

Formal Verification of the Ethereum 2.0 Beacon Chain^{*}

Franck Cassez (✉)^{1, }, Joanne Fuller¹, and Aditya Asgaonkar²

¹ ConsenSys, New York, USA

² Ethereum Foundation, Zug, Switzerland

franck.cassez@consensys.net joanne.fuller@consensys.net
aditya.asgaonkar@ethereum.org

Abstract. We report our experience in the formal verification of the reference implementation of the Beacon Chain. The Beacon Chain is the backbone component of the new Proof-of-Stake Ethereum 2.0 network: it is in charge of tracking information about the *validators*, their *stakes*, their *attestations* (votes) and if some validators are found to be dishonest, to *slash* them (they lose some of their stakes). The Beacon Chain is mission-critical and any bug in it could compromise the whole network. The *Beacon Chain reference implementation* developed by the Ethereum Foundation is written in Python, and provides a detailed operational description of the state machine each Beacon Chain’s network participant (node) must implement. We have formally specified and verified the absence of runtime errors in (a large and critical part of) the Beacon Chain reference implementation using the verification-friendly language Dafny. During the course of this work, we have uncovered several issues, proposed *verified* fixes. We have also synthesised *functional correctness specifications* that enable us to provide guarantees beyond runtime errors. Our software artefact with the code and proofs in Dafny is available at <https://github.com/ConsenSys/eth2.0-dafny>.

1 Introduction

The Ethereum network is gradually transitioning to a more secure, scalable and energy efficient *Proof-of-Stake (PoS) consensus protocol*, known as Ethereum 2.0 and based off GasperFFG [2]. The Proof-of-Stake discipline ensures that participants who propose (and vote) for blocks are chosen with a frequency that is proportional to their stakes. Another major feature of Ethereum 2.0 is *sharding* which enables the main blockchain to split into a number of independent and hopefully smaller and faster chains. The transition from the current Ethereum 1 to the final version of Ethereum 2.0 (Serenity) is planned over a number of years and will be rolled out in a number of phases. The first phase, Phase 0, is known as the Beacon Chain. It is the backbone component of Ethereum 2.0 as it coordinates the whole network of *stakers* and shards.

^{*} This work was partially supported by the Ethereum Foundation, grant FY20-285, Q4-2020.

The Beacon Chain. The Beacon Chain (and its underlying protocol) is in charge of enforcing consensus, among the nodes, called *validators*, participating in the network, on the state of the system. The set of validators is dynamic: new validators can register by staking some ETH (Ethereum crypto-currency). Once registered, validators are eligible to participate and *propose* and *vote* for new blocks (of transactions) to be appended to the blockchain. The Beacon Chain shipped on December 1, 2020. At the time of writing (October 14, 2021), close to 250,000 validators have staked 7,780,000 ETH (\$30 Billion USD). Considering the coordination role and the amount of assets managed by the Beacon Chain, it is a mission-critical component of the Ethereum 2.0 ecosystem. The *Beacon Chain reference implementation* developed by the Ethereum Foundation is written in Python, and provides a detailed operational description of the state machine each Beacon Chain’s network participant (node) must implement.

Our Contribution. Our contribution is many-fold:

- We have formally specified and verified the absence of runtime errors in (a large and critical part of) the Beacon Chain reference implementation using the verification-friendly language Dafny.
- During the course of this work, we have uncovered several issues, proposed *verified* fixes, some of which have been integrated in the reference implementation, and others have resulted in substantial improvements (accuracy, readability) of the reference implementation.
- We have also manually synthesised *functional correctness specifications* that enable us to provide guarantees beyond runtime errors.
- Our software artefact with the code and proofs in Dafny is publicly available in our repository at <https://github.com/ConsenSys/eth2.0-dafny>.

Related Work. The Ethereum Foundation has supported several projects related to applying formal methods for the analysis of the Beacon Chain (and other components). A foundational project³ was undertaken in 2019 by Runtime Verification Inc. and provided a formal and *executable* semantics in the K framework, to the reference implementation [1]. The semantics was validated and the reference implementation could be tested which resulted in a first set of recommendations and fixes to the reference implementation. Although it may be possible to formally verify the Beacon Chain with the K-framework tools, to the best of our knowledge it has not been done yet. Runtime Verification Inc. have also formally specified and verified (in Coq [11]) the underlying GasperFFG [2] protocol. Our work complements these formal verification projects. Indeed, our objective is to provide guarantees for the *absence of bugs* (runtime errors), and *loop termination* which goes beyond testing. We have chosen to use a verification-friendly programming language, Dafny [10], as it enables us to write the code in a more developer-friendly manner (compared to K). However, we have used the code bases from the previous projects to guide us (e.g., semantics) during the course of this project.

³ <https://github.com/runtimeverification/beacon-chain-spec>

2 The Beacon Chain Reference Implementation

In this section we introduce the system we want to formally verify, what are the potential benefits and impacts of such of study, and we set out the goals of our experiment.

2.1 System Description and Scope of the Study

As a robust decentralised system, the Beacon Chain aims to implement a *replicated state machine* [9] that is fault-tolerant to a fraction of unreliable participants (e.g., participants that can crash). The replicated state machine is implemented with a number of networked identical state machines running concurrently. This provides redundancy and a more reliable system. The state of each machine changes on an occurrence of an *event*. As the machines operate asynchronously, two different machines may receive different events that cannot be totally ordered time-wise. This is why before processing an event and changing their states, the state machines run a *consensus protocol* to decide which event they should all process next. The consensus protocol aims to guarantee (under certain conditions) that an agreement will be reached which ensures that events are processed in the same order on each machine.

2.2 The Beacon Chain Reference Implementation

The Beacon Chain (Phase 0) reference implementation [6] describes the *state machine* that every Beacon node (participant) has to implement. The idea is that anyone is allowed to be a participant in the decentralised Ethereum 2.0 ecosystem when it is fully deployed. However, as the consensus protocol is Proof-of-Stake there must be a mechanism for participants to register and stake, to slash a participant's stake if they are caught⁴ misbehaving, i.e., not following the consensus protocol, and to reward them if they are honest. The Beacon Chain provides these mechanisms. It maintains records about the participants, called *validators*, ensuring fairness (each honest participant should have a voting power, for new blocks, related to its stake), and safety (a dishonest participant may be slashed and lose part of their stakes).

The full Beacon Chain (Phase 0) reference implementation [6] comprises three main sections:

1. the *Beacon Chain State Transition* describing the Beacon state machine which is the most complex component;
2. The *Simple Serialize (SSZ) library* for how to encode/decode (serialise/deserialise) data that have to be communicated over the network;
3. the *Merkleise library* for how to build efficient encoding of data structures into Merkle trees, and how to use them to verify Merkle proofs.

⁴ In a distributed system with potentially dishonest participants, it is not always possible to detect who is dishonest (byzantine). However, sometimes a participant can sometimes be proved to be dishonest.

The State Transition. The *Beacon Chain state transition* part is the most critical part and at the operational level the complexity stems from:

- time is logically divided into *epochs*, and each epoch into a fixed number of *slots*; the state is updated at each slot;
- at the beginning of each epoch, disjoint subsets of validators are assigned to each slot to participate in the block proposal for the slot and attest (vote) for *links* in the chain;
- the state updates that apply at an epoch boundary are more complex than the other updates;
- the actual state of the chain is a *block-tree* i.e., a tree of blocks, and the canonical chain is defined as a particular *branch in this tree*. How this branch is determined is defined by the *fork choice rule*.
- the *fork choice* rule relies on properties of nodes, *justification* and *finalisation*, in the block-tree. The state update describes how nodes in the block-tree are deemed justified/finalised. The rules for justification and finalisation are introduced in a separate document, the GasperFFG [2] protocol.

SSZ and Merkleise. These libraries are self-contained and independent from the state transition. We used them as a feasibility study and we had verified them before this project started. We have provided a complete Dafny reference implementation for them in the `merkle` and `ssz` packages [3].

2.3 Motivation for Formal Verification

As mentioned previously, the Beacon Chain shipped on December 1, 2020 and up to date, 250,000 validators have staked 7,780,000 ETH (\$30 Billion USD). It is clear that any bug, or logical error, could have disastrous consequences resulting in losses of assets for regular users, or downtimes and degradation of service, or losses of rewards for the validators.

There are regular opportunities (forks) to update the code of Beacon Chain nodes, so continuously running projects like ours is very valuable as what is important is to find and fix bugs before attackers can exploit them. The operational description of the Beacon Chain in the reference implementation is provided in Python. It was written by several reference implementation writers at the Ethereum Foundation and due to its size it is hard for one person to have a complete picture of it. It is the reference for any Beacon Chain client implementer. As a result, inaccuracies, ambiguities, or bugs in the reference implementation will lead to erroneous and/or buggy clients that can compromise the integrity, or the performance of the network. Moreover the reference implementation uses a defensive mechanism against unexpected errors:

(Rule 1) “State transitions that trigger an unhandled exception (e.g. a failed assert or an out-of-range list access) are considered invalid. State transitions that cause a `uint64` overflow or underflow are also considered invalid.” [6]

However this creates a risk that errors unrelated to the logic of the state transition function may introduce spurious exceptions. At the time of writing, there are at least 4 different Ethereum 2.0 client softwares that are used by validators. Bugs in the reference implementation may be handled differently in the various clients, and in some cases lead to a split in the network⁵. The correctness of the consensus mechanism is guaranteed for up to 1/3 of malicious nodes, that is, nodes deviating from the reference implementation, be it intentionally or unintentionally (e.g., because of a bug in the code). Hence, we should try to make sure we reduce (buggy) unintentionally malicious nodes.

2.4 Objectives of the Study

Our goal is to improve the overall safety, readability and usability of the reference implementation. Testing is of course an option, and Beacon Chain clients all implement some form of testing. In this project we are interested in proving the absence of bugs which goes beyond what testing techniques can do: testing can show the presence of bugs but not their absence (Dijkstra, 1970).

The primary aspect of our project was to make sure that the code was free of runtime errors (e.g., over/underflows, array-out-of-bounds, division-by-zero, ...). This provides more confidence that when an exception occurs and a state is left unchanged as per (Rule 1), the root cause is a genuine problem related to the state transition having been given an ill-formed block: if `state_transition(state, signed_block)` triggers an exception, it should imply that there is a problem with the `signed_block` not that some intermediate computations resulted in runtime errors. A secondary goal was to try and synthesise *functional specifications* from the reference implementation. This can help developers to design tests, and contributes to the specifications being language-agnostic. For instance, it can help write a client in a functional language which results in a more inclusive ecosystem.

3 Formal Specification and Verification

In this section we present the challenges of the project, motivate our methodology and conclude with our results' breakdown.

3.1 Challenges

The main challenges in this formal verification project are in the verification of the code of the `state_transition` component of the Beacon Chain. The SSZ and Merkleise libraries are much smaller, simpler, and independent components that can be dealt with separately.

The reference implementation for the Beacon Chain [6] introduces data types and algorithms that should be *interpreted* as Python 3 code. As a result it may

⁵ A network split can be caused if some clients reject a chain that is being followed by the other clients, which leads to a hard fork-like situation.

not be straightforward for those who are not familiar with Python to understand the meaning of some parts of the code. More importantly, the reference implementation is not executable and may contain type mismatches, incompatible function signatures, and bugs that can result in runtime errors like underflows or array-out-of-bounds.

Listing A.1. The state transition function.

```

1  def state_transition(
2      state: BeaconState,
3      signed_block: SignedBeaconBlock,
4      validate_result: bool=True
5  ) -> None:
6      block = signed_block.message
7      # Process slots (including those with no blocks) since block
8      process_slots(state, block.slot)
9      # Verify signature
10     if validate_result:
11         assert verify_block_signature(state, signed_block)
12     # Process block
13     process_block(state, block)
14     # Verify state root
15     if validate_result:
16         assert block.state_root == hash_tree_root(state)

```

A typical function in the reference implementation is written as a sequence of control blocks (including function calls) intertwined with *checks* in the form of **assert** statements. The `state_transition` function (Listing A.1) is the component that computes the update of the Beacon Chain’s state. The state (of type `BeaconState`) records some information including the validators’ stakes, the subsets of validators (*committees*) allocated to a given slot, and the hashes⁶ of the blocks that have already been added to the chain. A state update is triggered when a (signed) *block* is added to Beacon Chain. The state machine implicitly defined by the reference implementation generates sequences of states of the form:

$$s_0 \xrightarrow{b_0} s_1 \xrightarrow{b_1} s_2 \dots \xrightarrow{b_n} s_{n+1} \dots \quad (\text{StateT})$$

where s_0 is given (initial values), b_0 is the *genesis block* and for each $i \geq 1$, $s_{i+1} = \text{state_transition}(s_i, b_i)$.

There are several challenges in testing or verifying this kind of code:

- the functions calls (lines 8, 13) *mutate* the input variable `state`; those functions also call other functions that mutate the state.
- the semantics is not fully captured by the Python 3 interpretation because of the defensive mechanism [S1] (Section 2.3, page 4).
- a *valid* state transition is the opposite of an *invalid* state transition (characterised by [S1]). Determining when a computation is not going to trigger runtime errors or failed **asserts** is non-trivial. This is due to the use of mutating functions that can contain **assert** statements on values that are the results of intermediate computations.

⁶ The actual blocks are recorded in the `Store` which is a separate data structure.

- overall the code in the reference implementation does not explicitly define what *properties* `signed_block` should satisfy to guarantee that executing the function `state_transition(state, signed_block)` is not going to trigger an exception. The implicit semantics of the code is: if an exception occurs in executing `state_transition` with input `signed_block`, then this block must be invalid (assuming `state` is always valid).
It follows that, if the code contains a bug that triggers a runtime error unrelated to `signed_block` (e.g., an intermediate computation that overflows, or an array-out-of-bounds in a sorting algorithm), `signed_block` is declared invalid and not added to the chain. To alleviate this problem, we have collected the conditions (predicates) under which the addition of a block should not fail, which clearly defines when a block is valid.
- as there is no reference *functional specification* it is not immediate to understand when a block is invalid, and to write (unit) tests.
- finally the correctness of parts of the code rely on hidden assumptions, e.g., the total amount of ETH is X so no overflow should happen.

The challenges pertaining to the SSZ and Merkleise libraries are more manageable. First, the reference implementation is shorter. Second, even if there is no functional specification available, it is reasonably easy to synthesise them. Due to the previous weaknesses, the reference implementation [6] has been the subject of several informal explainers [15,5,6].

3.2 Methodologies

Resource Constraints. Resource-wise, the timeframe for our project was approximately 8 months (October 2020 to June 2021), with a team of two formal verification researchers (first two co-authors) and one Beacon Chain expert researcher (third co-author).

Verification Technique. The reference implementation is **not** the operational description of a distributed system, but rather a sequential state machine, as per (`StateT`), Section 3.1. Thus, techniques and tools that are adequate for the goals we set are related to *program formal verification*.

There are several techniques to approach program verification, ranging from fully automated (e.g., static analysis/abstract interpretation [4], software model-checking [8]) to interactive theorem proving [13]. Most static analysers are unsound (they cannot prove the absence of bugs) which disqualifies them for our project. It is anticipated that fully automated verification techniques can be effective to detect runtime errors but may have limited applicability to proving functional correctness.

On the other side of the spectrum, interactive theorem provers offer a complete arsenal of logics/rules that can certainly be used for this kind of projects. However they usually require encoding the software to be verified in a high-level mathematical language that is rather different to a language like Python. The level of expertise/experience required to properly use these tools is also high. Overall this seemed incompatible with our available resources.

A middle-ground between fully automated and interactive techniques is deductive verification available in verification-friendly programming languages like Dafny [10], Why3 [7], Viper [12] or Whiley [14]. Deductive verification lets verification engineers *propose* proofs and check them fully automatically.

We opted for Dafny [10], an award-winning verification-friendly language. Dafny is actively maintained⁷ and under continuous improvement. It offers imperative/object oriented and functional programming styles. Moreover, some of us had a previous exposure to Dafny (working on the SSZ/Merkleise libraries early in 2020), and we could be fully operational quickly, and it was compatible with our resources. We are convinced that similar results could be achieved with Why3, Viper or Whiley but did not have the resources to launch concurrent experiments.

Verification Strategy. Our strategy to write the Beacon Chain reference implementation in Dafny and detect/fix runtime errors, and prove some functional properties is three-fold:

1. **Identify simplifications.** The reference implementation is complex and trying to encode it fully in Dafny may result in inessential details hindering our verification progress. One example is the different data types (classes) for `Attestations`. There are several variations of the type `Attestations` and functions to convert between them. For our verification purposes, using `PendingAttestations` instead of the fully fledged `Attestations` was adequate. Another example is the abstraction of *hashing* functions. We assumed an *uninterpreted* collision-free hash function as we did not aim to prove any probabilistic properties involving this function.
2. **Translate the reference implementation in Dafny.** This helped the formal verification researchers to familiarise themselves with the reference implementation. During this phase, we focussed on adding *pre* and *post* conditions to the functions of the reference implementation to guarantee the absence of runtime errors. We were also able to prove some interesting invariants: the data structure that contains the block-tree is indeed a *well-formed tree*. This structure is implemented with links from nodes to their parent (where `null` is a possible parent in the code). The invariant states that the block-tree that is built with the `state_transition` function satisfies: *i*) the set of ancestors of any block contain blocks with strictly smaller *slot* number and is finite (no cycles) *ii*) the set of ancestors of any block in the block-tree always contains the genesis block (with slot 0).
3. **Synthesise functional specifications.** In the last phase, we manually synthesised functional specifications for each function in the reference implementation. We proved that each function in the reference implementation satisfied its functional specification. This enabled us to prove more complex properties as we could do the formal reasoning and proofs on the functional specifications and the results would carry over to the reference implementation. This was an effective solution to be able to prove properties of the

⁷ <https://github.com/dafny-lang/dafny>

reference implementation with lots of *mutations* (side-effects) without having to embed them deep in the proofs.

3.3 Results

The complete code base is freely available in [3]. There are several resources apart from the verified code: a Docker container to batch verify the code, and some notes/videos to help navigate the Dafny specifications.

Coverage. We estimated that we have verified 85% of the reference implementation. The remaining 15% are simplifications e.g., data types, or using a fixed set of validators instead of a dynamic set. Adding the remaining details to the released version would require a substantial amount of work and at the same time it seems that the likelihood of finding new issues is low. Since the Beacon Chain has shipped in December 1, 2020, only a few minor issues have been uncovered and promptly fixed which seems to confirm the previous claim.

Absence of Runtime Errors. All of the functions we have implemented in Dafny are annotated with pre (**requires**) and post (**ensures**) conditions that are verified, including *loop termination*. The Dafny version of function **state_transition** is given in Listing A.2. Other functions are written similarly e.g., **process_slots** and **process_block**. The Dafny verifier enforces the absence of runtime errors like division by zero, under/overflows, array-out-of-bounds. It follows that our code base is provably free of this kind of defect. Moreover, additional checks can be added like the **assert** statement at line 28. We have added all the **assert** statements from the reference implementation and proved that they could not be violated. This requires adding suitable pre-conditions.

Regarding loop termination proofs, most of the proofs are based on relatively simple ranking functions. An example of a non-trivial proof termination can be found in a functional correctness proof: *the ancestors of a given block form a strictly decreasing sequence, slot-wise, and consequently end up in the genesis block*. The corresponding code is in the [Forkchoice.dfy](#) file.

Functional Correctness. Beyond the absence of runtime errors, we have synthesised *functional specifications* based off the reference implementation code. For instance we have decomposed the state update in **state_transition** into a sequence of simpler steps, **updateBlock**, **forwardStateToSlot**, **nextSlot** and proved that the result is a composition of these functions. This provides more confidence that the code is functionally correct as our decomposition specifies smaller changes in the state. It also enables us to prove properties on the functional specifications and transfer them to the imperative version of the code.

Impact of our Project. During the course of this projects we have reported several issues, some of them bugs (3), some of them need for clarifications (5) in the reference implementation. The issues we have uncovered are tracked in the *issues tracker* of our github repository. Some of the bugs we reported have been fixed and our *clarifications* category has led to several improvements in

the writing of the reference implementation. Moreover, we have provided a fully documented version of the reference implementation in Dafny. The Dafny code contains clear pre and post conditions that can help developers understand the effect of a function and can be used to write unit tests.

Listing A.2. Dafny version of `state_transition`

```

1  method state_transition(s:BeaconState,b:BeaconBlock)
2      returns (s': BeaconState)
3      // A valid state to start from
4      requires |s.validators| == |s.balances|
5      requires is_valid_state_epoch_attestations(s)
6      // b must a block compatible with s
7      requires isValidBlock(s, b)
8      // Functional correctness
9      ensures s' ==
10         updateBlock(forwardStateToSlot(nextSlot(s),b.slot),b)
11      // Other post-conditions
12      ...
13      ensures s'.slot == b.slot
14      ensures s'.latest_block_header.parent_root ==
15         hash_tree_root(
16             forwardStateToSlot(nextSlot(s), b.slot)
17             .latest_block_header
18         )
19      ensures |s'.validators| == |s'.balances|
20      ...
21  {
22      // Finalise slots before b.slot.
23      s' := process_slots(s, b.slot);
24
25      // Process block and compute the new state.
26      s' := process_block(s', b);
27
28      // Verify state root (from eth2.0 specs)
29      assert (b.state_root == hash_tree_root(s'));
30  }
```

Statistics. Table 1, page 11, provides some insights into the actual code, per file. We have tried to keep the size of each file small and provide optimal modularity in the proofs. The files in the packages fall into one of the three categories: `file.dfy` is the Python-reference implementation translated into Dafny; `file.s.dfy` contains the functional specifications we have synthesised and `file.p.dfy` any additional proofs (Lemmas) that are used in the correctness proofs. It is hard to estimate the *lines of code to lines of proofs ratio* for many reasons: *i*) it is not always possible to locate all the proofs in a separate unit (e.g. a module in Dafny), as this can create circular dependencies.

It follows that counting lines of proofs as lines in the Lemmas is not an accurate measure; *ii*) in some of the proofs, we have, on purpose, provided redundant hints. As a result some proofs can be shortened but this may be at the expense of readability (and verification time). For this project, a conservative (and empirical) *lines of code to lines of proofs ratio* seems to be around 1 to 7.

Table 1. Statistics. A file providing functional specifications. A file providing proofs (lemmas in Dafny). **#LoC** (resp. **#DoC**) is the number of lines of code (resp. documentation), **Lem.** the number of proper lemmas, **Imp.** the number of proved imperative functions with pre/post conditions.

Files	Package	#LoC	Lem.	Imp.	#Doc	$\frac{\#Doc}{\#LoC}$ (%)	Proved
ActiveValidatorBounds.p.dfy	beacon	52	3	0	29	56	3
BeaconChainTypes.dfy	beacon	54	0	0	171	317	0
Helpers.dfy	beacon	1003	9	89	670	67	98
Helpers.p.dfy	beacon	136	13	0	114	84	13
Helpers.s.dfy	beacon	136	9	6	67	49	15
AttestationsTypes.dfy	beacon/attestations	30	0	0	68	227	0
ForkChoice.dfy	beacon/forkchoice	229	3	15	172	75	18
ForkChoiceTypes.dfy	beacon/forkchoice	9	0	0	17	189	0
Crypto.dfy	beacon/helpers	7	0	1	3	43	1
EpochProcessing.dfy	beacon/statetransition	384	0	14	127	33	14
EpochProcessing.s.dfy	beacon/statetransition	398	24	0	336	84	24
ProcessOperations.dfy	beacon/statetransition	361	0	11	119	33	11
ProcessOperations.p.dfy	beacon/statetransition	160	10	0	74	46	10
ProcessOperations.s.dfy	beacon/statetransition	410	12	6	137	33	18
StateTransition.dfy	beacon/statetransition	215	0	8	126	59	8
StateTransition.s.dfy	beacon/statetransition	213	11	1	100	47	12
Validators.dfy	beacon/validators	11	0	0	53	482	0
Merkleise.dfy	merkle	504	9	18	135	27	27
BitListSeDes.dfy	ssz	262	7	3	64	24	10
BitVectorSeDes.dfy	ssz	155	4	3	53	34	7
BoolSeDes.dfy	ssz	22	0	2	3	14	2
BytesAndBits.dfy	ssz	90	7	6	44	49	13
Constants.dfy	ssz	104	0	0	36	35	0
IntSeDes.dfy	ssz	130	2	2	20	15	4
Serialise.dfy	ssz	514	3	5	36	7	8
DafTests.dfy	utils	62	0	4	25	40	4
Eth2Types.dfy	utils	227	1	3	77	34	4
Helpers.dfy	utils	220	11	3	103	47	14
MathHelpers.dfy	utils	293	18	6	105	36	24
NativeTypes.dfy	utils	28	0	0	13	46	0
NonNativeTypes.dfy	utils	8	0	0	6	75	0
SeqHelpers.dfy	utils	69	8	2	58	84	10
SetHelpers.dfy	utils	74	6	0	50	68	6
TOTAL		6570	170	208	3212	49	378

4 Findings and Lessons Learned

During the course of our formal verification effort we found subtle bugs and also proposed some clarifications for the reference implementations. In addition, our work was the opportunity to start some discussions about how to improve the readability of the reference implementation, e.g., by using pre and post conditions rather than `assert` statements. In this section we provide more insights into some of the main issues we reported⁸, and also on the practicality of this kind of project.

4.1 Array-out-of-bounds Runtime Error

The function `get_attesting_indices` (Listing A.3) is called from within several important components of the `state_transition` function including the processing of rewards and penalties, justification and finalisation, as well as the processing of attestations (votes).

Listing A.3. Python code for `get_attesting_indices`.

```

1  def get_attesting_indices(
2      state: BeaconState,
3      data: AttestationData,
4      bits: Bitlist[MAX1]
5  ) -> Set[ValidatorIndex]:
6      """
7      Return the set of attesting indices corresponding to
8      'data' and 'bits'.
9      """
10     committee=get_beacon_committee(state, data.slot, data.index)
11     return
12     # Collect indices in committee for which bits is set
13     set(index for i, index in enumerate(committee) if bits[i])

```

The last line (13) of `get_attesting_indices` collects the indices in the array `committee` that have a corresponding bit set to true in array `bits` and returns it as a *set* of indices. The length of `bits`, noted $|\text{bits}|$, is `MAX1`. Consequently, the following relation must be satisfied to avoid an array-out-of-bounds error: $|\text{committee}| \leq \text{MAX1}$. It follows that to prove⁹ the absence of array-out-of-bounds error in Dafny, the specification of `get_attesting_indices` (in Dafny) requires a pre-condition, $|\text{get_beacon_committee}(\dots)| \leq \text{MAX1}$ (line 10). This pre-condition naturally imposes a post-condition for `get_beacon_committee` and trying to prove this post-condition we uncovered a very subtle bug: depending on the number of *active validators* V in `state`:

$V \leq 4,194,304$: there is no array-out-of-bounds error as we can prove that $|\text{get_beacon_committee}(\dots)| \leq \text{MAX1}$ for *all* values of the input parameters `data.slot` and `data.index`,

⁸ <https://github.com/ConsenSys/eth2.0-dafny/issues>

⁹ In Dafny, this check is built-in so you cannot avoid this proof.

- 4, 194, 304 < V < 4, 196, 352:** there is at least one value of the input parameters `data.slot` and `data.index` for which `|get_beacon_committee(...)| > MAX1`, which results in an array-out-of-bounds, and
- 4, 196, 352 ≤ V:** for all input combination of `data.slot` and `data.index`, there is an array-out-of-bounds `|get_beacon_committee(...)| > MAX1`.

This previously undocumented bug was difficult to detect. It required many hours of effort to model the dynamics of the problem; the analysis was quite complex due to the multiple interrelated parameter calculations, as well as the use of floored integer division. The full description and the analysis of this bug has been reported as issue¹⁰ to the reference implementation github repository. The issue was confirmed by the reference implementation writers.

4.2 Beyond Runtime Errors

We have also been able to establish some well-formedness properties of the data structure that represents the *block-tree* built by each node. Each added block has a stamp, the *slot number* and a link to its *parent*. The block-tree is the tree representation of the parent relation. The block-tree should satisfy the following properties:

- Every block *b* except the genesis block has a parent,
- Every block *b* with parent *p* is such that the slot of *b* is strictly larger than the slot of *p*,
- the transitive closure of the parent relation produces chains of blocks that are totally ordered using the < relation on slot,
- the smallest element of each chain has slot 0 (and consequently is the genesis block).

We have established these properties in `ForkChoice.dfy` using a list of invariants on the `Store`.

Another noticeable contribution compared to other approaches (like testing) is that we have proved the termination of all loops. For the majority of the loops, the *ranking function* used to prove termination is rather straightforward. An example of a more complicated (decreasing) ranking function can be found in the proof of a (functional correctness) lemma in `ForkChoice.dfy`: the proof relies on the slot number of a block's parent being strictly smaller than the slot number of a block itself. The lemma establishes that the graph defined by the parent relation on the blocks in the store, is always well-formed and is a (block-)tree: the list of ancestors of any block in the store is ordered (slot-wise) and the smallest element is the genesis block.

4.3 Finalisation and Justification

During the course of the project we benefited from the guidance of the third co-author who has comprehensive expertise in various aspects of the Beacon Chain,

¹⁰ <https://github.com/ethereum/consensus-specs/issues/2500>

including the *fork choice* part, and identified the *fork choice* implementation of the reference implementation as a component that needed verification.

The *fork choice rules* are designed to identify a *canonical branch* in the block-tree which in turn defines the *canonical chain*. To achieve this goal, we first assumed a fixed set of validators. Then we built a Dafny proof of the GasperFFG [2] protocol and tried to prove properties about the *justified and finalised blocks* in the block-tree. We could mechanically prove Lemmas 4.11 and 5.1, Theorem 5.2 from [2]. Note that a complete proof in Coq is available in [11] but it does not use the Beacon Chain data structures. We only managed to push these properties up to a certain level on the functional specifications of our code base and not on the actual reference implementation. Doing so would require us to add a substantial amount of details and to modify the structure of several proofs which was not doable in our timeframe. This experimental work is archived in branch `goal1` of the repository. There is a currently ongoing work focussing on this topic: designing the mechanised proofs¹¹ of the refinement soundness of the state transition function (Phase 0) w.r.t. the GasperFFG protocol.

4.4 Reflection

Verification Effort. The effort for formal verification took 16 person-months. This figure is for the *Beacon Chain State Transition* and does not include the time spent on the SSZ and Merkleize libraries that were completed before this project started. The division of time was primarily between the second and third components of the project. Translation of the reference implementation in Dafny, took approximately 6 person-months¹². Synthesis of functional specifications (manually), including proofs, took approximately 10 person-months. The time allocation for the identification of simplifications is more difficult to assess. Though some consideration was given initially, this aspect was ongoing, as our understanding of the reference implementation evolved.

Trust Base. The validity of the verification results assumes the correctness of the Dafny specification and the Z3 verifier. Dafny is actively maintained and under continuous improvement. And in the rare instance where Dafny behaves unpredictably, bug reports are responded to in a timely manner. During the course of this project a few bugs were reported. For example it was found that the definition of an inconsistent `const` could lead to unsound verification results and reported as an issue¹³ (fixed) to the Dafny language github repository.

Practicality of the Approach. The use of Dafny does not require any specific knowledge beyond standard program verification (Hoare style proofs) and first-order logics. There is ample support (videos, tutorials, books) to help learning how to write Dafny programs and proofs. The main difficulties/challenges in writing and verifying projects of this size with Dafny (and the same holds for

¹¹ <https://github.com/runtimeverification/beacon-chain-verification>

¹² This translation includes the proof of absence of runtime errors.

¹³ <https://github.com/dafny-lang/dafny/issues/922>

other verification-friendly automated deductive verifiers) are: **1.** when the verification fails, it requires some experience to interpret the verifier feedback and make some progress, and **2.** the unpredictability (time-wise) of the reasoning engine; this is due to the fact that *verification conditions* that are generated by Dafny are in semi-decidable theories of the underlying SMT-solver (Z3). In our experience, adding a seemingly innocuous line of proof may result in either a surge or a drastic reduction of verification time.

5 Conclusion

Overall this project was a significant undertaking. The complexity of the state transition mechanism, combined with the ambitious project scope, makes this one of the largest formal verification projects to be completed using Dafny. Even with the model simplifications, the Python language is not particularly compatible with the fundamentals that underpin formal verification, which presented continual challenges. Upon reflection: *i*) the project would have benefited from a larger team and *ii*) consideration of the application of formal verification methods earlier, ideally within the design process, would have had a positive impact.

The interest generated from this project provided an opportunity to facilitate Dafny training for the reference implementation writers at the Ethereum Foundation. This training included the translation of code into Dafny, as well as the more advanced topic of proof construction. Participants were able to gain insight into the formal verification process which could provide valuable context when drafting future reference implementations and specifications.

Acknowledgements. We thank the anonymous referees for their constructive feedback which helped improve the initial version of the paper. We thank the reference implementation writers at the Ethereum Foundation for their insightful feedback. This project was supported by an Ethereum Foundation Grant, FY20-285 and we thank Danny Ryan (Ethereum Foundation) and Ben Edgington (ConsenSys) for their help in setting up this project and their support and encouragements. We also thank Roberto Saltini (ConsenSys) for his contribution at the early stage of the project.

References

1. Alturki, M., Bogdanas, D., Hathhorn, C., Park, D., Roşu, G.: An executable K model of ethereum 2.0 beacon chain phase 0 specification. Project Report (2020), <https://github.com/runtimeverification/beacon-chain-spec>
2. Buterin, V., Hernandez, D., Kamphofner, T., Pham, K., Qiao, Z., Ryan, D., Sin, J., Wang, Y., Zhang, Y.X.: Combining GHOST and casper. CoRR **abs/2003.03052** (2020), <https://arxiv.org/abs/2003.03052>
3. ConsenSys: Formal verification of the ethereum 2.0 specifications in dafny. (2021), <https://github.com/ConsenSys/eth2.0-dafny>
4. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
5. Edgington, B.: (2020), <https://benjaminion.xyz/eth2-annotated-spec/>

6. Ethereum Foundation: Beacon chain specifications (2020), <https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/beacon-chain.md>
7. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science*, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8,
8. Jhala, R., Majumdar, R.: Software model checking. *ACM Comput. Surv.* **41**(4) (Oct 2009). <https://doi.org/10.1145/1592434.1592438>,
9. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978). <https://doi.org/10.1145/359545.359563>,
10. Leino, K.R.M.: Accessible software verification with Dafny. *IEEE Softw.* **34**(6), 94–97 (2017). <https://doi.org/10.1109/MS.2017.4121212>,
11. Li, E., Serbanuta, T., Diaconescu, D., Zamfir, V., Rosu, G.: Formalizing correct-by-construction casper in coq. In: *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020*. pp. 1–3. IEEE (2020). <https://doi.org/10.1109/ICBC48266.2020.9169468>,
12. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) *Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security*, vol. 50, pp. 104–125. IOS Press (2017). <https://doi.org/10.3233/978-1-61499-810-5-104>,
13. Nipkow, T., Klein, G.: *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated (2014)
14. Pearce, D.J., Utting, M., Groves, L.: An introduction to software verification with Why3. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) *Engineering Trustworthy Software Systems - 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018, Tutorial Lectures. Lecture Notes in Computer Science*, vol. 11430, pp. 1–37. Springer (2018). https://doi.org/10.1007/978-3-030-17601-3_1,
15. Ryan, D.: (2020), <https://notes.ethereum.org/@djrtwo/Bkn3zpwxB#Phase-0-for-Humans-v0100>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

