

Harnessing SMT Solvers for Formally Verifying DeFi Protocols

Mooly Sagiv



Tel Aviv University

And Also



Jaroslav Bendik



Jochen Hoenicke



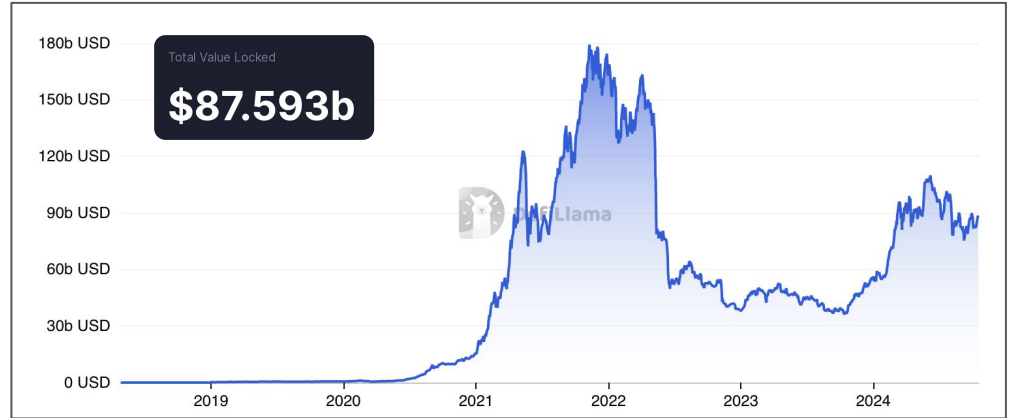
Antti Hyvarinen








Roy Perry

DeFi in one slide

- Economic process completely defined by code
- Fairly complex code
- Examples
 - Stable coins
 - Lending
 - Exchange
 - Options
 - Auctions
- 300 Billion dollars in the bear market



Pain Point: Financial losses due to bugs in smart contracts

 Reentrancy attacks	\$110M
 Rounding Errors	\$232M
 Price manipulation attacks	\$486M
 Other logical mistakes	\$493M
 Access Control and Governance	\$1.3B

Why Formally Verify DeFi?



Code is law



Billions of dollars at stake



Code is typically small/modular with natural



But bugs are hard to find
Happens in rare scenarios



New code is produced frequently

<https://www.ibtimes.com/economy-markets>



Maker
@MakerDAO

UPDATE ON MULTI-COLLATERAL DAI:
The code is ready and formally verified. The first time ever

He further revealed that significant safeguards were put in place to prevent an attack similar to the 2023 exploit. For one, Euler went through code security provider Certora's verification. "Certora's formal verification has successfully proven the 'Holy Grail' property for the Euler v2 Vault, ensuring that accounts stay healthy under all conditions. This robust approach would have prevented the Euler v1 vulnerability, providing strong assurance for the security-first Euler v2," he explained.



This is proper

7:16 PM



51



63



500



14

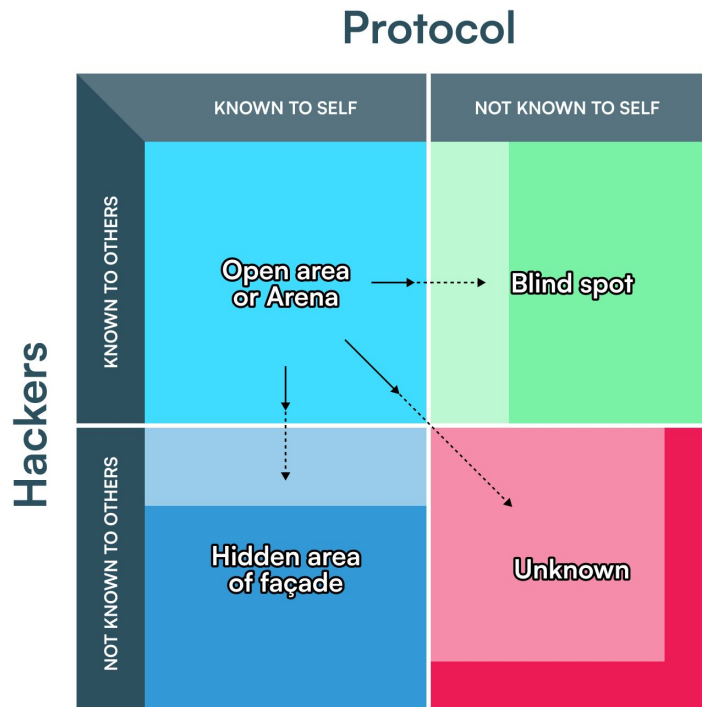


Agenda

- Overview of the Certora Prover technology
- Automatic Market Makers (AMM)
- Blackbox SMT timeout Mitigation applied to AMM

Formal Verification Prevents Unknown Attacks

- Security is asymmetric:
 - System defenders must rule out all vulnerabilities
 - Attackers need only find one
- Verification can break the asymmetry and rule out unknown threats
 - Need to prove the right properties!

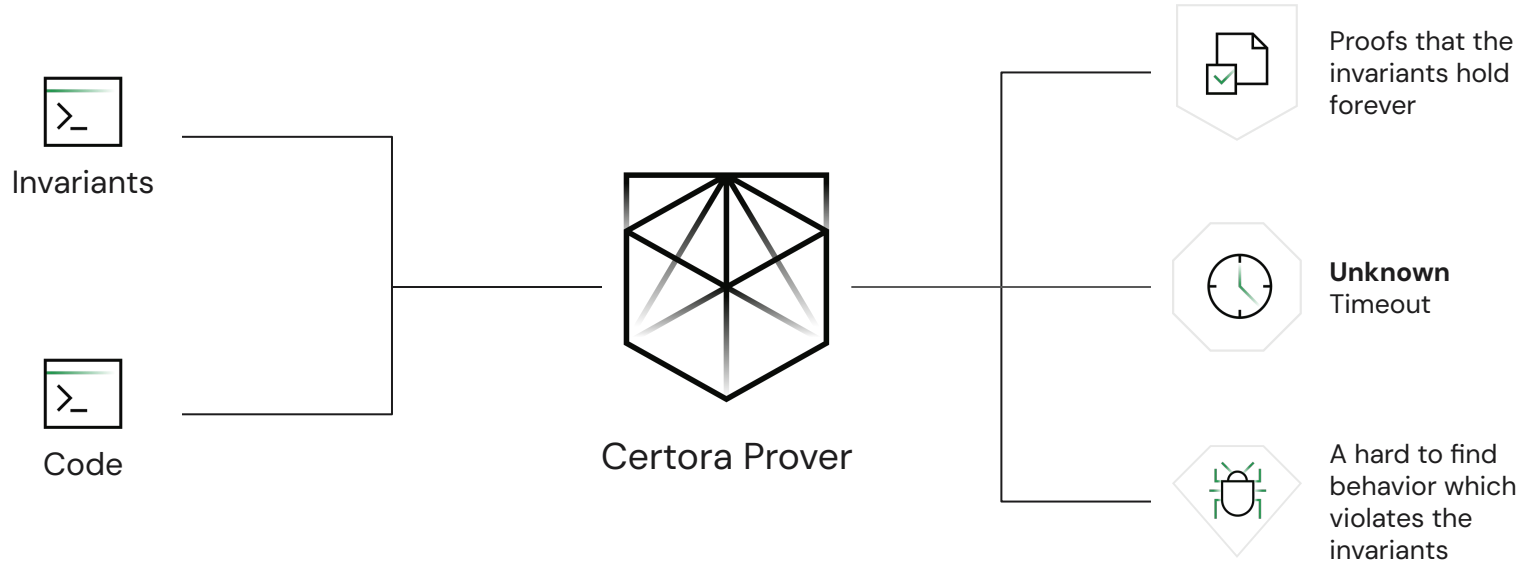


https://en.wikipedia.org/wiki/Johari_window

Credit Ernesto Boado

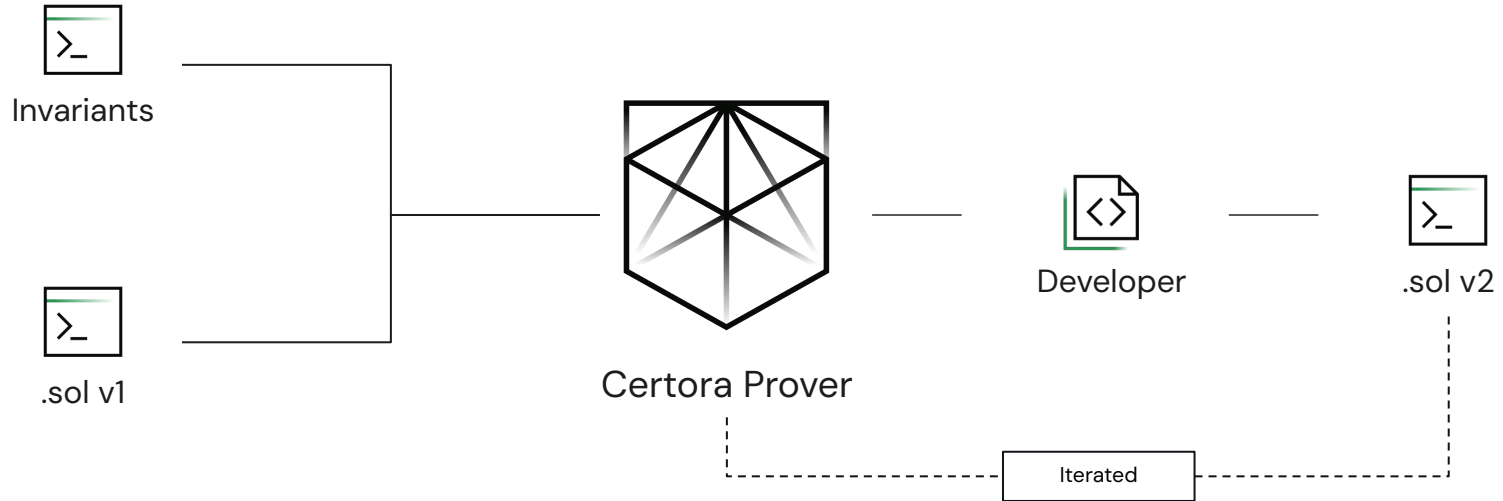
The Certora Prover:

Automatic formal verification



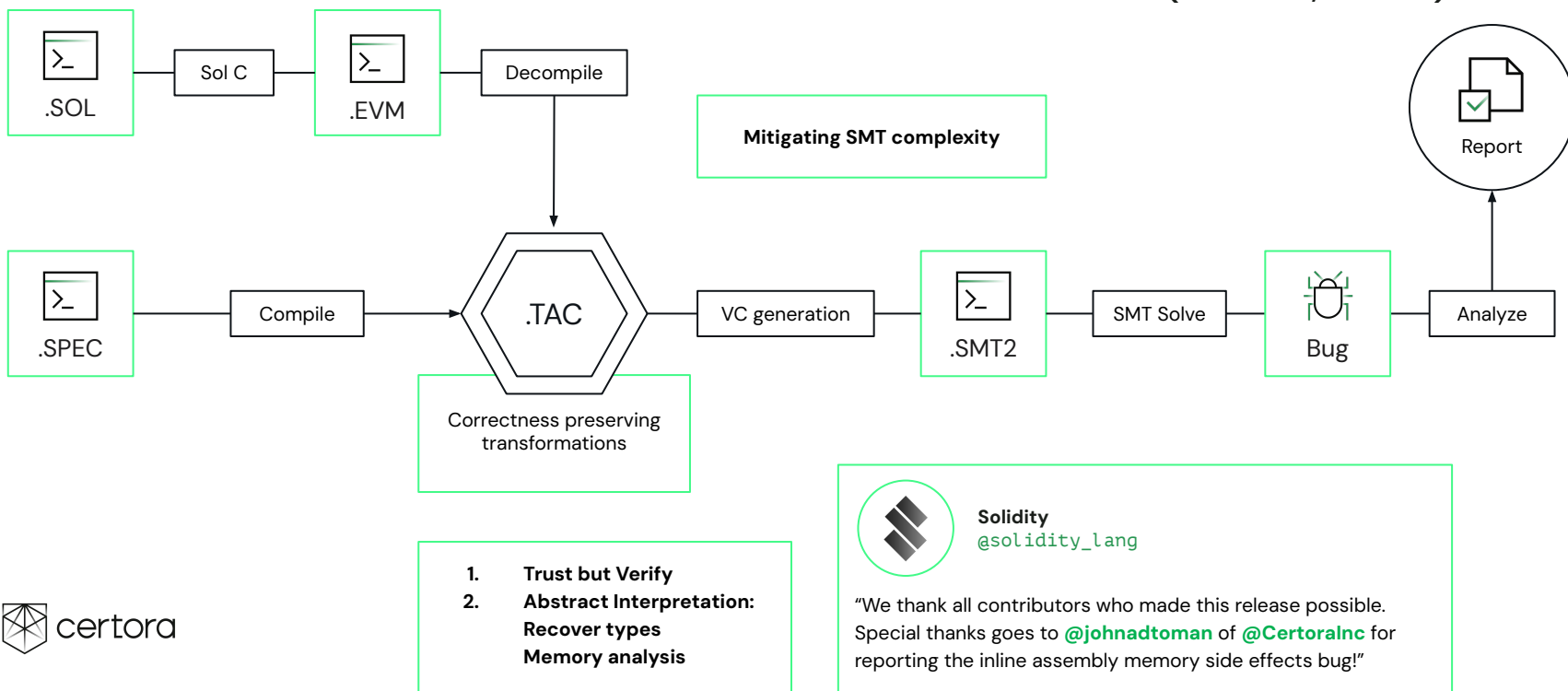
CI Integration

Verification driven development



Certora Prover Architecture

Similar versions for Rust(WASM, eBPF)



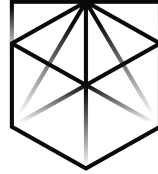
Simple Example

Money transfer

< >

CODE

```
transfer (address from, address to, uint256
amount) {
  require (balances[from] >= amount);
  balancesFrom = balances[from] - amount;
  balancesTo = balances[to] + amount;
  balances[from] = balancesFrom;
  balances[to] = balancesTo;
}
```



Certora
Prover

< >

TEST

```
From = "Alice"
To = "Alice"
Amount = 18
old.balances(Alice) = 20
new.balances(Alice) = 38
```



< >

INVARIANT

```
total =  $\sum_{a: \text{address}} \text{balances}[a]$ 
```



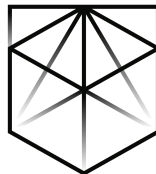
Simple Example

Money transfer

< >

CODE

```
transfer (address from, address to, uint256
amount) {
  require (balances[from] ≥ amount);
  balances[from] := balances[from] - amount;
  balances[to] := balances[to] + amount;
}
```



Certora
Prover

< >

PROOF

```
 $\Sigma$  a:address old.balances[a]
 $\Sigma$  a:address new.balances[a]
```



< >

INVARIANT

```
total =  $\Sigma$  a: address balances[a]
```



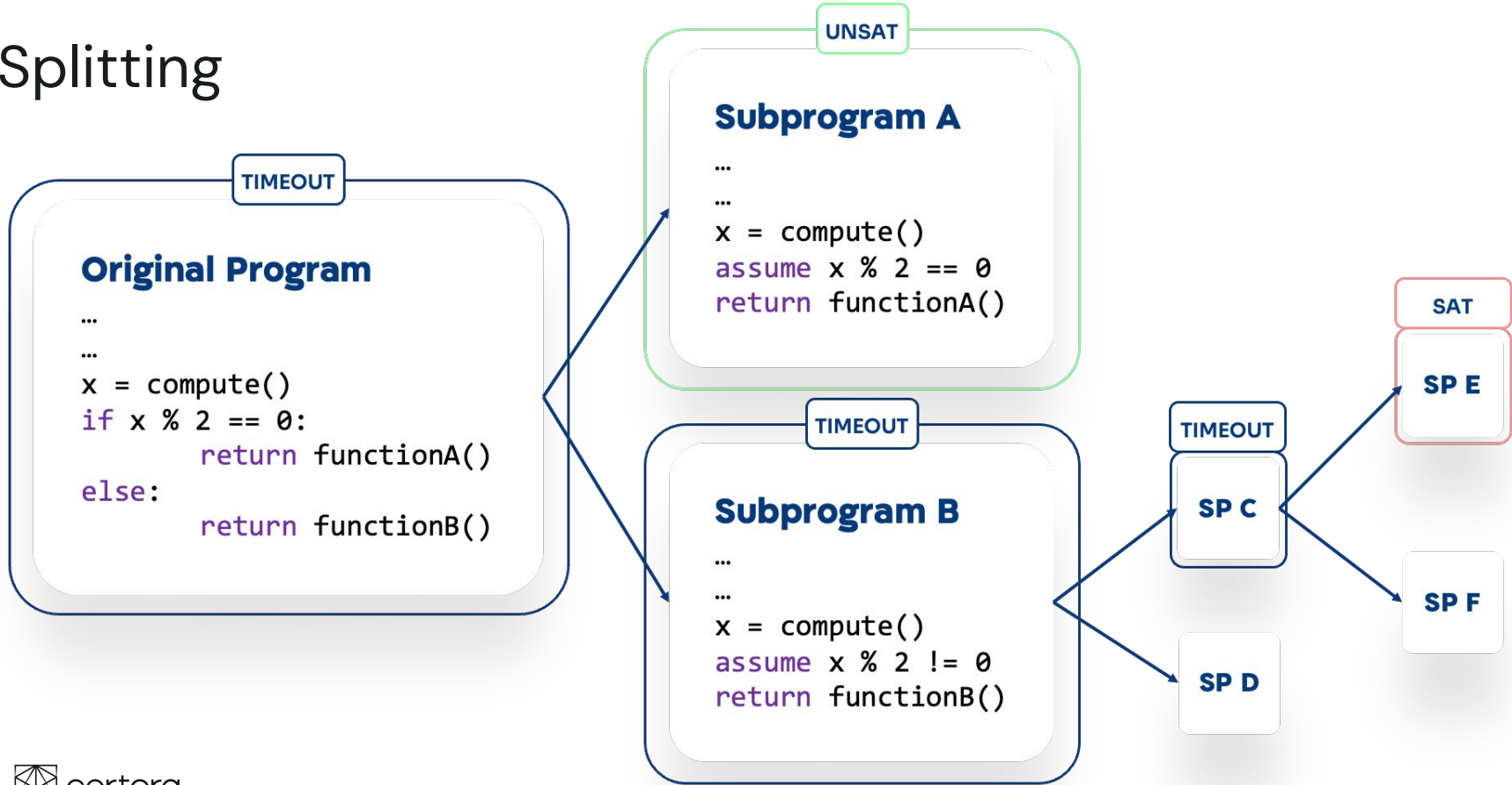
Design Decisions

- Bytecode analysis => Arbitrary code but harder
 - Static analysis recovers compiled code properties and identifies compiler errors
- Inline loops and procedures unless specified otherwise
- Heavily rely on static analysis
- Abstractions
 - Built in
 - Linear over-approximation
 - User defined

Certora Prover Scaling Techniques

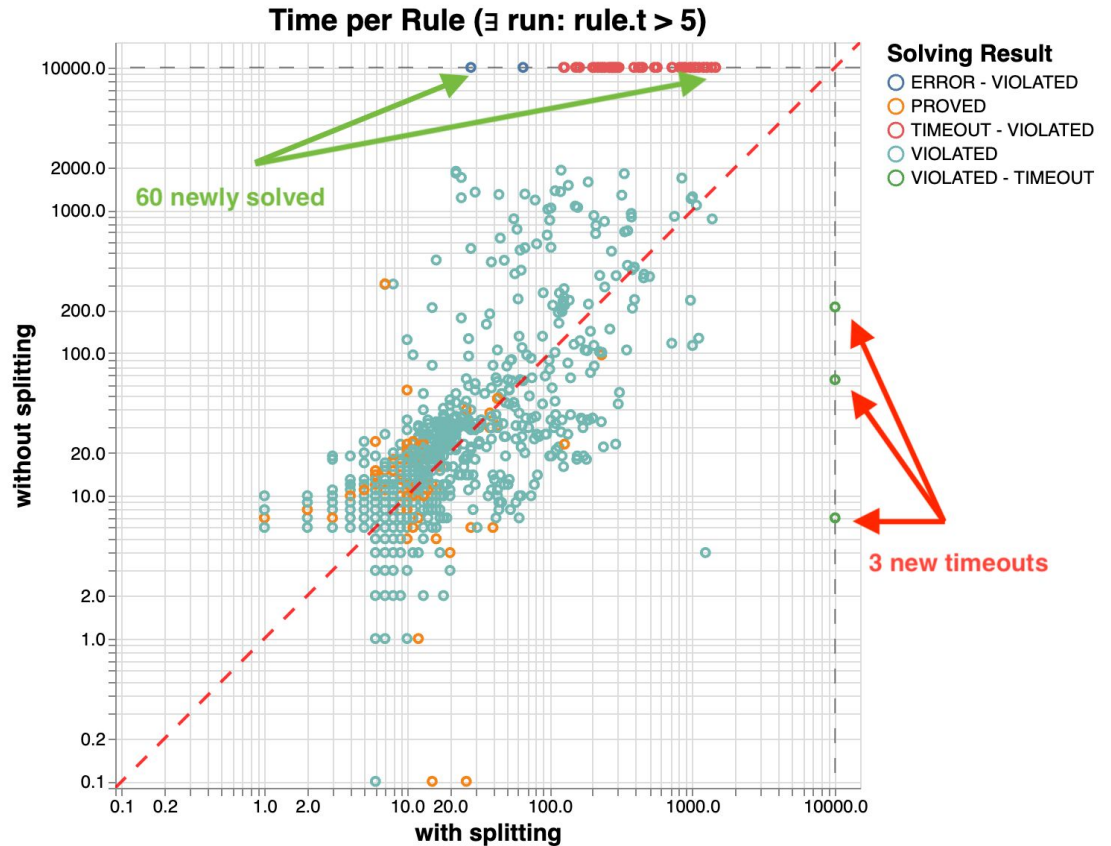
- Static analysis (OOPSLA '2024), e.g.:
 - Memory analysis
 - Pattern rewriter
- Splitting control flow paths
- Parallelisation
- Intervals Rewriter
- Over approximation
 - Uninterpreted nonlinear abstractions with axioms
- Quantifiers grounding

Splitting



Splitting

Experimental Evaluation



Parallelisation – Two Main Approaches

Sequential Splitter:

- One split at a time
- Large parallel smt solvers portfolio
 - Z3, cvc5, cvc4, smtinterpol, bitwuzla, yices, ...
 - Various solver configurations & random seeds
 - Various encodings (LIA, NIA, BV)

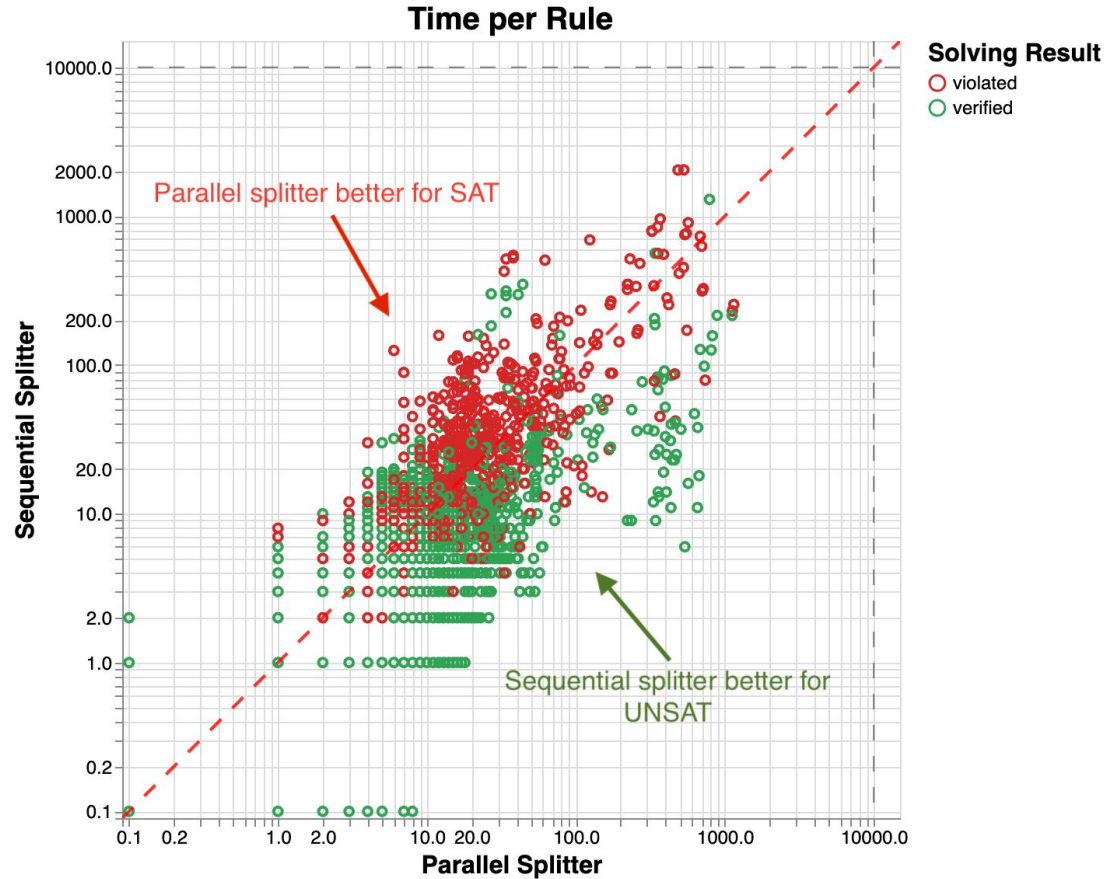
Parallel Splitter:

- Many splits in parallel
- Small solvers portfolio per split

Sidenote: we parallelize also solving of individual rules

Parallel Splitter

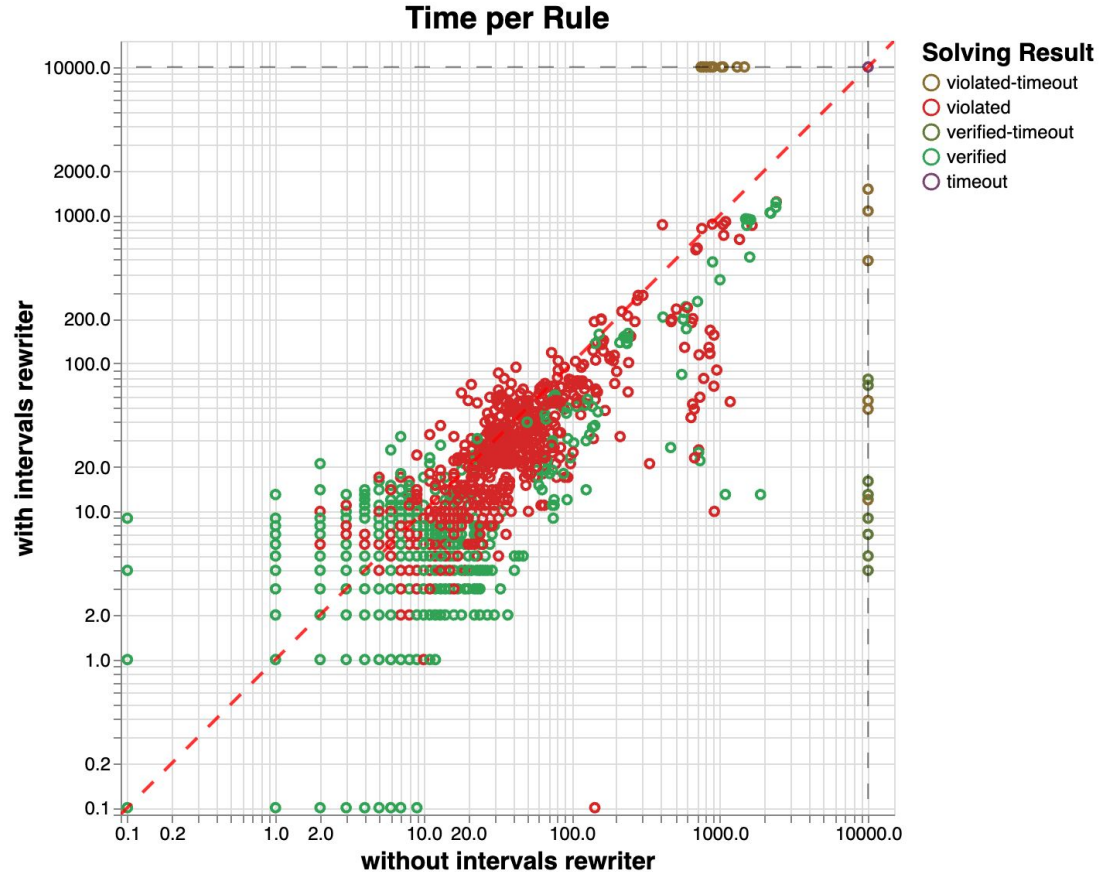
Experimental Evaluation



Interval Rewriter

- Similar to constant propagation, but propagating intervals of possible values of Boolean and numeric expressions
- Both forward and backward propagation, possibly fixpoint
- Remove branches that are unreachable or irrelevant
- Remove/simplify commands that are redundant
- Add explicit assumes to the program about the deduced intervals
- The SMT solvers also do this, so why should we?
 - We see the program structure

Interval Rewriter Experimental Evaluation



Introduction to AMMs

Automatic Market Makers (AMMs)

- ◆ A decentralized source of crypto token exchange between every two parties in the blockchain
- ◆ Assets (tokens) are provided by any actor (liquidity providers), and are aggregated in smart contracts which are usually called **pools**
- ◆ Aggregation of tokens leads to better capital efficiency – the more there are, the easier it is to leverage oneself (avoid price slippage)

AMMs(DeFi) vs. Traditional Exchanges

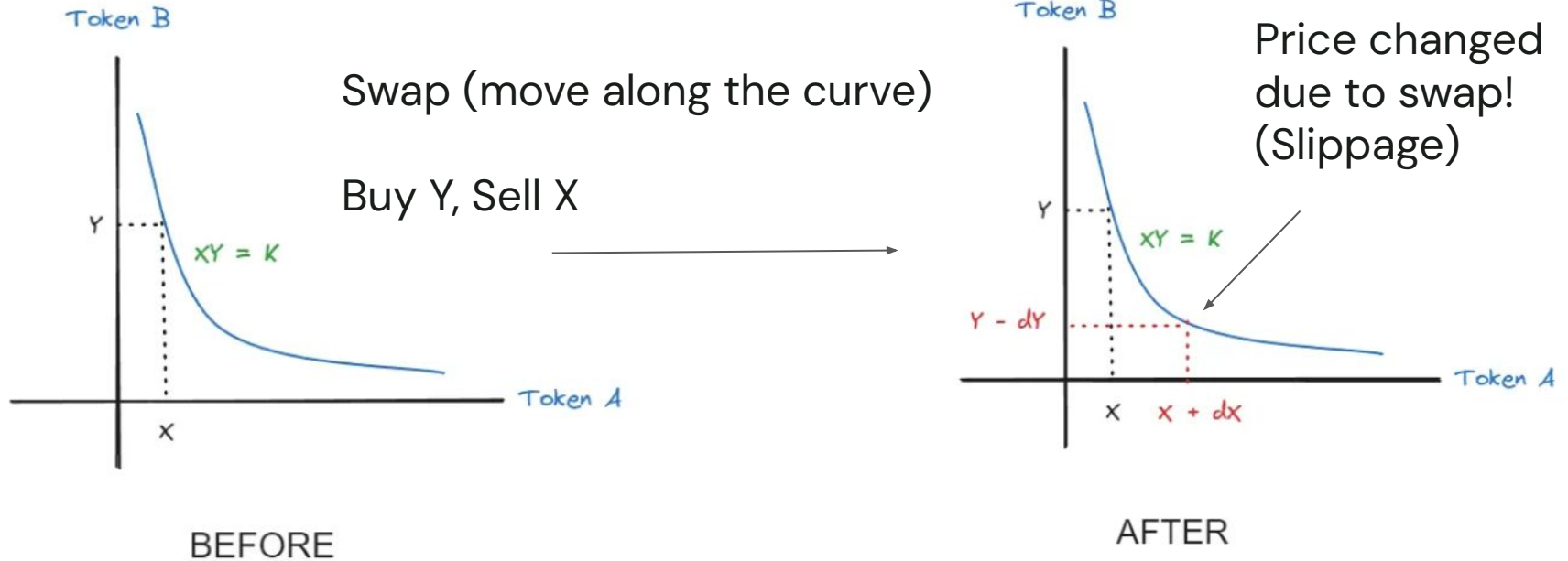
- Usually exchanges work according to the ledger model
 - ◻ Buyers and sellers open positions, and the ledger aims to match between the two parties
 - ◻ Highly inefficient in a decentralized fast environment like a blockchain
- Exchange dynamics (transactions) change the price based on supply and demand
- AMMs tend to reflect supply and demand of the tokens in the pool in the pricing mechanism

Example – constant product pool

- Very simple (but ubiquitous) type of an AMM is a constant product pool
- A single pool has two types of tokens (e.g. ETH and USDC*) with reserves for each, which are the sum of all tokens supplied to the pool.
- X – total ETH, Y – total USDC
- Every swap (tokens exchange) must maintain an invariant:
 - $X * Y = K = L^2$ (pricing curve)
 - where L is a measure of the total liquidity of the pool.
- The price, at any point along the curve, is determined by the ratio of tokens:
- $P(\text{buy USD, sell ETH}) = X / Y$

* USDC is the crypto stable coin of USD, being equal in worth +- 0.1%.


Example – constant product pool (cont)



Example – constant product pool (cont)

- Buying more of token X increases its price in the pool (less reserves – more demand)
- The more liquidity (reserves) there is, the more tokens it takes to swap to change the price
- Liquidity providers (LPs) earn swap fees (economical incentive to lock tokens)
- Any Pool should always hold enough reserves to pay back to all of its LPs (Pool solvency – everybody wants to withdraw)

Top DeFi AMMs

	 UNISWAP	\$5B		 Balancer	\$762M
	 PancakeSwap	\$1.8B		 Sushi	\$184M
	 Curve	\$1.8B			

Harnessing SMT Solvers for Reasoning about AMMs

Thinking of Properties

- Start with formulating properties that characterize the system under inspection
- A violation of a fundamental property indicates the existence of a bug
- A proof of such property (for every state and every possible operation) is the ultimate goal – the protocol is (relatively) safe.
- Assumptions have to be made (context, token balance limits, etc.)
- A property is eventually converted into several rules, to be implemented in CVL

Properties of AMMs

- It's impossible to buy tokens from a swap() operation without selling any (no free tokens!)
- No one can **earn** from immediate inverse operations (deposit-withdraw, swap back and forth)
 - No benefits to the issuer from sequences of transactions

What happens in the code?

- Usually every operation inside the Pool (the program) involves mathematical calculations, based on the implementation of the pricing curve
- Almost always **nonlinear**
- In Solidity (Ethereum), only **integer arithmetic** is used (division rounding errors)
- For example:
 - ⬡ $dy = \text{Buy}(dx)$ – how many tokens one has to sell (Y) in order to buy (X) of the other.
 - ⬡ $dL = \text{liquidity}(dX, dY)$ – how many liquidity points one achieves by supplying tokens to the pool.
- And these operations are the ones we formally test using our rules!

Timeouts

- If the rule is rather complex (many storage state updates, multiple function calls), we might encounter a solver timeout
- This means we have no final result whether the rule is correct or not
- But the problem is that even if we reduce the rule to a minimal test, we might still get timeouts because the underlying math is highly nonlinear – pain point of SMT solvers

Approaches for resolving timeouts

- Modular proofs – smaller steps, less code / paths involved in each rule
- Code summarization
- Over approximations (Linear Integer Arithmetic over-approximation; **LIA**)
 - Builtin
 - Abstracting solidity functions
- Modulate solver settings
(splitting depth, increase time per sub-program, custom portfolio)

Resolving timeouts – code summarization

- Code summarization replaces actual code (**functionality**) with uninterpreted functions (**behavior / axioms**)
- Relies on the idea that in order to prove a claim, one needs only some relation between input and output
- All other aspects of the function are useless for the proof and thus could be “skipped”
- The summarized function is replaced by an uninterpreted function call
- Saves SMT solving – better performance**
- We employ this methodology to **pure** functions

Example: tick to price conversions (Uniswap)



getSqrtPriceAtTick:

$$\sqrt{1.0001}^{tick}$$

- Inline assembly
- Bitvector operations with 256 bits
- Many multiplications
- Fixed-point arithmetic

```
function getSqrtPriceAtTick(int24 tick) internal pure returns (uint160 sqrtPriceX96) {
    unchecked {
        uint256 absTick;
        uint256 price;
        assembly {
            let mask := sar(255, tick)
            absTick := xor(mask, add(mask, tick))
            if gt(absTick, MAX_TICK) {
                mstore(0, 0xce8ef7fc)
                revert(0x1c, 0x04)
            }
            price := xor(shl(128, 1), mul(xor(shl(128, 1), 0xffffcb933bd6fad37aa2d162d1a594001), and(absTick, 0x1)))
        }
        if (absTick & 0x2 != 0) price = (price * 0xffff97272373d413259a46990580e213a) >> 128;
        if (absTick & 0x4 != 0) price = (price * 0xffff2e50f5f656932ef12357cf3c7fdcc) >> 128;
        ... // 15 similar lines removed to fit on slide
        if (absTick & 0x40000 != 0) price = (price * 0x2216e584f5fa1ea926041bedfe98) >> 128;
        if (absTick & 0x80000 != 0) price = (price * 0x48a170391f7dc42444e8fa2) >> 128;
        assembly {
            if sgt(tick, 0) { price := div(not(0), price) }
            sqrtPriceX96 := shr(32, add(price, sub(shl(32, 1), 1)))
        }
    }
}
```



Code summarization – example

- In UniswapV3/V4, the price range is made discrete according to ticks:
 - Square root of price(tick) = (as 64+96 bit fixed-point number)
- The code introduces two functions to convert one to the other:
 - function `getSqrtPriceAtTick(int24 tick)` internal returns (uint160) – **exponent**
 - function `getTickAtSqrtPrice(uint160 sqrtPriceX96)` internal returns (int24) – **logarithm**
 - Both are highly nonlinear– nightmare for solvers
- Our summary – uninterpreted functions with axioms
 - function `getSqrtPriceAtTickCVL(int24 tick)` internal returns (uint160)
 - function `getTickAtSqrtPriceCVL(uint160 sqrtPriceX96)` internal returns (int24)
 - Monotonicity: $x < y \Rightarrow \text{getSqrtPriceAtTickCVL}(x) < \text{getSqrtPriceAtTickCVL}(y)$
 - Inverse: $\text{getTickAtSqrtPriceCVL}(x) = \max\{\text{tick} \mid \text{getSqrtPriceAtTickCVL}(\text{tick}) \leq x\}$

Summarization Soundness

- Replacement of code with summary is **sound**, as long as the function satisfies the axioms **for every input**. We do not miss behavior, but rather generalize it – no path is excluded.
- Separately check
 - Overflow cases
 - Functions satisfy axioms
 - Code error / panic
- Upside: no bugs that were formerly present are excluded
- Downside: Potentially introducing spurious paths that don't exist in reality (false positives)

Summarization Procedure

- ◆ We look for “hard” functions in the code and summarize them with uninterpreted functions.
- ◆ We add axioms if needed, in conjunction to proving those for the underlying code (assert and assume).
- ◆ Continue until timeouts are resolved.
- ◆ If rules are proven – we are done, it means we proved it for the actual code too.
- ◆ If rules are violated – investigate the counterexample
 - If path is unfeasible – constrain the summary further
 - If path is feasible – **Possible bug!**

Speedups

	Original code*	With Summarization
Nonlinear operations count	89 - 97	3
Max polynomial degree	10	2
RULE #1		
Result - Time	UNSAT (Z3) - 2m 41s	UNSAT (Z3) - 1m 49s
RULE #2		
Result - Time	TIMEOUT (6% UNSAT)	UNSAT (several) - 35m 54s (100% UNSAT)
RULE #3		
Result - Time	UNSAT (Z3, CVC5) - 16m 50s	UNSAT (Z3, CVC4) - 3m



*we still summarize basic operations like $mulDiv(x,y,z) = x*y/z$ for reasonable performance.

Some Recent Success Stories

1. Formally [verified the Holy Grail](#) of Euler V2
2. Lido Dual Governance mechanism
3. External formal specification competitions with many participants writing formal specifications
 - a. > 100 participants
 - b. > 5000 rules
 - c. > \$500,000 rewarded
 - d. Follow us on twitter [@certorainc](#)

Challenges Prover

1. Specs are hard
 - a. Mutation testing and Unsat Cores help
2. Unpredictability of SMT solvers
3. Ease of use
 - a. Counterexample understanding
 - b. Timeouts
 - c. Summarization
 - d. Soundness
 - e. Static analysis robustness

Take away

1. Code is law is an interesting concept
2. SMT solvers can be used to formally verify high-level properties by software developers
3. Bugs are very useful to initially explain the value of FV
4. But Proofs is the only way to make FV standard

Come Join Us make FV standard in DeFi and beyond

1. Senior compiler writer
2. FV wizards
3. FV evangelists
4. FV and code security content creators
5. Sabbaticals
6. Competition participants
 - a. Solidity and Rust
7. No interns

Acknowledgements

The Certora Team



The Uniswap Team



The SMT community



Andrew
Reynolds



Nikolaj Bjorner

Thank You

