runtime verification

Formal Verification for DeFi projects

Everett Hildenbrandt
CTO
Runtime Verification Inc.

Overview

runtime verification

- Formal Verification
- Mitigating Blockchain Vulnerabilities
- Demo: ACT Specification of SafeAdd

Who are we?

runtime verification

Runtime Verification Inc. is a startup company aimed at using formal methods to perform security audits on virtual machines and smart contracts on public blockchains

It is dedicated to improving the safety, reliability, and correctness of software systems in the blockchain field (and other fields, too!)

runtime verification

FORMAL VERIFICATION.

What’s Formal Verification

runtime verification

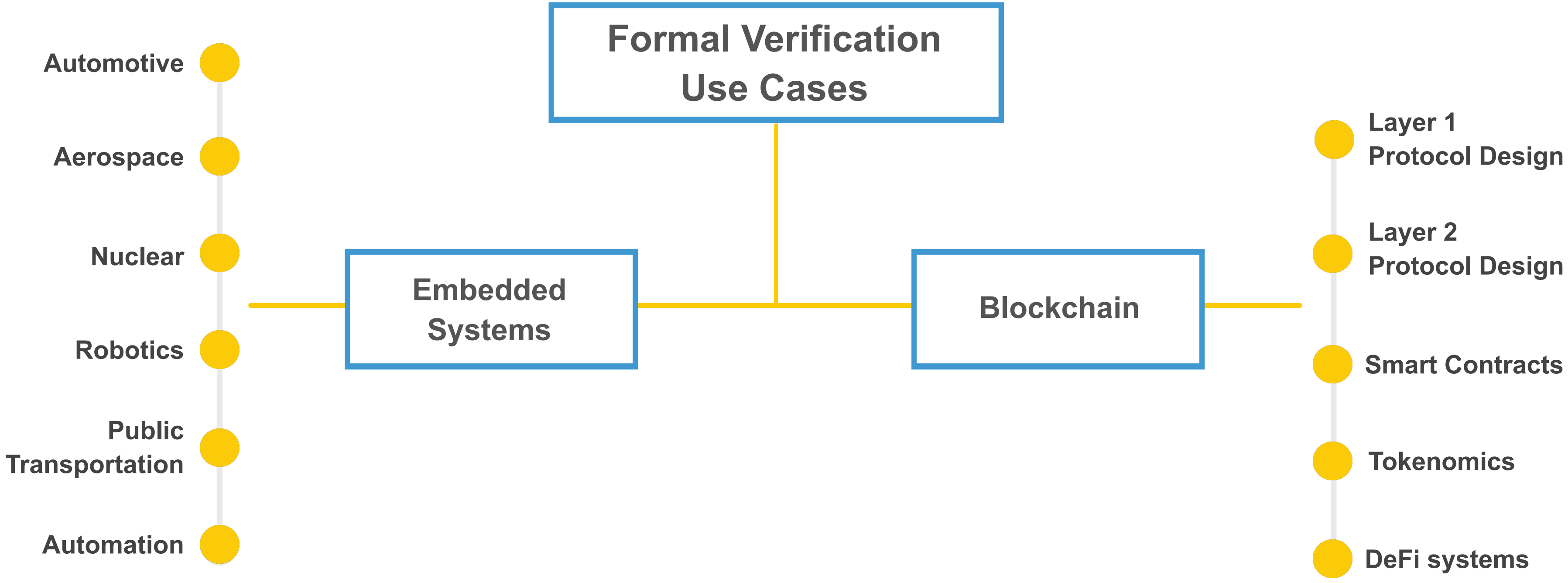
Formal verification is the process of checking if a design satisfies its system requirements (properties). In other words, to check if the code behaves as expected, using some *formalism*.

Formal verification covers all input scenarios and also detects corner case bugs, catching many common design errors and uncovering ambiguities in the design.

(Like property based testing, but using symbolic inputs, not random inputs.)

Formal Verification Use Cases

runtime verification



MITIGATING BLOCKCHAIN VULNERABILITIES.

Blockchain Security

- Blockchain and Smart Contracts are an attractive target for hackers:
  - Open-sourced & easy to access Smart Contracts.
  - Short code in Smart Contracts.
  - Funds (millions of $) on the contracts (DeFi).

- It can be done in a multitude of ways:
  - **Design Vulnerabilities.**
  - **Code Vulnerabilities.**
  - Social Engineering.
  - Legal Pressure.
  - Phishing.

- **RV focuses on the first two categories: design and code vulnerabilities.**

Design and Code Vulnerabilities

Design vulnerabilities are flaws in the business logic of a blockchain or a smart contract.

Code vulnerabilities occur when the programming to implement a given design doesn’t actually work as the design intended.

Mitigating Vulnerabilities

DESIGN VULNERABILITIES

- **Specification**. Make human-readable description of the system as detailed and complete as possible to leave no cases out.
- **Audit**. A third party reviews the design specification, looking for incorrect assumptions and flaws in the logic.
- **Modelling**. Make computer-readable specification of the system using a programming language or using a modelling language (such as K).
- **Verification**. Poke the model a bunch, until you're convinced it behaves as expected and prove there aren't any issues.

CODE VULNERABILITIES

- **Testing.** It makes sure the code works on a few cases with, for example, unit tests and regression tests.
- **Audit.** A 3rd party reviews the code, looking for general programming errors, logical errors and known bugs.
- **Analysis**. A computer tool takes a look at the code for you (broadly static and dynamic). Some look for specific problems and others produce a summary to be interpreted by the user.
- **Verification**. Specify the intended behavior of the code with a computer-readable design model and check that the model and the code agree.

Specifications Specifications

- To do formal verification, you need:
    - The code you want to verify (Solidity source)
    - A specification of the code (what *should* the code do?)

- How to write a specification?
    - On paper (in natural language)
    - Using a mathematical formalism (K, Coq, Isabelle, F*, ACT, etc…)

- What do with the specification?
    - Check that the code matches it! (by hand, using tool)

Demo: ACT Specification of SafeAdd

Solidity Code

```
pragma solidity >=0.4.21;

contract SafeAdd {
    function add(uint x, uint y) public pure returns (uint z) {
        require((z = x + y) >= x);
    }

    function addBad(uint x, uint y) public pure returns (uint z) {
        z = x + y;
    }
}
```

ACT Specification

runtime verification

```
behaviour add of SafeAdd
interface add(uint256 X, uint256 Y)

iff in range uint256

    X + Y

iff

    VCallValue == 0

returns X + Y
```

Real World ACT Specification: MCD

- Repository: https://github.com/makerdao/k-dss
- 1011 correctness specifications
- When code is updated, or KEVM is updated, specifications re-checked

Just One Example

- This presentation was to emphasize that you need a *specification*
  - ACT is a great language for demoing that.
  - Many real-world verification efforts do *not* use ACT though.
  - Even best to start with a natural language (eg. English) spec.
- You can, for instance, write K specs directly
  - ACT actually ends up generating several K specs.
  - Tutorial on proving in K directly:

Video 1: https://www.youtube.com/watch?v=QHsERS1PyXk

Video 2: https://www.youtube.com/watch?v=aRvYFAcKdD4

Video 3: https://www.youtube.com/watch?v=p7M8gnLmWh0

Video 4: https://www.youtube.com/watch?v=3OiEOUr-9XM

Video 5: https://www.youtube.com/watch?v=P9PSqRCv4p0

Video 6: https://www.youtube.com/watch?v=JGUXrNyfU_A

We are Hiring

runtime verification

- Smart Contract Auditor
- Haskell Developer
- Compiler Engineer
- Internship

Send us your resume and cover letter at careers@runtimeverification.com

runtime verification

Questions?

https://runtimeverification.com/

@rv_inc

https://discord.com/invite/CurfmXNtbN

contact@runtimeverification.com