-truth-about-blockchain, 2017, last visited: August 17, 2019.

[18] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton, NJ, USA: Princeton University Press, 2016.

[19] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, Dec. 1975.

[20] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[21] 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 *Conference on Certified Programs and Proofs*. ACM, 2016, pp. 154–165.

[22] J.-C. Filliâtre and P. Letouzey, "Functors for proofs and programs," in *Programming Languages and Systems*. Berlin, Heidelberg: Springer, 2004, pp. 370–384.

[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Symposium on*

[23] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*. Berlin, Heidelberg: Springer, 2010, pp. 348–370.

Operating Systems Design and Implementation. Boston, MA: USENIX Association, 2002, pp. 255–270.

[25] "Bitcoin Regtest Mode," https://bitcoin.org/en/developer-examples#regtest-mode, 2011, last visited: August 17, 2019.

[26] "Blockchain," http://blockchain.info, 2017, last visited: August 17, 2019.

[27] A. Tar, "Proof of Work, Explained," https://cointelegraph.com/explained/proof-of-work-explained, 2017, last visited: August 17, 2019.

[28] J. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin backbone protocol: Analysis and applications," in *Advances in Cryptology - EUROCRYPT*. Berlin, Heidelberg: Springer, 2015, pp. 281–310.

[29] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *CRYPTO*. Springer, 2017, pp. 357–388.

[30] R. Pass, L. Seeman, and A. Shelat, "Analysis of the blockchain protocol in asynchronous networks," in *Advances in Cryptology - EUROCRYPT*. Cham: Springer, 2017, pp. 643–673.

[31] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68.

[32] 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 *Symposium on Operating Systems Principles*, 2009, pp. 207–220.

[33] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *Symposium on Operating Systems Principles*, 2014, pp. 165–181.

[34] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, "Verifying security invariants in ExpressOS," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013, pp. 293–304.

[35] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 17:1–17:29, 2017.

[36] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: Safety verification by interactive generalization," in *Conference on Programming Language Design and Implementation*. ACM, 2016, pp. 614–630.

[37] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," in *Conference on Programming Language Design and Implementation*. ACM, 2013, pp. 321–332.

[38] S. Liu, S. Nguyen, J. Ganhotra, M. R. Rahman, I. Gupta, and J. Meseguer, "Quantitative analysis of consistency in NoSQL key-value stores," in *Quantitative Evaluation of Systems*. Cham: Springer, 2015, pp. 228–243.

[39] V. Rahli, I. Vukotic, M. Völp, and P. Esteves-Verissimo, "Velisarios: Byzantine fault-tolerant protocols powered by Coq," in *Programming Languages and Systems*. Cham: Springer, 2018, pp. 619–650.

[40] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and Proving with Distributed Protocols," *PACMPL*, vol. 2, no. POPL, pp. 28:1–28:30, 2018.

[41] N. Schiper, V. Rahli, R. Van Renesse, M. Bickford, and R. L. Constable, "Developing correctly replicated databases using formal tools," in *International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 395–406.

[42] V. Rahli, "Interfacing with proof assistants for domain specific programming using EventML," in *International Workshop On User Interfaces for Theorem Provers*.

[43] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden *et al.*, *Implementing mathematics*. Prentice-Hall, 1986.

[44] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[45] A. W. Appel, "Verified software toolchain," in *Programming Languages and Systems*. Berlin, Heidelberg: Springer, 2011, pp. 1–17.

[46] M. G. Hayden, "The Ensemble system," Ph.D. dissertation, Ithaca, NY, USA, 1998.