runtime verification

Understanding the K Framework

Everett Hildenbrandt, CTO

Overview

runtime verification

- K Vision: Semantics Based Tooling
- Building a Language
- Using the Semantics
- Real World K

Want to Follow Along?

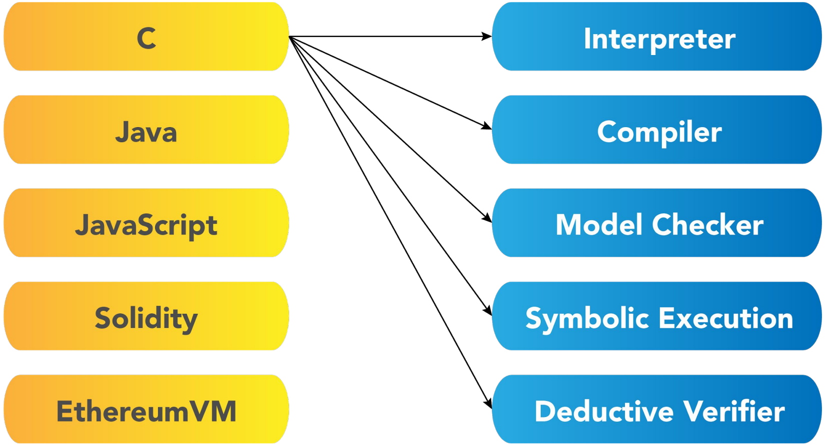
runtime verification

- See if you can get K installed before we get to the live part: https://github.com/kframework/k/releases

K Vision

The Problem: Too Many Tools

- How many tools is too many tools?



C

Java

JavaScript

Solidity

EthereumVM

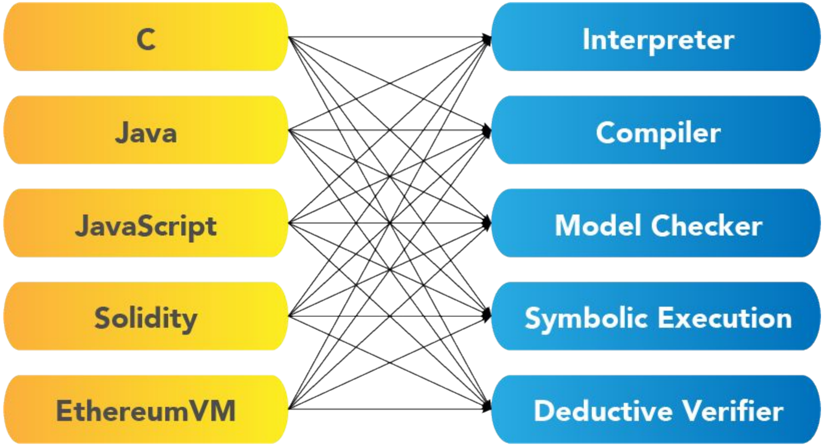
Interpreter

Compiler

Model Checker

Symbolic Execution

Deductive Verifier



C

Java

JavaScript

Solidity

EthereumVM

Interpreter

Compiler

Model Checker

Symbolic Execution

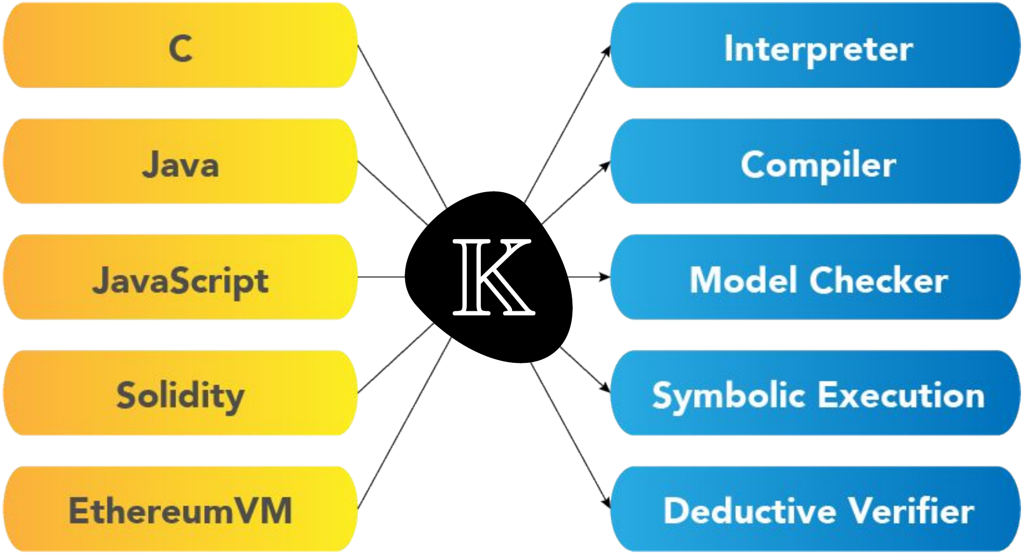
Deductive Verifier

- Quite a bit of repeated effort.

The K Approach

runtime verification

- Develop each language and each tool once:



- Save on the implementation effort!
- Updates to tools benefit *all* the languages!

What is K?

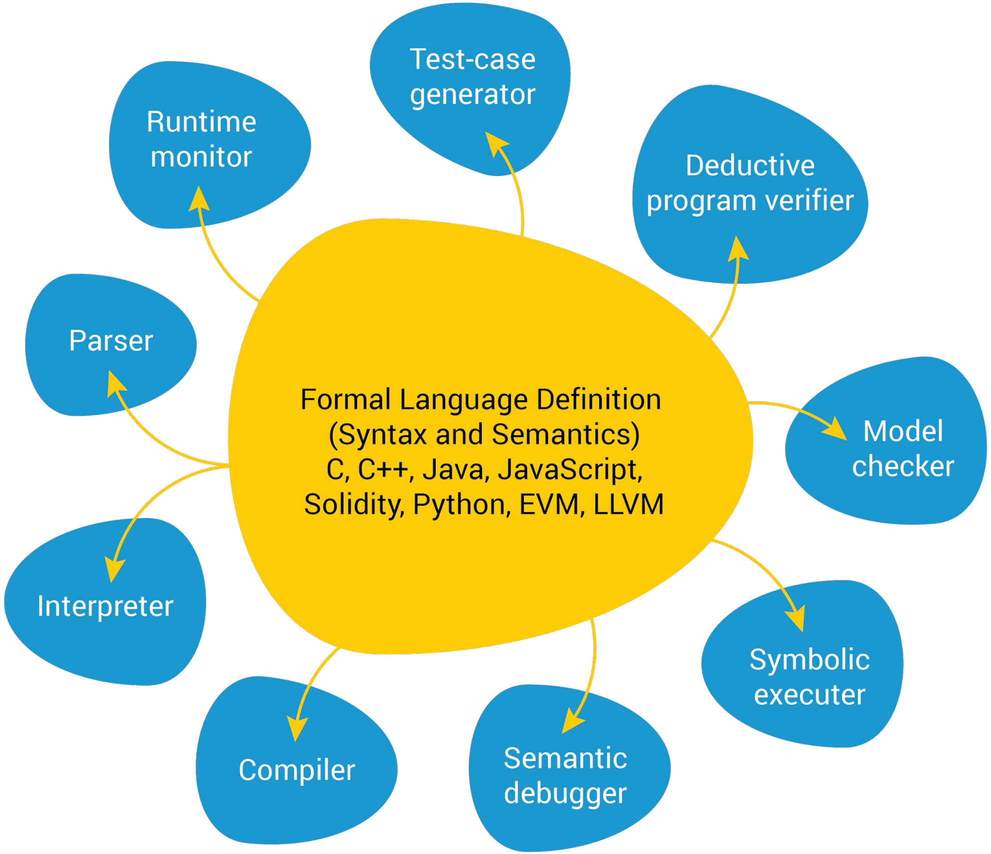
- K is an *operational semantics framework*.
  - Specify your language or system in the K modelling language.
  - The K compiler derives a bunch of tools for you from this specification.

- Given a K specification, there are two main backends you can use:
  - LLVM backend is for *concrete execution*, you get a fast interpreter out of it.
  - Haskell backend is for *symbolic execution*, you get a model checker and verification engine out of it.

- All of the tools built by RV are powered by K. Smart contract verification offered by RV is done with K.

- Webpage: **https://kframework.org**

The K Vision



Formal Language Definition
(Syntax and Semantics)
C, C++, Java, JavaScript,
Solidity, Python, EVM, LLVM

Building a Language

First: Install K

- https://kframework.org - Website with K documentation and install instructions.
- https://github.com/kframework/k/releases - Download releases for your distro
- Example for Ubuntu Bionic:
- Simple imperative language which allows for programs like this:

```
curl --output kframework.deb -sSL
https://github.com/kframework/k/releases/download/v5.0.0-0a12faf/kframework_5.0.0_amd64_bionic_202102242149.deb
sudo apt-get install ./kframework.deb
```

- Check that it worked:

```
which kompile
kompile --version
```

- Solutions to exercises here: https://github.com/ehildenb/understanding-k-framework

Build A Calculator

- Should be able to evaluate expressions like the following:

```
3 + 3
5 * 7
7 / (8 * 2)
2 ^ (4 - 2)
```

- Will make use of the *functional* fragment of K definitions

- Solution: https://github.com/ehildenb/understanding-k-framework/blob/master/01_calc.k.sol

Add Booleans

- Should be able to evaluate expressions like the following:

```
3 + 3 < 8
5 * 7 >= 9
7 / (8 * 2) == 4
2 ^ (4 - 2) != 2
(3 < 4) && (4 <= 3)
```

- Will still make use of the *functional* fragment of K definitions

- Solution: https://github.com/ehildenb/understanding-k-framework/blob/master/02_calc-bool.k.sol

Add Variables and Substitutions

- Add assignment:

```
a = 3 + 3 ;
b = 5 * 7 ;
c = 7 / (8 * 2) ;
```

- Need to add in the *stateful* portion of K definitions, called *configurations*
- Need to define *expression evaluation*; we'll use a *substitution based* approach here

- Solution: https://github.com/ehildenb/understanding-k-framework/blob/master/03_subst.k.sol

Turn into a Programming Language

- Add assignment:

```
a = 3 + 3 ;
b = 5 * 7 ;
c = 7 / (8 * 2) ;
d = a * b - c ;
```

- Need to use *K sequence* operator, which allows focusing on next part of computation
- Need to add rules to update the memory

- Solution: https://github.com/ehildenb/understanding-k-framework/blob/master/04_assignment.k.sol

Conditionals and Loops

- Add conditionals:

```
a = 3 + 3 ;
b = 5 * 7 ;
if (a < b) { c = 1 ; } else { c = 0 - 1 ; }
```

- And loops:

```
n = 10 ; s = 0 ;
while (0 < n) {
  s = s + n ; n = n - 1 ;
}
```

- Solution: https://github.com/ehildenb/understanding-k-framework/blob/master/05_control-flow.k.sol

Using the Semantics

Parsing and Running

- We’ve already been using this to run our calculator
- Parsing requires use of the `kast` tool:

  ```
  kast program.imp --output [pretty|kore|kast|json]
  ```

- Running requires use of the `krun` tool:

  ```
  krun program.imp
  ```

- Both of these tools only make use of the *concrete* backend of K, the LLVM backend.

Proving

- Proving requires that we kompile with the Haskell backend:

```
kompile --backend haskell ...
```

- And that we write a specification:

```
claim <k> n = 3 ; => . ... </k>
      <mem> MEM => MEM [ n <- 3 ] </mem>


kprove claim-spec.k [--debugger]
```

Real World K

K Example: Verification with KEVM

- Online: https://jellopaper.org

- K semantics of the Ethereum Virtual Machine.
  - Passes same conformance test-suite as other clients.
  - Enables symbolic execution (and thus verification) of EVM bytecode.

- Example standalone K proof (`transfer` function of an ERC20)

- Example lightweight symbolic analysis based audit (from Uniswap v1)

- Example full verification based audit (ETH2 deposit contract)

- Large-scale proving with K and ACT (from Multi-Collateral Dai system - 1011 proofs)

K Example: Firefly Tooling

Online: https://fireflyblockchain.com

ERC20 Verifier: https://erc20.fireflyblockchain.com/

- Ethereum client based on KEVM
  - Drop-in replacement for `ganache-cli`
  - K instrumentation added to collect additional information about your test-suite
- List of planned features
  - Currently we support test running, code coverage, and blackbox random testing.
  - Have prototypes of property testing (whitebox testing), runtime monitoring.
- Example Report
- **All powered by K**!

Same Tooling - Different Language

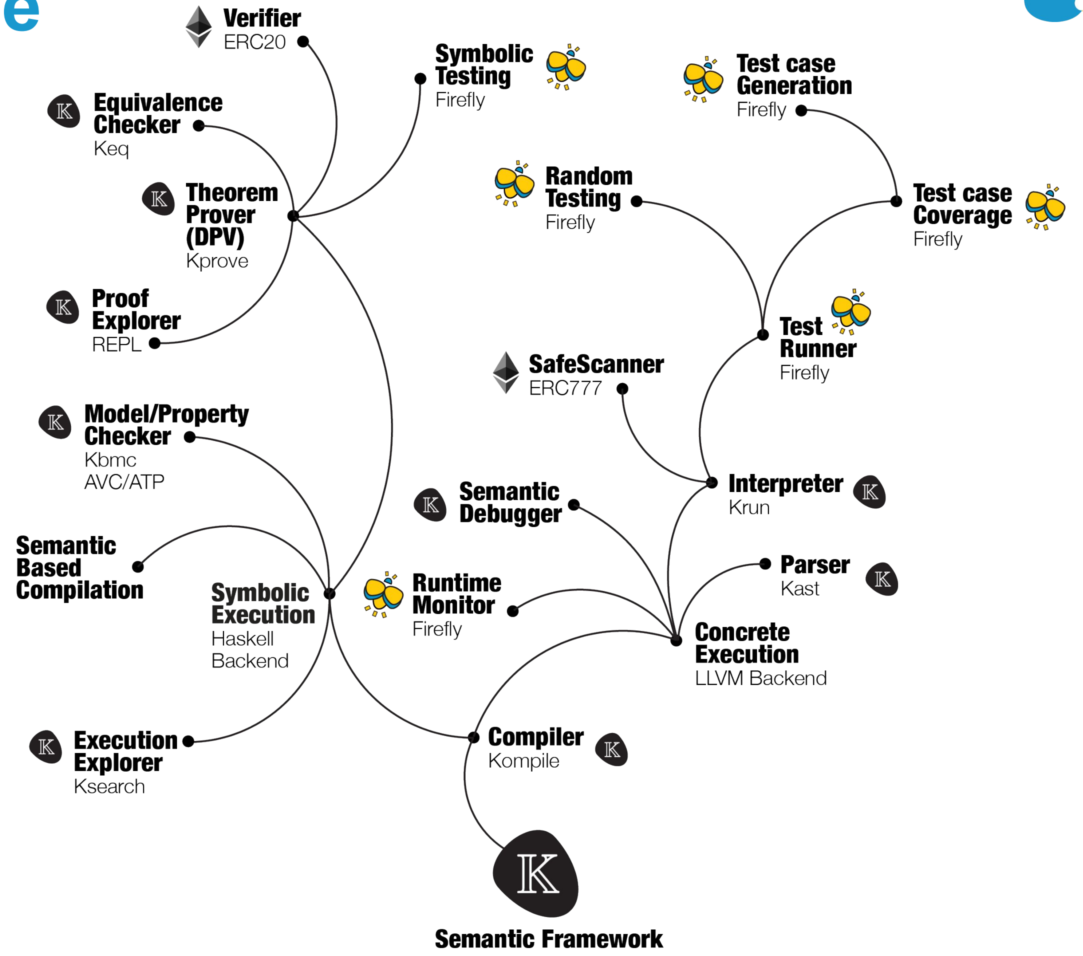
runtime verification

- KEVM is our most mature blockchain semantics.

- We also have semantics for several other blockchain languages:
  - KIELE - For Cardano
  - KMichelson - For Tezos
  - KWasm - For Polkadot and Elrond
  - KTEAL and KAlgoClarity - For Algorand (in development)
  - And others!

- Great economy of developer time!
  - Tooling development time for each semantics is limited
  - Consistent experience for developers across semantics

K Tooling Tree



runtime verification

Powered By K

- Effective security auditing requires two components:
  - Human inspection
  - Tool-based analysis

- From a single semantics (eg. KEVM), we derive tooling that:
  - Allows you to do full formal verification,
  - Allows you to do lightweight symbolic analysis, and
  - Gives you more insight into the quality of your test-suite.

- The same tooling exists (or is in development) for several other blockchains (and non-blockchain projects).

- Doesn’t matter which blockchain you’re using, K will support it!

Questions?

Big thanks to HelloDecentralization for hosting us, and Silvia for helping us get set up!