We look at formal verification & symbolic execution with two Trail of Bits Web3 security team members. Additionally, we review the value these techniques bring and compare them to other tools.

Introduction

Formal verification

is the act of proving or disproving a given property of a system using a mathematical model.

Symbolic execution

is one technique used for formal verification. Symbolic execution explores the different paths in a program, creating a mathematical representation for each path.

Plainly stated, symbolic execution is converting your code to a set of mathematical expressions. Or even more plainly, making your code math.

Is this the silver bullet for your auditing journey? Let's find out.

The Team

Josselin interview thumbnail

I had the absolute pleasure of interviewing Trail of Bits head of blockchain engineering,Josselin and security engineer from Trail of Bits, Troy, about testing methodologies and formal verification. I have links to the full interviews with both of them (Troy, Josselin) if you'd like to watch!

To understand what formal verification and symbolic execution actually are we need a quick refresher on some of testing in Web3. If you haven't seen my invariant testing video, be sure to watch that first before reading this.

When testing, what tools do we use to have high assurance that our program works the way we want it?

Layer 0| Manual Review

Every auditor and smart contract engineer should be able to manually review their own code. ChatGPT isn't good enough. Manual review is great, but we need to make sure we don't rely purely on humans for results, we need to have automated processes so we can have more assurance that bugs are found.

Layer 1 | Unit Test

Obviously, you have unit tests, which test a very specific thing. Having a unit test for every line of your code will give you good code coverage. And it's the bare minimum for testing.

Example Solidity Function

For example, if the above is our solidity, and our setNumber

function should set number

to newNumber

. A unit test would be able to catch this.

Foundry unit test output

And if we're using Foundry, we could get an output that looks like the above.

If a test fails, this means that our test caught an issue, and we can go back into our code to fix.

Most frameworks like Foundry, Hardhat, Apeworx, Truffle, and Brownie all have unit testing capability.

Layer 2| Fuzz Test

Fuzzing is where you take random inputs and run them through your program. So you have to define things in your code that you always want to hold true.

— Troy

Fuzz testing is the new bare minimum for Web3 security.

Cuz I say so.

If you haven't watched my video on fuzzing… Do it!

For fuzzing/invariant/property tests, you need to understand the property or invariant of your system to do fuzzing. Once you have your property defined, you throw random data at your system to break that property. If you find something that breaks it, you know you have an edge case that you need to refactor your code to handle.

Foundry, Echidna, and Consensys Diligence all have fuzzers.

Layer 3 | Static Analysis

Unit testing and fuzz testing are known as dynamic testing. Dynamic means that you're actually doing something, like actually running our code.

- Troy

In static analysis, we just look at our code or have some tool look at our code. For example, this code here has a classic reentrancy vulnerability in our withdrawal function.

Vulnerable Code

If we run a static analysis tool likeSlither, it'll automatically detect this error. This is great for very quickly picking out very specific parts of your code that are known to be bad practices.

Even the Solidity compiler could be considered a static analysis tool.

Layer 4 | Formal Verification

Formal verification is going to be the act of proving or disproving a given property of the system. This is usually done through a mathematical model of the system and the property.

— Josselin

There's that word again, property. You see that almost no matter what you're doing in your testing, you need to understand the properties of your system, and right there, Josselin gave us some of the keys between fuzz testing and formal verification.

- Fuzz testing tries to break properties by throwing random data at your system.
- Formal verification tries to break properties using mathematical proofs

There are many different ways to do formal verification such as:

- Symbolic execution
- Abstract interpretation
- Model Checking

For this blog, we're gonna focus on symbolic execution, as that's one of the most popular ways currently done in Web3.

Symbolic Execution

Example of path exploration

You can learn more about symbolic execution in general fromthis MIT video.

Symbolic execution is a technique where you try to explore the different paths of the program and represent those paths as

mathematical expressions to try to prove something. For every execution path, you're going to create a mathematical representation. So, the first thing we need to do is figure out what we wanna prove or disprove.

Sample Code

For our demo above, let's say our invariant is that f

should never revert, and that's what we're going to try to prove or disprove. Now, this might seem like a silly example, but you can imagine that this was a function called withdraw money, and you want users to always be able to get their money out. Which would seem like a much less silly example.

In symbolic execution, we're going to convert this function to a mathematical/logical representation of every execution path from our code. Once we have a set of math functions, we can push those into a solver, which will tell us if a property is true or false, or if our invariant is true or false. To use symbolic execution for formal verification, we follow these steps:

1. Explore all possible paths
2. Convert paths to mathematical expressions
3. Send mathematical expressions into a SMT/SAT Solver

Exploring the paths

Visual of the process of formal verification with Symbolic Execution

In this example, we have 2 paths:

- Path 1: We return (a + 1)
- Path 2: a + 1 overflows and we revert

If we pass the max uint256

value, and try to add 1 to it, solidity will revert (as of solidity 0.8). These are the two possible paths our code can take. Software tools can find these automatically, but we will show you those later.

Now that we have our 2 paths, we convert them to mathematical expressions. One of the most popular set of expressions is to convert them to a set of booleans, like so:

Path 1

- a < type(uint256).max

Path 2

- a == type(uint256).max
- a + 1 < a

Path 2

can only be achieved if a > a + 1

since we revert when 1 is added to a

. This set of booleans will then be placed into a SAT solver, which will try to find values for a

to make all the booleans true. If it is able to find a value for a

, it will consider the group of booleans "solvable" or sat

.

With our small example, you can see how it's easy to make a group of booleans, but if we get more complex functions, it gets harder. We want to convert this list of booleans to a SMT-LIB language to give to a sat solver, the above example, could be re-written in SMTLIB as such:

; Declare a symbolic integer variable 'a' as a 256-bit integer

(declare-const a (_ BitVec 256))

; Create a context for Path 1

(push)

; Add assertions for Path 1

; assert a is not equal to uint256.max

(assert (distinct a #xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff))

(check-sat)

; Remove the context for Path 1

(pop)

; Create a context for Path 2

(push)

; Add assertions for Path 2

; assert a is equal to uint256.max

(assert (= a #xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff))

; bvult is "bit vector unsigned less than", so we are checking that a + 1 is less than a

(assert (bvult (bvadd a (_ bv1 256)) a))

; Check if Path 2 is satisfiable

(check-sat)

Now, if you take this code and paste it into a tool like Z3 or run it locally on your machine, it'll give you an output that looks something like this:

Z3 Output example

The two sat

outputs mean they were able to find an input to make the set of booleans for each path true — and since path 2 reverts, and our invariant is that it should never revert, we proved our invariant breaks!

Now I manually created this SMTLIB code with the help of ChatGPT. However, symbolic execution tools like Manticore, HEVM, and even the Solidity SMT checker, can give you an output like this. But all those tools come with a built-in Z3. So they'll even just skip the step of converting to booleans and just give you the output of the SMT Solver.

If you want to see a breakdown comparing many of the tools out there doing symbolic execution, check out this post by Palina.

Even the Solidity compiler itself can do this entire process behind the scenes:

- Explore the paths
- Convert the paths to a set of booleans
- Check to see if those paths are reachable or not

Using the solc compiler, we can run with Model Checker Engine, and we can look for an overflow.

Example of running the solc SMT Checker

If we run this, you'll see the Solidity Compiler was able to do symbolic execution. Now, reverts are pretty easy to find, but. We could instead look for reverted asserts

. A does not equal one, rerun this, but instead of overflow, look for asserts.

We first would add an assert into our code

Add an assert

Then run the solc checker looking for broken asserts (invariants).

Solidity assert checker

And we would see that again, it was able to find an input to break our assert mathematically.

Recap

So a lot of stuff just happened here. Let's recap.

1. We built some solidity.
2. We understood our invariant.
3. We used a symbolic execution tool like the built-in one in solidity to create a set of boolean expressions that represent every execution path of our code (this happened automagically)
4. Then we dumped them into a solver like Z3 (behind the scenes) to see if our property could be broken just by running this one function

We go through a full walkthrough of this example in the interview with Josselin, so be sure to check that out as well if you want to learn more.

Don't be afraid if this seems a bit complicated; be sure to ask questions and leave comments in the descriptions.

Limitations

Thumbnail of interview with Troy

Is this a silver bullet?

No.

Sometimes the server might not be able to solve the question too, like if the question is too complex. We usually provide a timeout to the server just because, you know, if you have to invert a hash function, you know, good luck to do that with the server.

- Josselin

Like any technology, these are not a one size fits all approach. Using symbolic execution can run into something called the path explosion problem; where there are too many paths for a computer to explore in a reasonable amount of time and a solver would never be able to finish.

In Practice

How practical is it to take all these steps? How hard is this to really do well?

This technique requires significant effort to be used. You need to understand how they work, and you need to understand their limitation and how to help them, and also significant effort to be maintained at the end of the day. I think what really matters are the properties. If you want to know if a bug can occur and if the property can be broken, you don't necessarily need formal method for that and you can use a fuzzer, which is way easier to use and provide like kind of the same type of value.

- Josselin

The Trail of Bits team has created secure-contracts.com to help developers understand properties so they can build and test code in a property-based testing way.

Sometimes a sufficiently powerful fuzzer is all you need, and symbolic execution and formal verification is overkill.

Additionally, even formal verification doesn't prove your code is bug-free.

All it does is mathematically prove your code does that one specific thing correctly.

I'm hoping as AI takes off doing a lot of this will become much easier and I guess we'll have to see.

Conclusion

But for now, hopefully, you learned at least the basics of symbol execution. If you'd like to learn more, leave a comment and a clap.

😸😸Follow Patrick!😸😸

Book a smart contract audit: Cyfrin

YouTube

Twitter

Medium

TikTok

Twitch Stream Uploads & Shorts

Formal Verification w/ Trail Of Bits

A huge thank you to Trail of Bits

[Troy

](https://twitter.com/0xalpharush) &

[Josselin

](https://twitter.com/Montyly) for taking the time on this interview. Additionally, special thanks to

[Hari

](https://twitter.com/_hrkrshnn),

[Runtime Verification

](https://twitter.com/rv_inc),

[Leo Alt

](https://twitter.com/leonardoalt), and

[Palina

](https://twitter.com/palinatolmach) for their help in understanding these concepts and work in the field.