Lesson 14 - Formal verification and ETH upgrades

Formal Verification

See overview resource

Introduction

Formal Verification is the process by which one proves properties of a system mathematically. In order to do that, one writes a formal specification of the application behavior. The formal specification is analogous to a Statement of Intended Behavior, but it is written in a machine-readable language.
The formal specification is later proved or disproved using one of the available tools.

The correctness verification is about respecting the specifications that determine how users can interact with the smart contracts and how the smart contracts should behave when used correctly.

There are two approaches used to verify the correctness:

- the formal verification and
- the programming correctness.

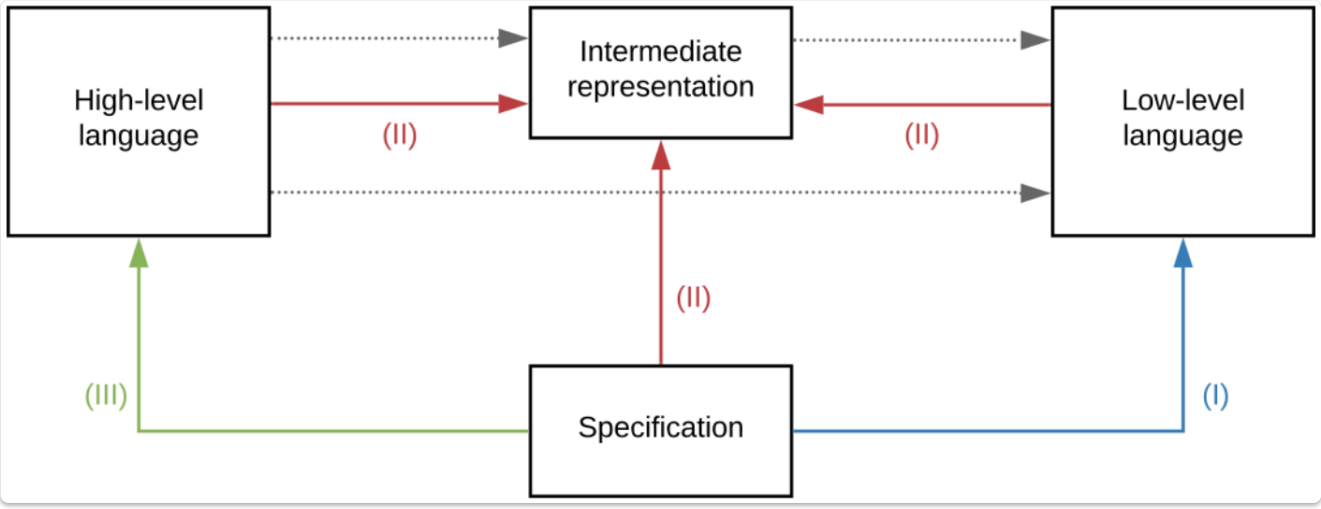
The formal verification methods are based on formal methods (mathematical methods), while the programming correctness methods are based on ensuring the programming as code is correct, which means the program runs without entering an infinite loop and gives correct outputs for correct inputs.

In the case of smart contracts verification, we can improve smart contracts security by ensuring the correctness of contracts using formal verification.

We may distinguish three major verification approaches :

1. a specification may be assessed directly at bytecode level. As contract sources are not necessarily available, this approach provides a way to assess some properties on already deployed contracts.
2. intermediate representations may be specifically designed as targets for verification tools such as proof assistants. This can offer a very suitable environment for code optimisation and dynamic verification according to specifications. The intermediate code representation may come both from compilation of a high-level contract or decompilation of low-level code.
3. some tools also reason directly on high-level languages. This approach offers a precious direct feedback to developers at verification time.

Different methods of verification based on the comparison object with the specification:



Contrary to Tezos and Cardano for instance, a formal semantics of the EVM was only described ex-post. And as the Solidity compiler changes rapidly, in the absence of formal semantics of the language that would allow correct-by-construction automatic generation of verification tools, the latter would need to follow the rate of change as well. These reasons make formal verification of smart contracts way harder on Ethereum than on other blockchains despite tremendous work initiated by several teams.

Two essential notions must be kept in mind to understand the challenge of formally verifying smart contracts :

1. **Verification can be thought of as testing a set of properties** (not all, only the ones we formalize) for *all* possible scenarios (including not only parameters but also message senders, blockchain state, storage, etc.). We don't prove smart contracts, we prove some of their properties by proving their correctness according to a specification.
2. **Proven correctness of all translations determines the level of confidence one can have in the entire framework.** If formal verification of a program is performed on its intermediate representation, a backwards translation will allow for meaningful messages to be displayed in the higher-level original language. Furthermore, if translation to machine bytecode is not secure, then no one can formally trust its execution, regardless of the verification effort at other levels. To be considered valid, a proof must be generated within a single, trusted logical framework, from the level of the specification language to the virtual machine execution level.

As an example of functional specifications, some properties of interest for an ERC20 implementation can be:

1. a *function level contract* stating that the function `transfer` decreases the balance of the sender in a determined `amount` and increases the destination balance in the same `amount` without affecting other balances. The sender must have enough tokens to perform this operation.
2. a *contract level invariant* prescribing that the sum of balances is always equal to the total supply.
3. a *temporal property* specifying a property that must hold for a sequence of transactions to be valid, e.g., `totalSupply` does not increase unless the `mint` function is invoked.

Pros and cons of formal verification

Formal verification of smart contracts isn't easy. There's a steep learning curve and a high entry threshold. A developer needs to undergo special training to be able to formalize requirements using formal verification tools.

Additionally, formal verification requires segregation of duties, in order to be effective. If the writer of the specifications is the coder of the application, the effectiveness of formal verification is greatly reduced.

Pros

- It doesn't rely on compilers, this allows formal verification to catch bugs that were introduced during compilation or other transformations of the source code.
- It's language-independent, you can easily verify smart contracts written in Solidity, Viper, or other languages that may appear in the future.
- A formal specification can work as documentation for the contract itself.

Cons

- It requires a specially prepared Ethereum execution environment.
- A formal specification of the contract itself is required. Creating a formal specification is a long and difficult process and demands a lot of preparation from your development team. Specifying a program involves abstracting its properties and is thus a difficult task.
- If there are any errors in the specification, they will be left undetected, every false requirement will be perceived as correct. Therefore, the verification won't detect a problem with the smart contract.

Formal verification vs Unit Testing

Unit testing is usually cheaper than any other type of audit, as it's performed when developing a smart contract. Proper unit tests should reflect the specification and cover the smart contract with the help of use cases and functionality the specification describes. However, unit tests are just as imperfect as informal specifications. Even if the smart contract code is fully covered with unit tests, the developer may miss some edge cases that could lead to bugs or even security vulnerabilities like overflows or unprotected functions.

Formal verification vs Code Audits

Formal verification is not a silver bullet, or a substitute for a good audit. Also in other industries where audits are conducted, they include not only the code base, but also the formal verification specification itself. Flaws in this specification will mean that some of the properties of the application are not actually proven in the end, with bugs possibly going unnoticed.

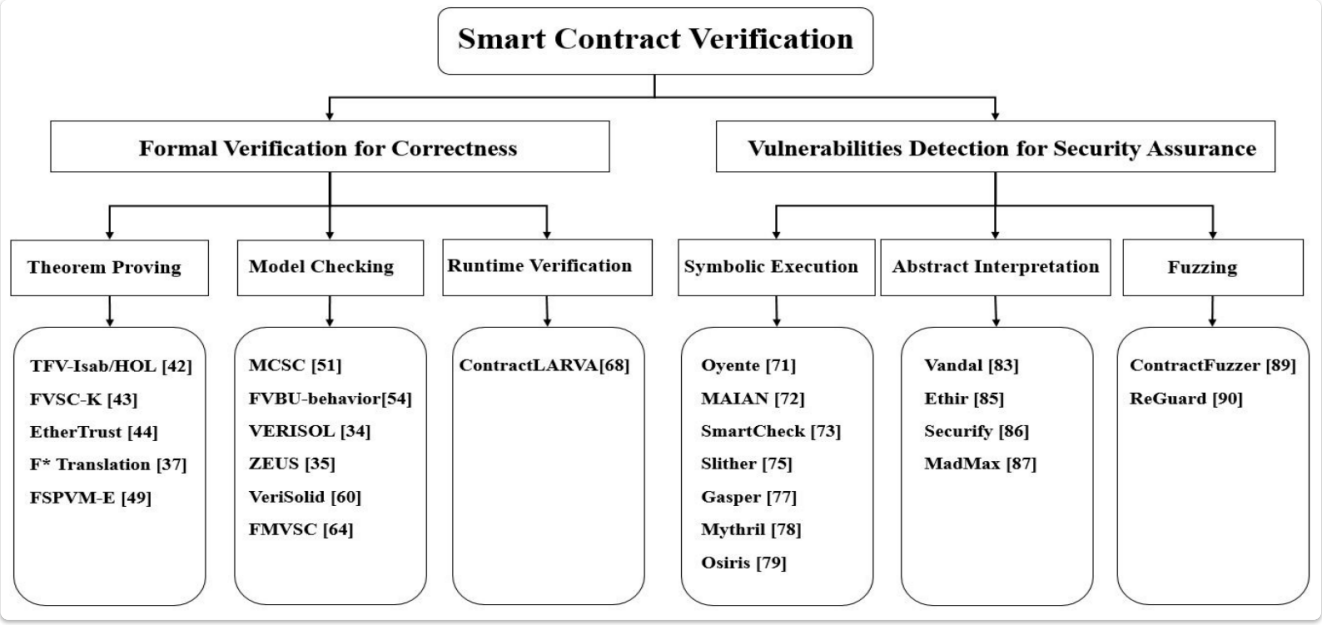
While no smart contract can be guaranteed as safe and free of bugs, a thorough code audit and formal verification process from a reputable security firm helps uncover critical, high severity bugs that otherwise could result in financial harm to users.

Formal verification vs Static Analysis Tools

Static analysis tools are used to achieve the same goal as formal verification. During static analysis, a dedicated program (the static analyzer) scans the smart contract or its bytecode in order to understand its behavior and check for unexpected cases such as overflows and reentrancy vulnerabilities.

However, static analyzers are limited in the range of vulnerabilities they can detect. In a sense, a static analyzer attempts to perform the same formal verification with a one-fits-all specification that simply states, a smart contract shouldn't have any known vulnerabilities. Obviously, a custom specification is much better, as it will detect each and every possible vulnerability within a given smart contract.

Tools



Taxonomy of the smart contract verification's Tools. Source: Verification of smart contracts: A survey

Manticore

Manticore is a dynamic symbolic execution tool, first described in the following blogposts by Trailofbits (1, 2).

Manticore performs the "heaviest weight" analysis. Like Echidna, Manticore verifies user-provided properties. It will need more time to run, but it can prove the validity of a property and will not report false alarms.

Dynamic symbolic execution (DSE) is a program analysis technique that explores a state space with a high degree of semantic awareness. This technique is based on the discovery of "program paths", represented as mathematical formulas called `path predicates`. Conceptually, this technique operates on path predicates in two steps:

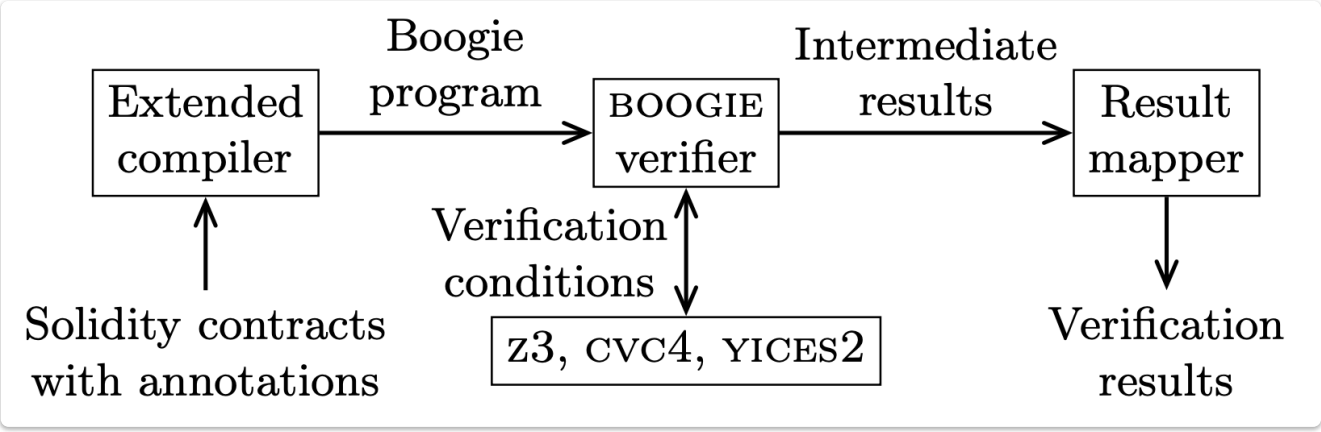
1. They are constructed using constraints on the program's input.
2. They are used to generate program inputs that will cause the associated paths to execute.

This approach produces no false positives in the sense that all identified program states can be triggered during concrete execution. For example, if the analysis finds an integer overflow, it is guaranteed to be reproducible.

---

solc-verify

Solc-verify is a source-level formal verification tool for Soldity smart contracts, developed in collaboration with SRI International.
Solc-verify takes smart contracts written in Solidity and discharges verification conditions using modular program analysis and SMT solvers.
Built on top of the Solidity compiler, solc-verify reasons at the level of the contract source code. This enables solc-verify to effectively reason about high-level functional properties while modeling low-level language semantics (e.g., the memory model) precisely.
The contract properties, such as contract invariants, loop invariants, function pre- and post-conditions and fine grained access control can be provided as in-code annotations by the developer. This enables automated, yet user-friendly formal verification for smart contracts.



Overview of the solc-verify modules. The extended compiler creates a Boogie program from the Solidity contract, which is checked by the boogie verifier using SMT solvers. Finally, results are mapped back and presented at the Solidity code level.

VeriSol

VeriSol (Verifier for Solidity) is a Microsoft Research project for prototyping a formal verification and analysis system for smart contracts developed in the popular Solidity programming language. It is based on translating programs in Solidity language to programs in Boogie intermediate verification language, and then leveraging and extending the verification toolchain for Boogie programs. The following blog provides a high-level overview of the initial goals or VeriSol.

This tool takes contracts written in Solidity and tries to prove that the contract satisfies a set of given properties or provides a sequence of transactions that violates the properties. VeriSol directly understands `assert` and `require` clauses directly from Solidity but also includes a notion called Code Contracts (coined from .NET Code Contracts), where the language of contracts does not extend the language but uses a (dummy) set of additional (dummy) libraries that can be compiled by the Solidity compiler.

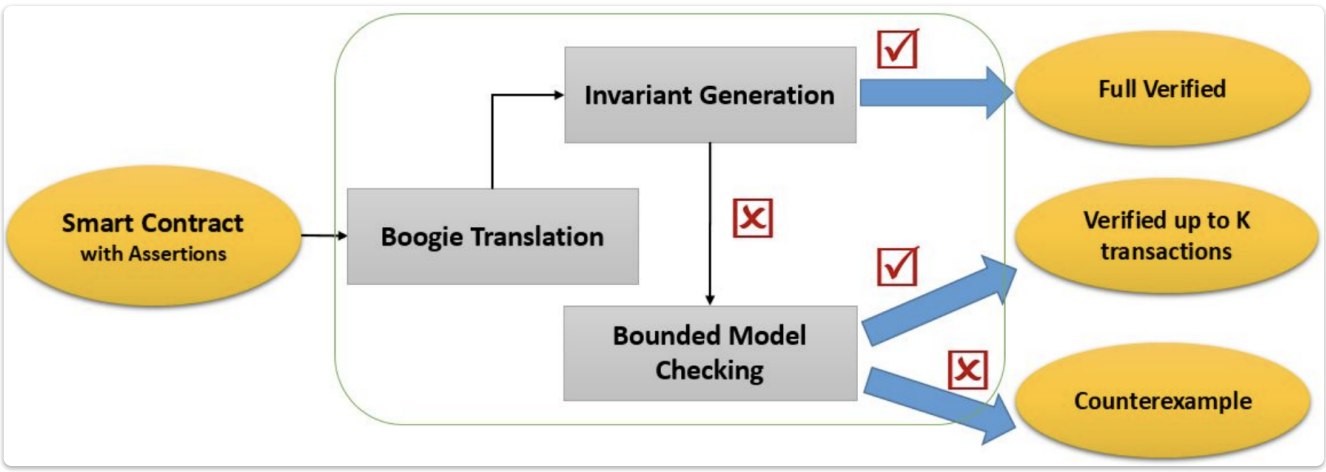
The function `transfer` equipped with assertions looks like this:

```
function transfer(address recipient, uint256 amount) public returns (bool) {
  _transfer(msg.sender, recipient, amount);
  assert (VeriSol.Old(_balances[msg.sender] + _balances[recipient]) ==
_balances[msg.sender] + _balances[recipient]);
  assert (msg.sender == recipient || ( _balances[msg.sender] ==
VeriSol.Old(_balances[msg.sender] - amount));

  return true;
}
```

The spec is straightforward. We expect the balance of the `recipient` to be increased by `amount`, and that amount of tokens should be decreased from the balance of `msg.sender`.

Schematic workflow of VERISOL:

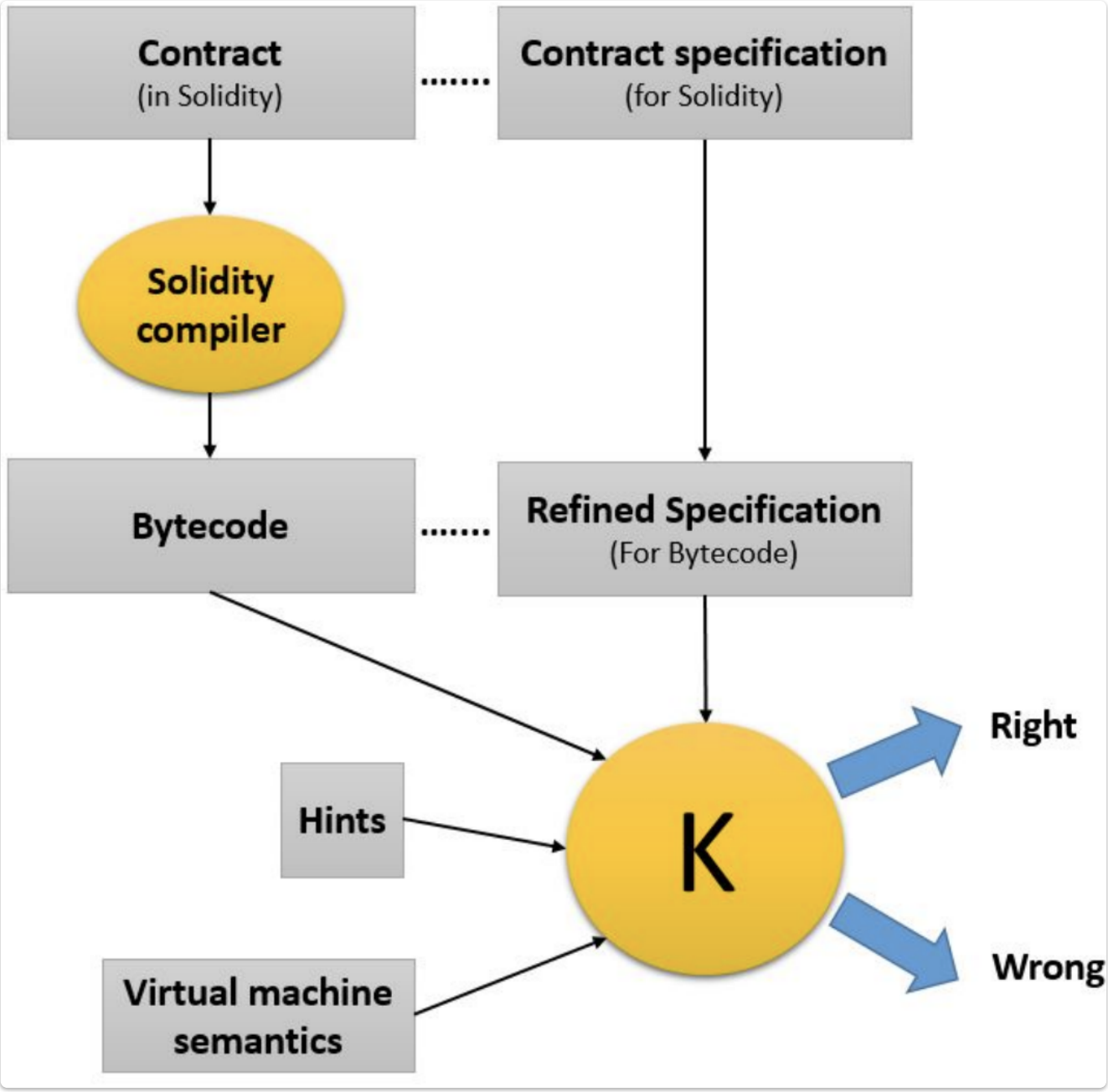


K Framework

The K Framework is one of the most robust and powerful language definition frameworks. It allows you to define your own programming language and provides you with a set of tools for that language, including both an executable model and a program verifier.

The K Framework provides a user-friendly, modular, and mathematically rigorous meta-language for defining programming languages, type systems, and analysis tools. K includes formal specifications for C, Java, JavaScript, PHP, Python, and Rust. Additionally, the K Framework enables verification of smart contracts.

The K-Framework is composed of 8 components listed in the following figure:



The KEVM provides the first machine-executable, mathematically formal, human readable and complete semantics for the EVM. The KEVM implements both the stack-based

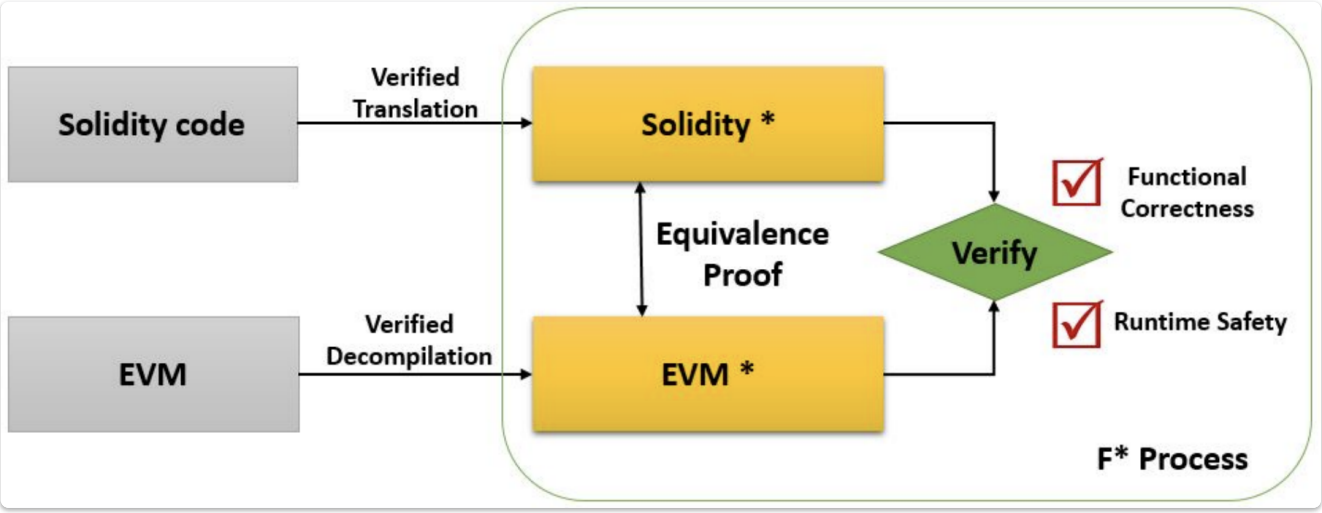
execution environment, with all of the EVM's opcodes, as well as the network's state, gas simulation, and even high-level aspects such as ABI call data.

If formal specifications of languages are defined, K Framework can handle automatic generation of various tools like interpreters and compilers. Nevertheless, this framework is very demanding (a lot of manual translations of specifications required, which are prone to error) and still suffer from some flaws (implementations of the EVM on the mainnet may not match the machine semantics for instance).

F* Translation

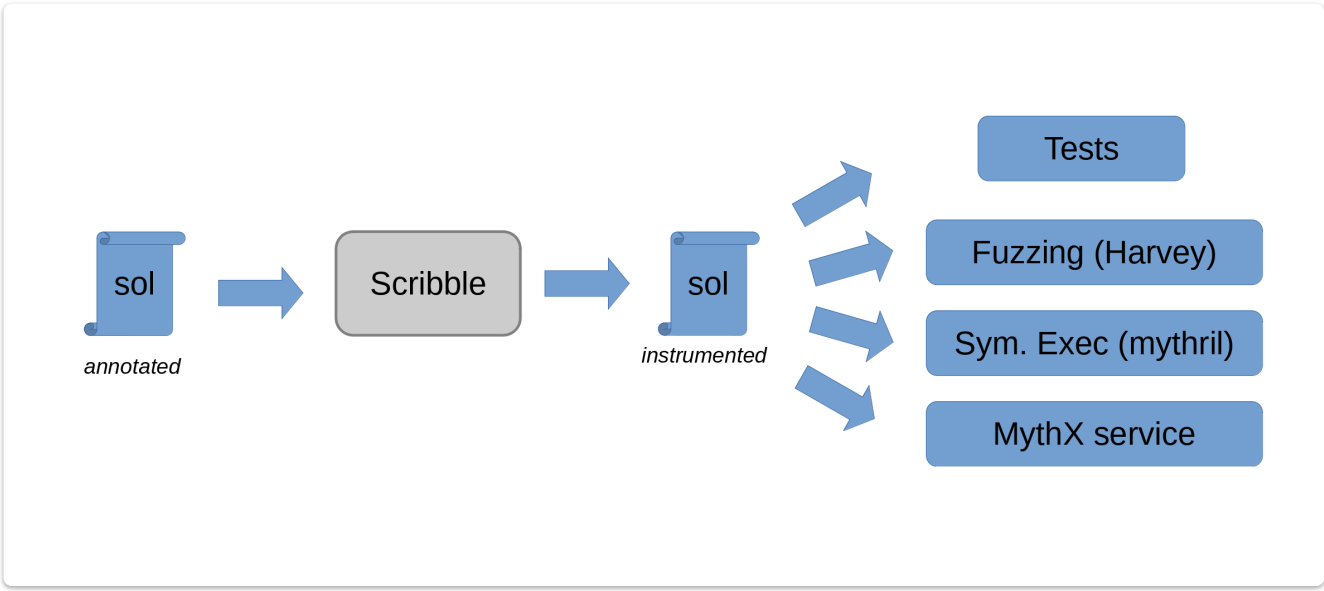
In cooperation between Microsoft Research and Harvard University, a framework is done to analyze and formally verify Ethereum smart contracts using F* functional programming language. Such contracts are generally written in Solidity and compiled down to the Ethereum Virtual Machine (EVM) byte-code. They develop a language-based approach for verifying smart contracts.

Two prototype tools based on F* are presented and a smart contract verification architecture is proposed and illustrated:



Consensys Scribble

Scribble is a runtime verification tool for Solidity that transforms annotations in the Scribble specification language into concrete assertions that check the specification.
In other words, Scribble transforms existing contracts into contracts with equivalent behaviour, except that they also check properties.
With these *instrumented* contracts, you can use testing, fuzzing or symbolic execution (for example using Mythril or the MythX service) to check if your properties can be violated.



See Documentation

Installation

It is available via npm :

```
npm install -g eth-scribble
```

You add invariants to the code in the following format

```
import "Base.sol";

contract Foo is Base {

        /// #if_succeeds {:msg "P1"} y == x + 1;

        function inc(uint x) public pure returns (uint y) {

        return x+1;

        }
```

```
}
```

Scribble will then create instrumented files for you that can be used for testing.

Solidity SMT Checker

See documentation
Also this useful blog

The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements.
That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true.

The other verification targets that the SMTChecker checks at compile time are:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.
- Out of bounds index access.
- Insufficient funds for a transfer.

To enable the SMTChecker, you must select which engine should run, where the default is no engine. Selecting the engine enables the SMTChecker on all files.

The SMTChecker module implements two different reasoning engines, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have different characteristics. The engines are independent and every property warning states from which engine it came. Note that all the examples above with counterexamples were reported by CHC, the more powerful engine.

By default both engines are used, where CHC runs first, and every property that was not proven is passed over to BMC. You can choose a specific engine via the CLI option `--model-checker-engine {all,bmc,chc,none}` or the JSON option `settings.modelChecker.engine={all,bmc,chc,none}`

From the command line you can use

```
solc overflow.sol \
    --model-checker-targets "underflow,overflow" \
    --model-checker-engine all
```

In Remix you can add

```
pragma experimental SMTChecker;
```

to your contract, though this is deprecated.

Resources

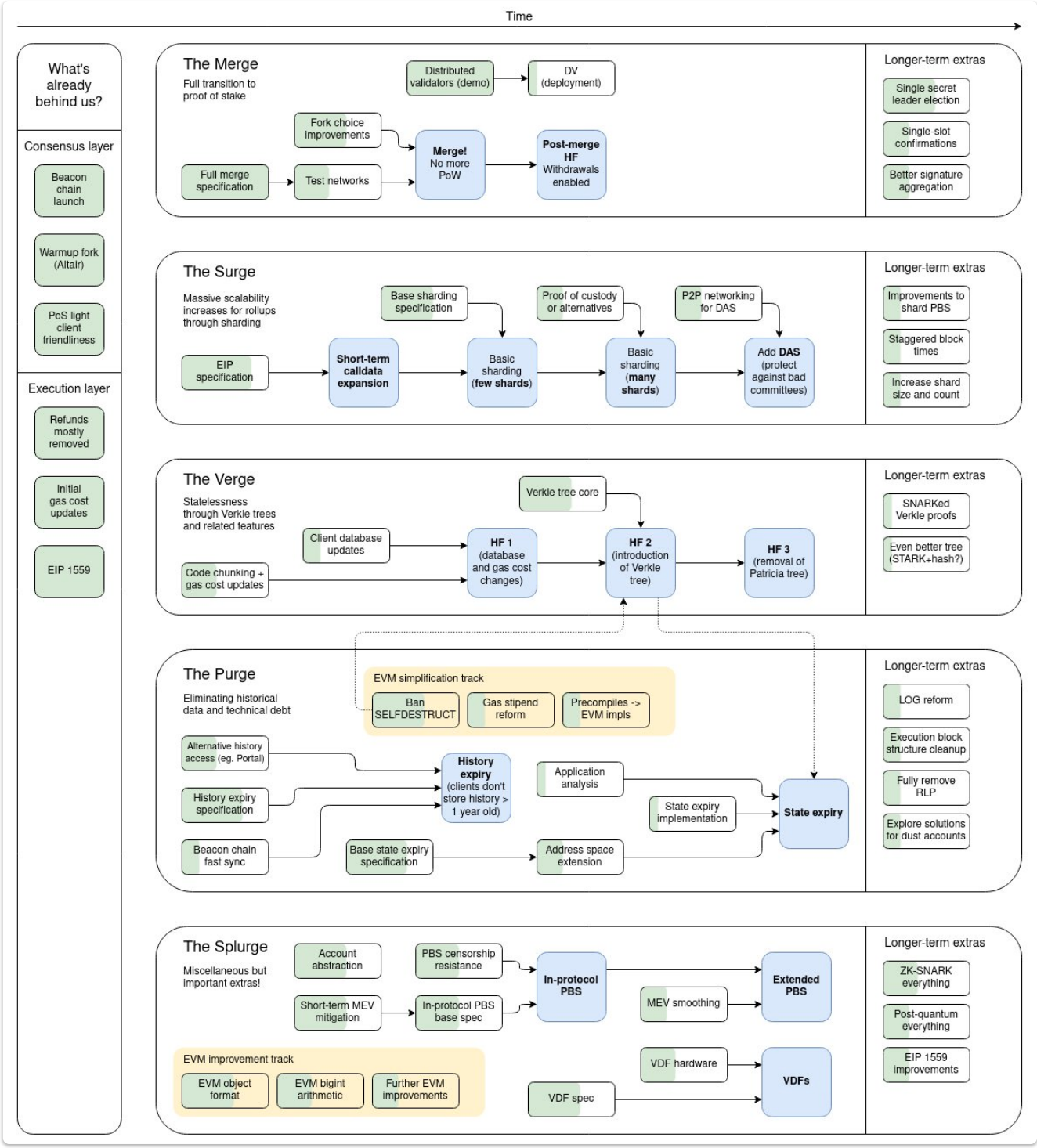
- https://github.com/kframework/evm-semantics#readme
- The K Tutorial
- https://runtimeverification.com/blog/k-framework-an-overview/
- https://medium.com/@teamtech/formal-verification-of-smart-contracts-trust-in-the-making-2745a60ce9db
- Verification of smart contracts: A survey
- Formal Verification of Smart Contracts with the K Framework
- Solc-verify, a source-level formal verification tool for Solidity smart contracts

Future of Ethereum

ETH 2 - where has the term gone ?

"To limit confusion, the community has updated these terms:"

- 'Eth1' is now the 'execution layer', which handles transactions and execution.
- 'Eth2' is now the 'consensus layer', which handles proof-of-stake consensus.



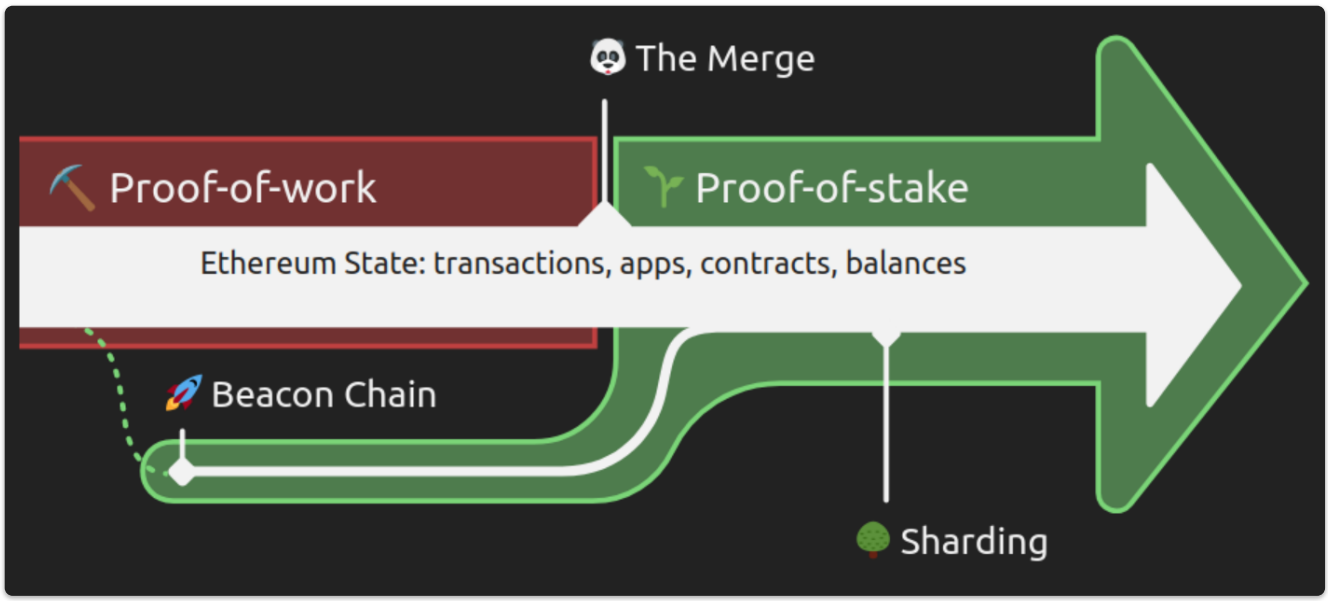
When finished this above system should be capable of 100k tx/sec (including roleups).
Source: https://twitter.com/VitalikButerin/status/1466411377107558402/photo/1

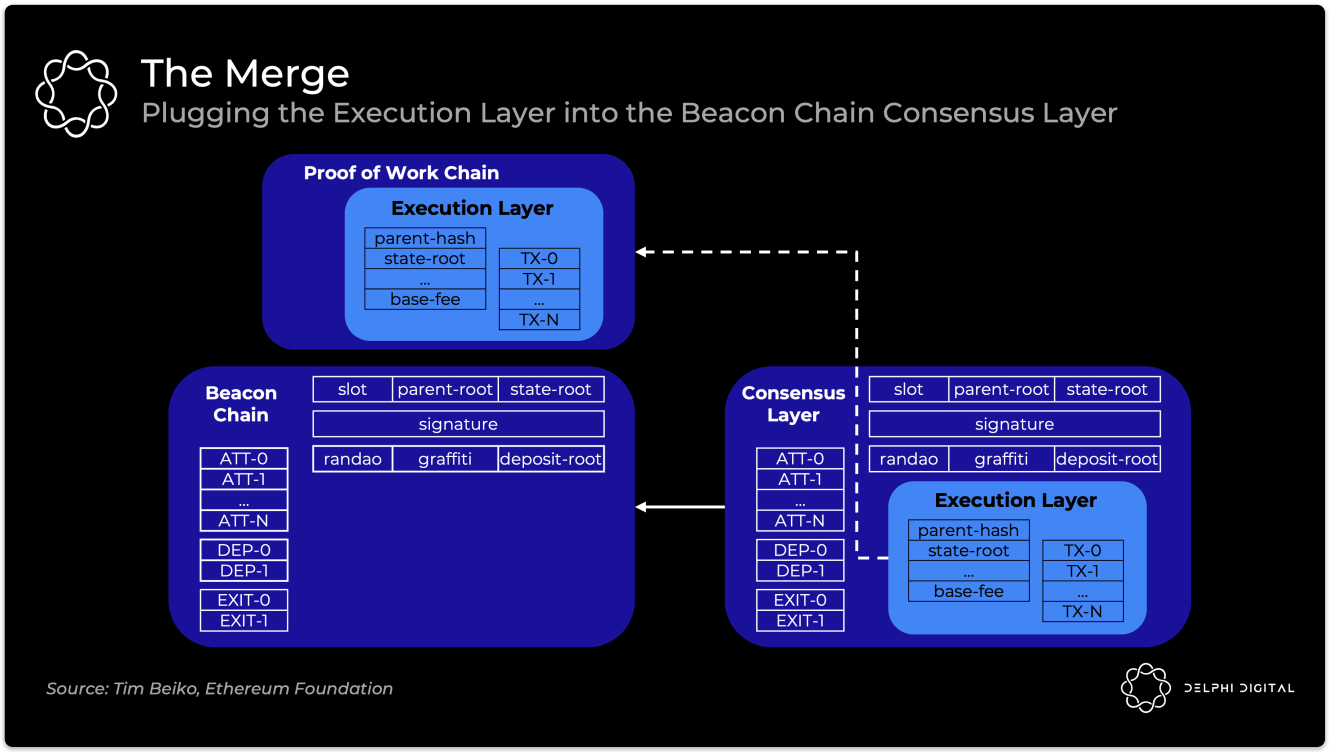
The Merge - Proof of stake update

- Replacing proof of work with the proof of stake beacon chain.
  i.e. merging existing beacon chain into ethereum.
- The Beacon Chain has not been processing Mainnet transactions. Instead, it has been reaching consensus on its own state by agreeing on active validators and their account balances.
- The blockchain state will not change.
- Estimated date for merge: 18 September 2022

- POS specs: https://github.com/ethereum/consensus-specs#phase-0

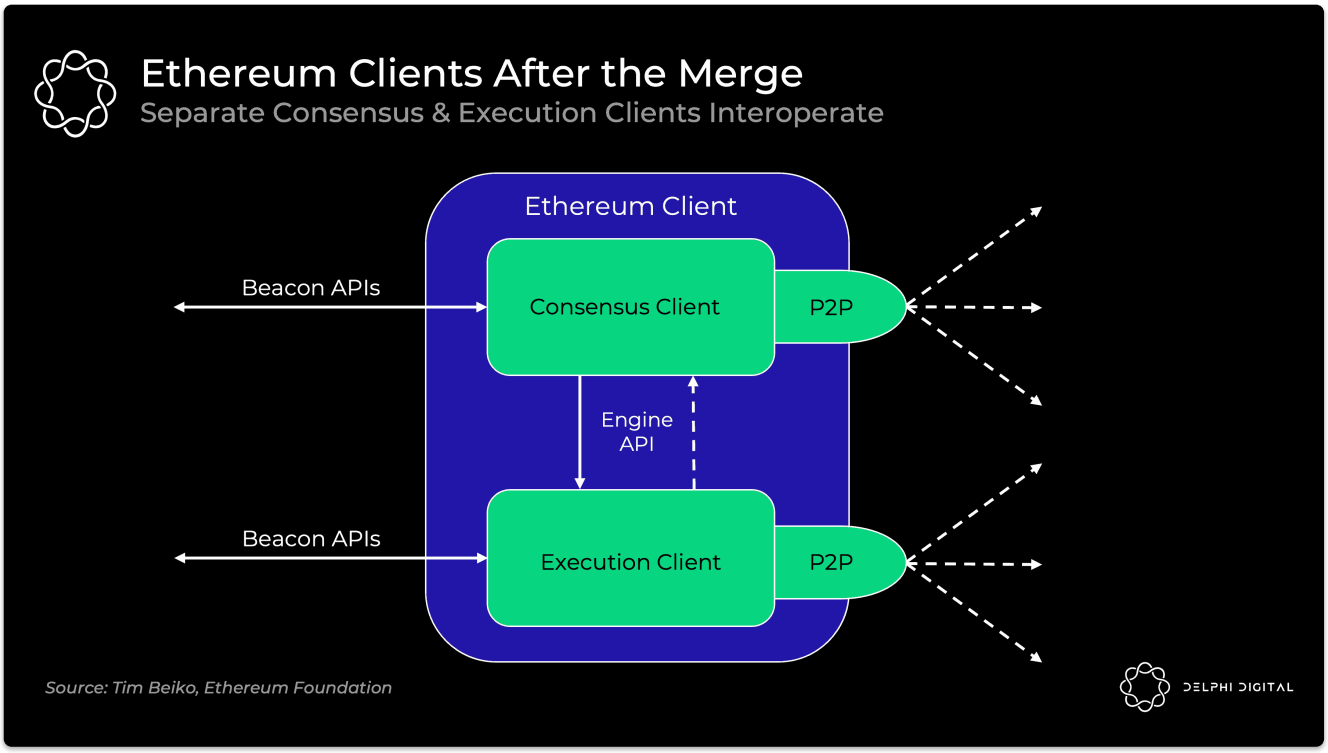
"The number `58750000000000000000000` joins the list of the most important integers of our time. The moment Ethereum's proof of work chain accumulates that much difficulty (read, hashes done), the entire network will switch over to proof of stake"





Ethereum clients after the merge

- Current Eth 1.0 clients continue to handle execution. They process blocks, maintain mempools, and manage and sync state. The PoW stuff gets ripped out.
- Consensus client – Current Beacon Chain clients continue to handle PoS consensus. They track the chain's head, gossip and attest to blocks, and receive validator rewards.



Consensus after the Merge

Ethereum will move to Gasper (Casper FFG + LMD GHOST (Latest Message Driven Greediest Heaviest Observed SubTree))

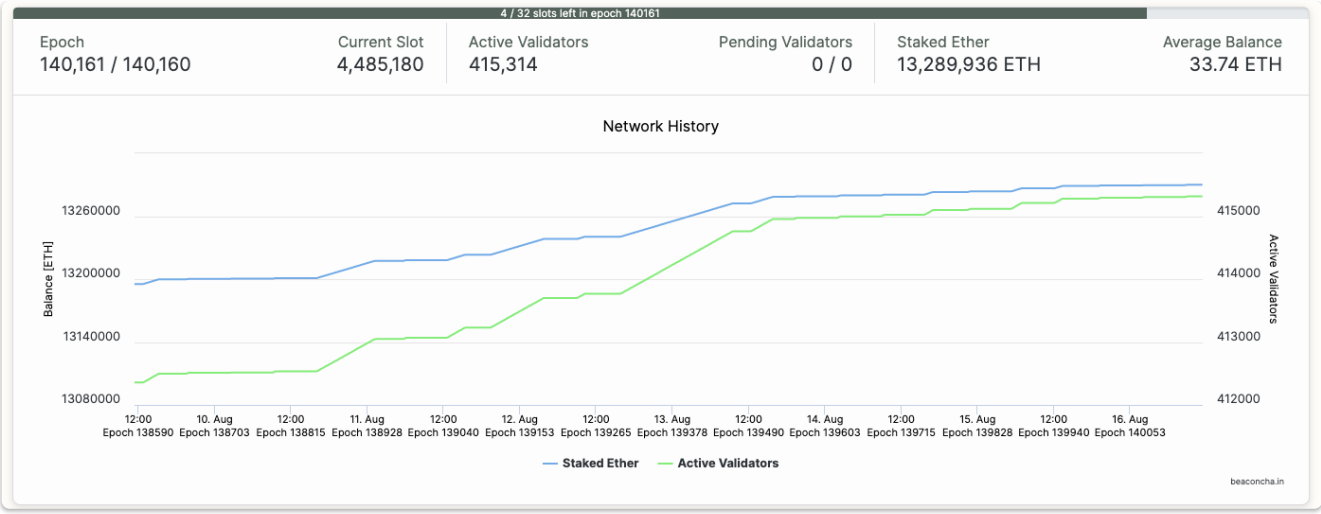
Consensus within eth2 relies on both LMD-GHOST – which adds new blocks and decides what the head of the chain is – and Casper FFG which makes the final decision on which blocks *are* and *are not* a part of the chain.
GHOST's favourable liveness properties allow new blocks to quickly and efficiently be added to the chain, while FFG follows behind to provide safety by finalising epochs.

The two protocols are merged by running GHOST from the last finalised block as decided upon by FFG. By construction, the last finalised block is always a part of the chain which means GHOST doesn't need to consider earlier blocks.

Safety favouring protocols such as Tendermint can halt, if they don't get enough votes.
Liveness favouring protocols such as Nakamoto continue to add blocks, but they may not coe to finality.
Ethereum will achieve finality by checkpointing
Epochs of about 6 mins have 32 slots with all validators attesting to one slot (~12K attestations per block)
The fork-choice rule LMD GHOST then determines the current head of chain based on these attestations.
Finality is achieved when sufficient votes are reached generally after 2 epochs.

Validator Selection and consensus in more detail

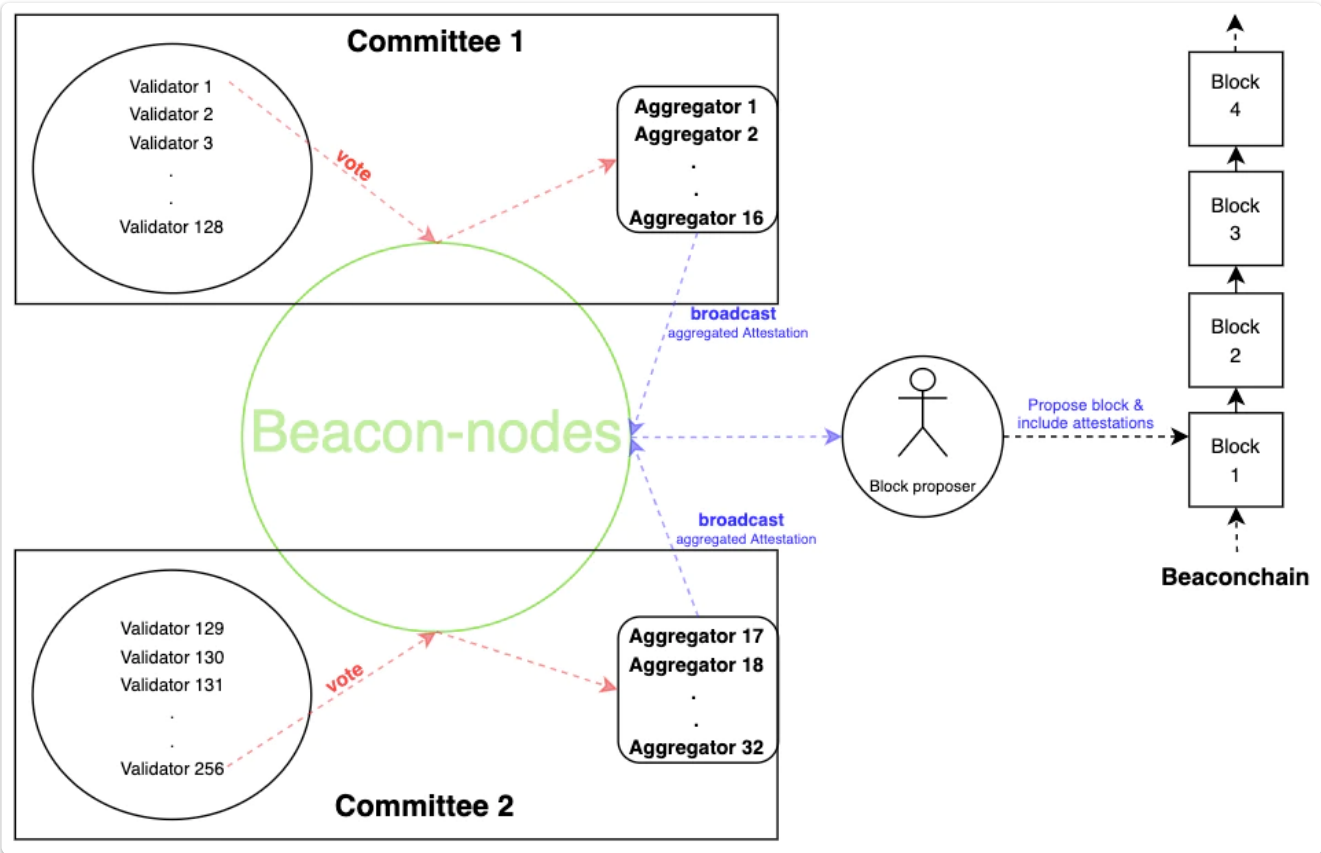
You can find stats about the validators here



A slot occurs every 12s and one validator is chosen to submit a block within that slot.
If the validator fails to do so, the slot is let empty.
The first block within an epoch (6.4 mins) is a checkpoint block.

Coming to consensus about the block

If a validator is not chosen to produce a block, it will instead vote on what it regards as the cirrent head of the chain and the checkpoint block. Within an epoch a validator will only vote once.



Validators are grouped into committees at random, their votes are aggregated and published in the block header.

Fork choice rule

When faced with a potential fork, we choose the fork that has the most votes, but when counting the votes, we only include the last one from any validator.

Voting rules

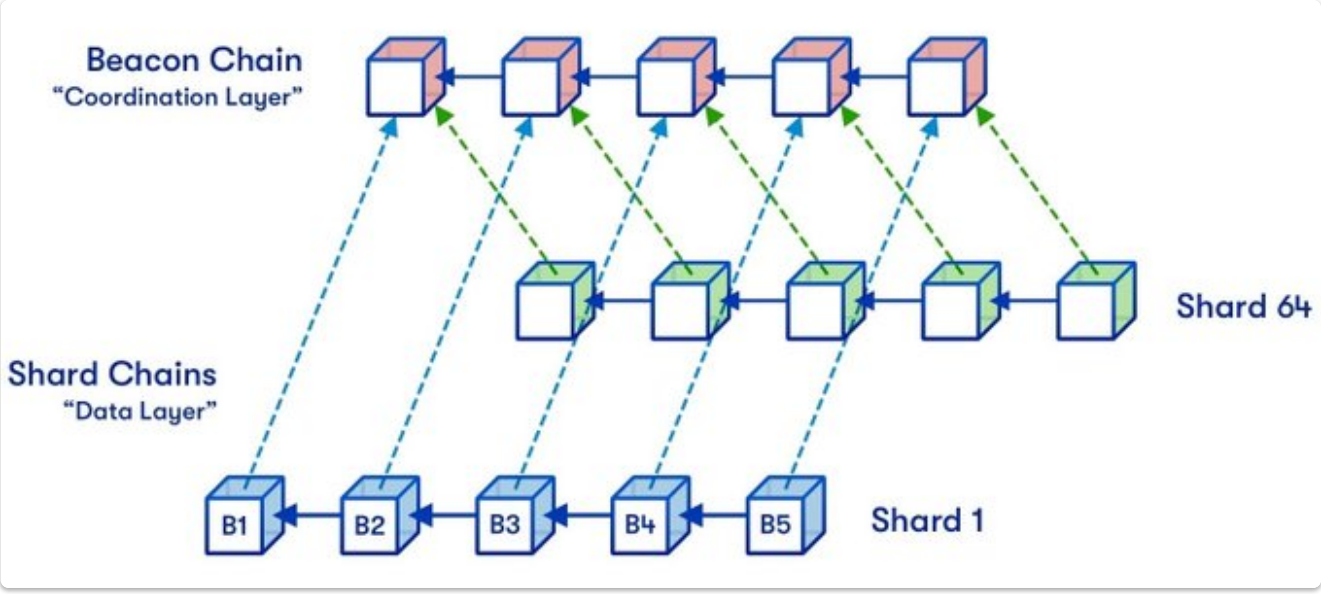
The validator must vote the chain they consider to be correct
The validator cannot vote for 2 blocks in any one time slot.

Finalisation

Validators vote on a pair of checkpoint blocks, to indicate that they are valid.
Once a checkpoit block gets sufficient votes, it is regarded as 'justified' once it's child checkpoint block becomes justified, then the parent is regarded as final.

The Surge - Calldata expansion & Sharding (Scaling)

- Existing rollups use calldata so decrease calldata gas cost as a quick way to reduce Tx costs. EIP 4488 (https://github.com/ethereum/EIPs/pull/4488) (merged)
- Sharding



Optimistic and ZK rollups under sharding

One major difference between the sharding world and the status quo is that in the sharding world, it will not be possible for rollup data to actually be part of the transaction that is submitting a rollup block into the smart contract. Instead, the data publication step and the rollup block submission step will have to be separate: first, the data publication step puts the data on chain (into the shards), and then the submission step submits its header, along with a proof pointing to the underlying data.

Optimism and Arbitrum already use a two-step design for rollup-block submission, so this would be a small code change for both.

For ZK rollups, things are somewhat more tricky, because a submission transaction needs to provide a proof that operates directly over the data. They could do a ZK-SNARK of the proof that the data in the shards matches the commitment on the beacon chain, but this is very expensive. Fortunately, there are cheaper alternatives.

If the ZK-SNARK is a BLS12-381-based PLONK proof, then they could simply directly include the shard data commitment as an input. The BLS12-381 shard data commitment is a KZG commitment, the same type of commitment in PLONK, and so it could simply be passed directly into the proof as a public input.

If the ZK-SNARK uses some different scheme (or even just BLS12-381 PLONK but with a bigger trusted setup), it can include its own commitment to the data, and use a proof of equivalence to verify that that commitment in the proof and the commitment in the beacon chain are committing to the same data.

Source: https://notes.ethereum.org/@vbuterin/data_sharding_roadmap

The Verge

Goal: Make Ethereum weakly stateless so that block validation can be fully stateless, enabling:

Validation on light machines, e.g. Raspberry Pi (without SSD) and mobile phones
Fraud proofs for the EVM
Fraud proof protected light client
EVM as a shard

How?

Use verkle tries as state commitment scheme because witnesses are small and efficient to verify
Gas cost changes to make state access cost reflect (more or less) the cost of witnesses (plus state expiry to keep active state under control, which is still relevant for block builders)

The Purge

The idea is to eliminate unused/old data to reduce the state size.
See State Expiry

The Splurge

A chance to remove old 'features'
See Feature Removal
Candidates include :

- Self Destruct

- 2300 Gas Stipend
- Gas Visibility

MEV after the merge

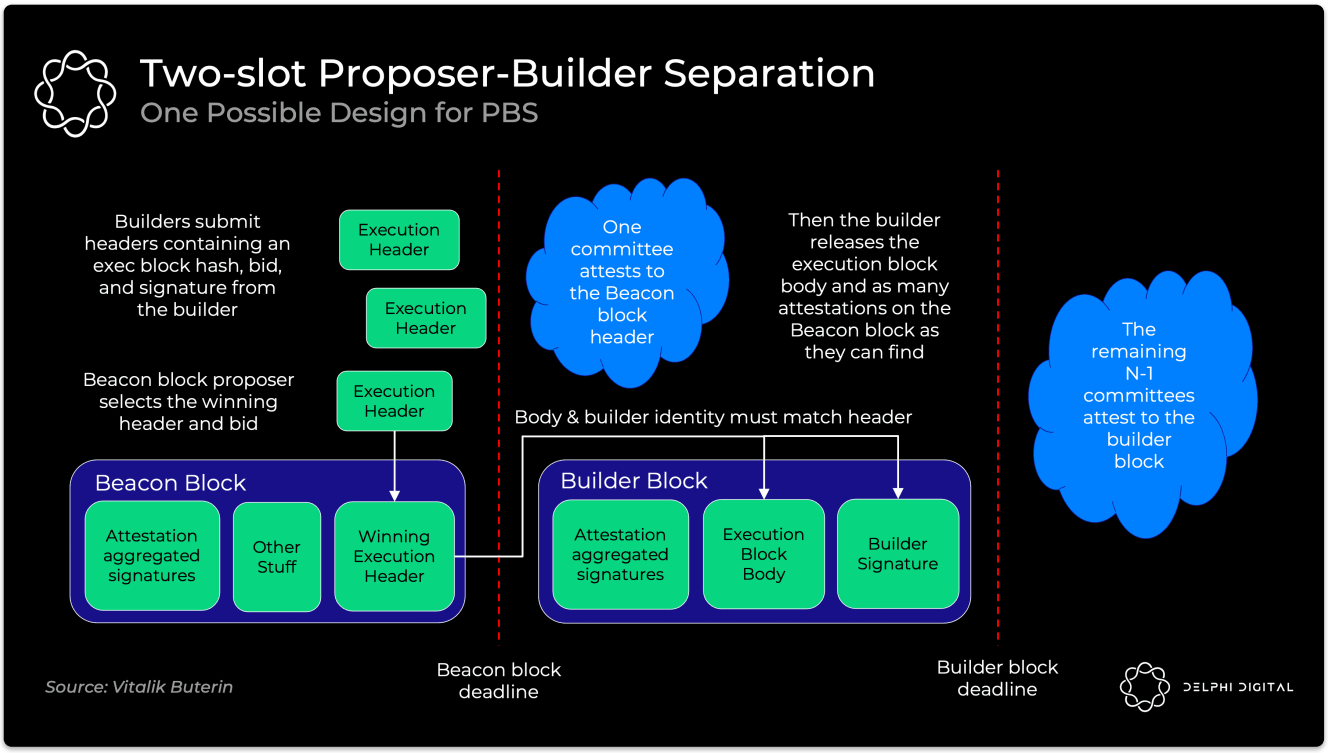
Proposer-builder Separation

From Hitchhikers guide to Ethereum
Consensus nodes today (miners) and after the merge (validators) serve two roles.
They build the actual block, then they propose it to other consensus nodes who validate it.
Miners "vote" by building on top of the previous block, and after the merge validators will vote directly on blocks as valid or invalid.

Proposer-builder Separation splits these roles by creating a new *in-protocol builder* role.
Specialized builders will put together blocks and bid for proposers (validators) to select their block.
This combats MEV's centralizing force.

Builders receive priority fee tips plus whatever MEV they can extract.
In an efficient market, competitive builders will then bid up to the full value they can extract from blocks (less their amortized costs such as powerful hardware, etc.).

A proposal for a two-slot PBS could look like this:

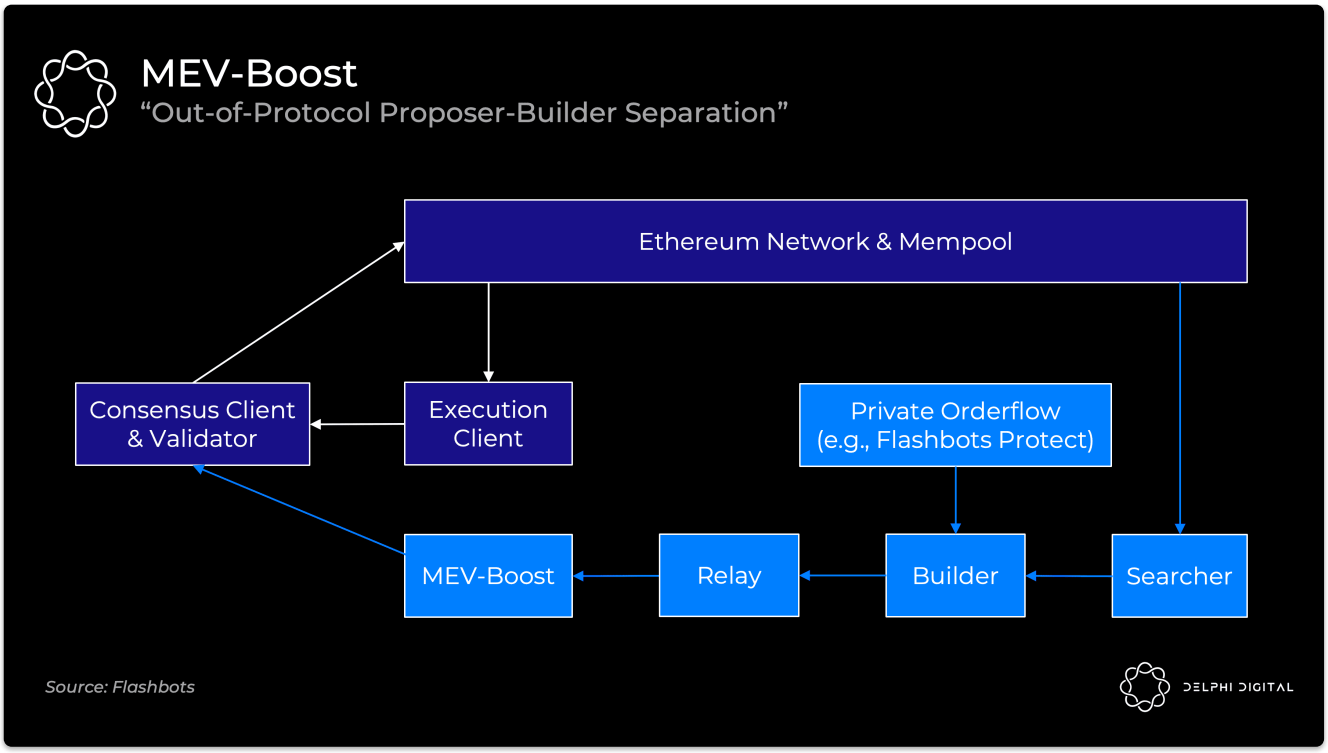


1. Builders commit to block headers along with their bids
2. Beacon block proposer chooses the winning header and bid. Proposer is paid the winning bid unconditionally, even if the builder fails to produce the body
3. Committee of attestors confirm the winning header
4. Builder reveals the winning body
5. Separate committees of attestors elect the winning body (or vote that it was absent if the winning builder withholds it)

This will not be available at the merge, as a stopgap Flashbots propose MEV Boost

MEV Boost

Validators post-merge will default to receiving public mempool transactions directly into their execution clients. They can package these up, hand them to the consensus client, and broadcast them to the network.
The average validator may not know how to extract MEV, so Flashbots a plugin to your client allowing you to outsource specialized block building.
Importantly, you still retain the option to use your own execution client as a fallback.



MEV searchers will continue to play the role they do today.
They'll run specific strategies (stat arb, atomic arb, sandwiches, etc.) and bid for their bundles to be included.
Builders then aggregate all the bundles they see as well as any private orderflow (e.g., from Flashbots Protect) into the optimal full block.
The builder passes only the header to the validator via a relay running to MEV-Boost.
Flashbots intends to run the relayer and builder with plans to decentralize over time, but whitelisting additional builders will likely be slow.

We will have more details about Verkle trees and stateless Ethereum tomorrow / Thursday