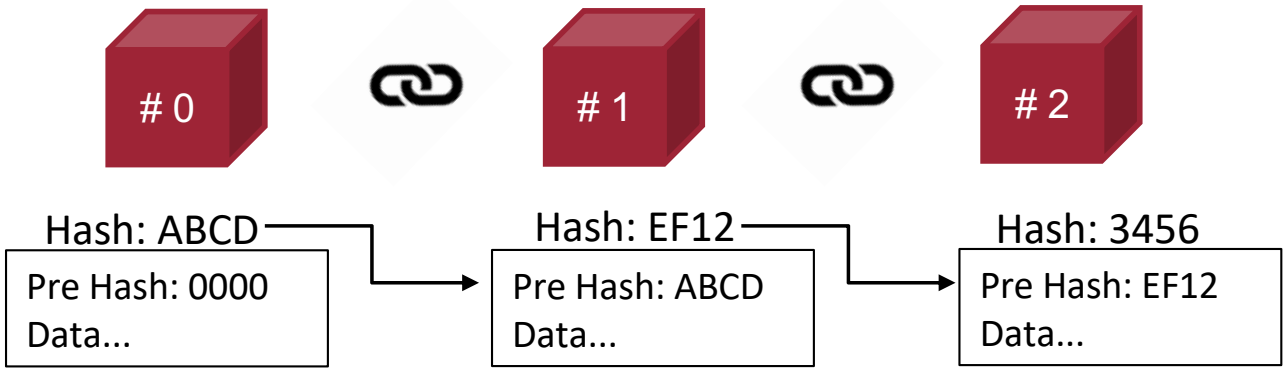
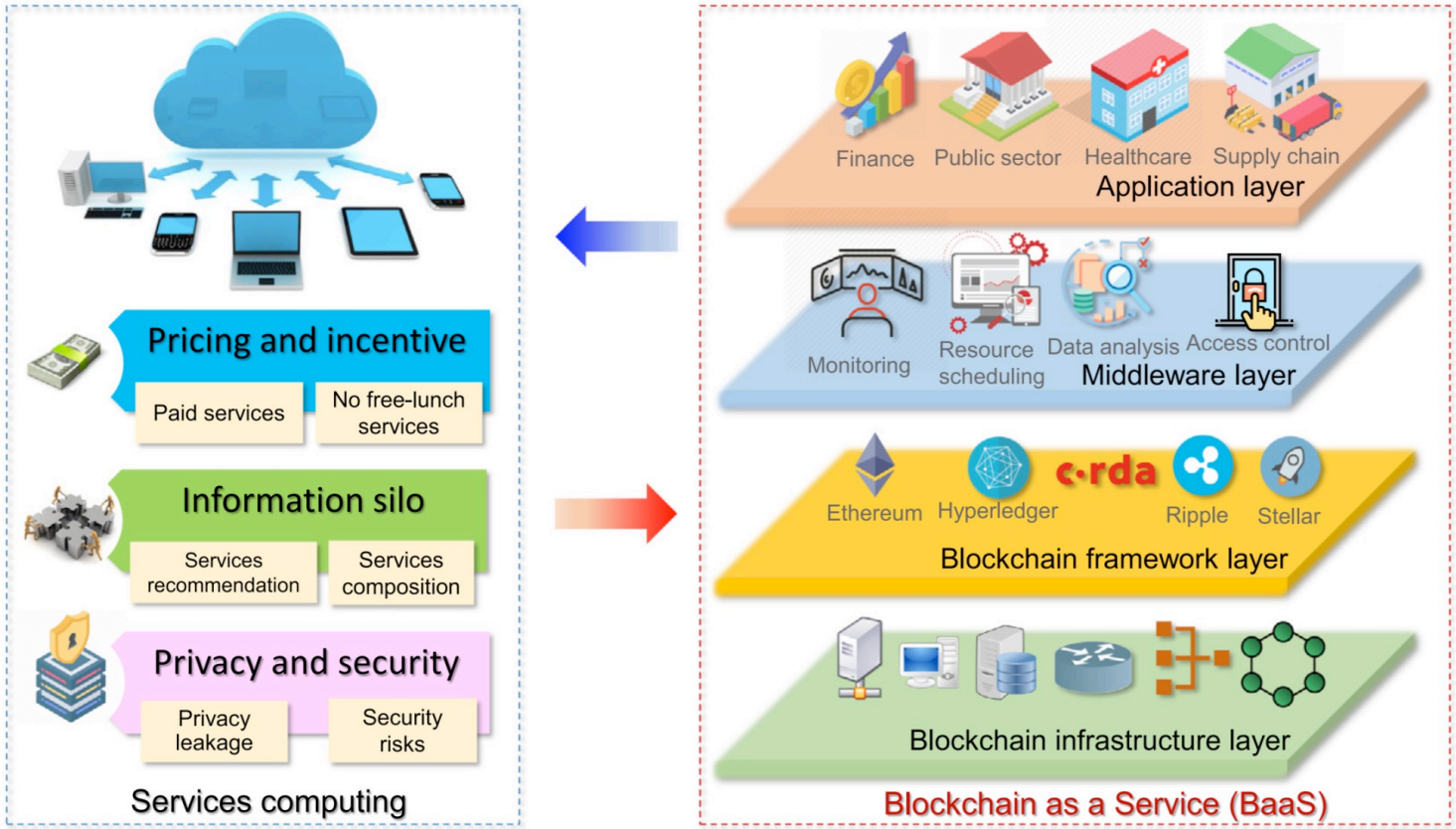
Blockchain Architecture

- Block
  - previous hash, data, hash
- Data
  - i.e., address, balance



Blockchain Applications



Xiaoyun Li, Zibin Zheng, Hong-Ning Dai, ”When services computing meets blockchain: Challenges and opportunities”, Journal of Parallel and Distributed Computing, Volume 150, April 2021.

Blockchain TVL

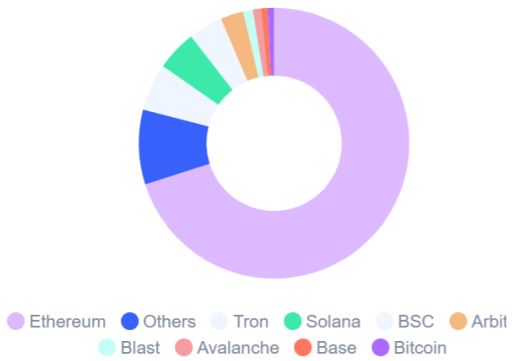
- TVL (Total Value Locked) of all blockchains reaches $170B.

#	Name	Protocols	1d change (TVL)	1w change (TVL)	1m change (TVL)	TVL	Mcap	Mcap/TVL
1	Ethereum ETH	1096	▲ 0.77%	▼ 2.00%	▼ 7.89%	$64.12B	$403.48B	3.33
2	Tron TRON	50	▲ 0.14%	▼ 2.60%	▼ 0.93%	$7.75B	$10.51B	1.08
3	BSC BNB	754	▲ 0.21%	▼ 2.92%	▲ 10.39%	$4.1B	$87.92B	12.41
4	Solana SOL	151	▲ 0.48%	▼ 7.26%	▲ 49.38%	$3.01B	$79.3B	9.49
5	Arbitrum ARB	606	▲ 0.37%	▼ 0.30%	▲ 2.68%	$2.45B	$3.95B	0.84

https://coinmarketcap.com/chain-ranking/

Ethereum TVL

- Ethereum is one of the most popular blockchain.
- $52.86 billion TVL



TVL by Chains

Ethereum TVL trend

TVL source: https://defillama.com/chain/Ethereum

What Are Smart Contracts?

computer programs running on blockchain

- Ethereum smart contracts are typically written in Solidity and then compiled to bytecode, running on EVM.

```
uint y; // state variable
// public method x2
function vul(uint x) public {
  if(x < 2 && y == 4)
    assert(0);
}
function dep(uint i) public
  { y = i; }
```

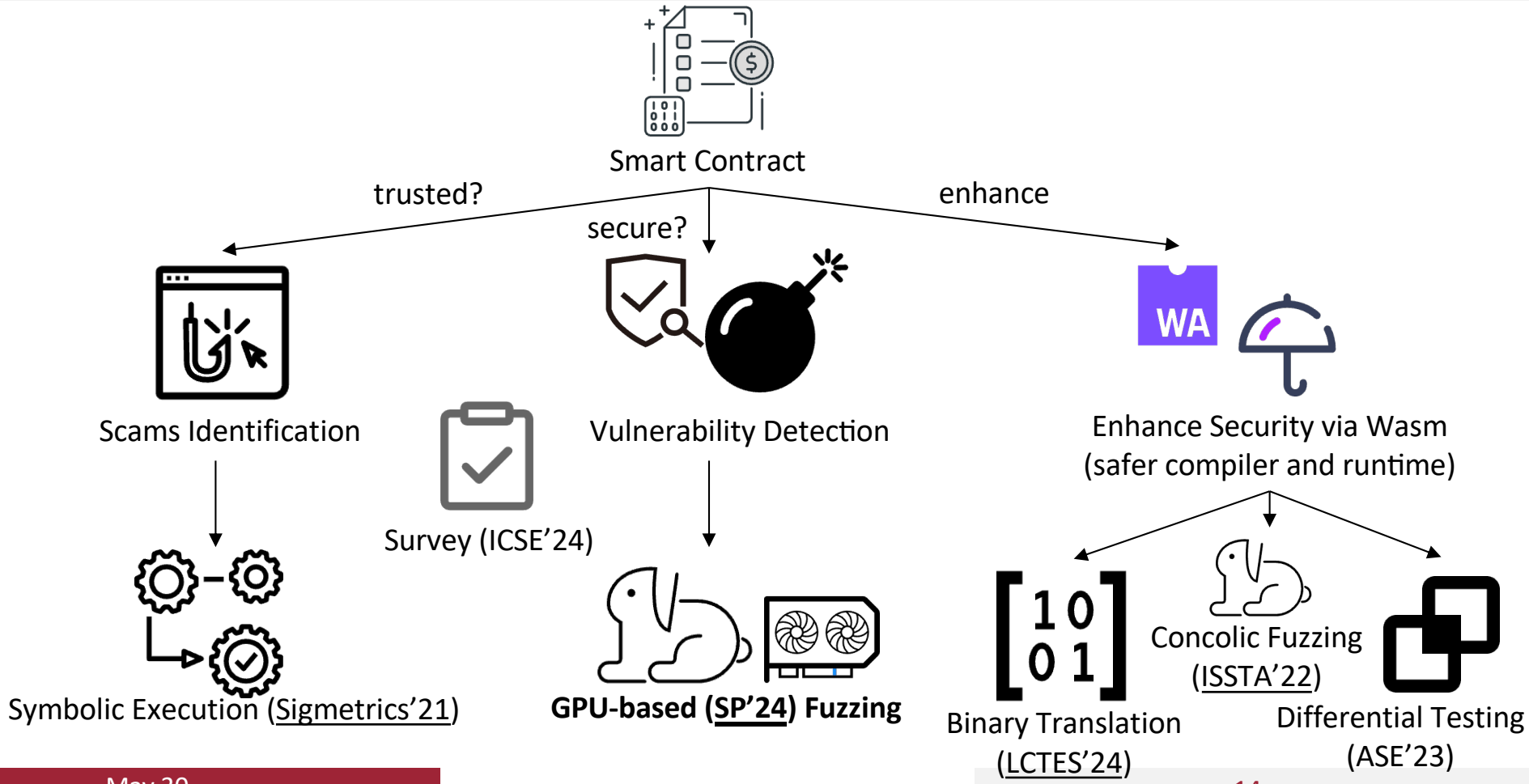
source code

```
PUSH 0xc3bae50d
CALLDATA(0)
EQ BNE // vul(uint x)
PUSH 0x71840a0d
CALLDATA(0)
EQ BNE // dep(bool i)
...
CALLDATA(4)
PUSH 4
EQ
... RETURN
```

EVM bytecode

The EVM bytecode (right side) is compiled from the source code in Solidity (left side).

Road Map


Smart Contract
trusted?
secure?
enhance
Scams Identification
Survey (ICSE'24)
Vulnerability Detection
WA
Enhance Security via Wasm
(safer compiler and runtime)
Symbolic Execution (Sigmetrics'21)
GPU-based (SP'24) Fuzzing
Binary Translation
(LCTES'24)
Concolic Fuzzing
(ISSTA'22)
Differential Testing
(ASE'23)

Are Smart Contracts Trusted?

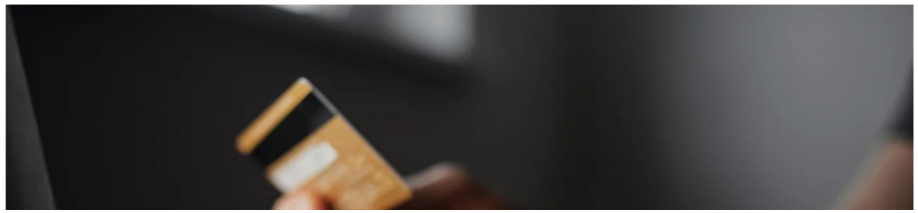
SADPonzi: Detecting and Characterizing Ponzi Schemes in Ethereum Smart Contracts, Sigmetrics’21

Can We Trust Smart Contracts?

- **No!**
- Billions of USD lost due to smart contract scams.
  - Ponzi, Rug Pull, Phishing, and Honeypot

FBI says $10 billion lost to online fraud in 2022 as crypto investment scams surged

By Sean Lyngaas and Hannah Rabinowitz, CNN
2 minute read · Published 9:49 PM EDT, Mon March 13, 2023



https://edition.cnn.com/2023/03/13/politics/fbi-online-fraud-report/index.html

Losses from cryptocurrency scams double to HK$1.7 billion in Hong Kong last year, with 67 per cent spike in cases

- According to police, 2,336 cryptocurrency-related scams reported in Hong Kong last year
- One woman vice-president of fashion company duped out of digital tokens worth HK$6 million trying to buy kitten

Clifford Lo + FOLLOW
Published: 7:00am, 7 Mar, 2023
Why you can trust SCMP

Police say cryptocurrency-linked scams made up more than half of the HK$3.2 billion lost in technology crimes in Hong Kong last year.
Photo: Getty Images

https://www.scmp.com/news/hong-kong/law-and-crime/article/3212575/losses-cryptocurrency-scams-double-hk17-billion-hong-kong-last-year-67-cent-spike-cases

Advantages Be Taken by Scam Initiators

- Anonymous
  - No one can track the identity.
  - The scam initiator can stay anonymous.
- Enforce
  - No one can stop the execution of smart contracts.
- Lose
  - No one can deny the transactions.
  - The victim’s money cannot be found back.

Ponzi Scam in Blockchain

coindesk

Inside a Crypto 'Ponzi': How the $6.5M Ban
Unravelled

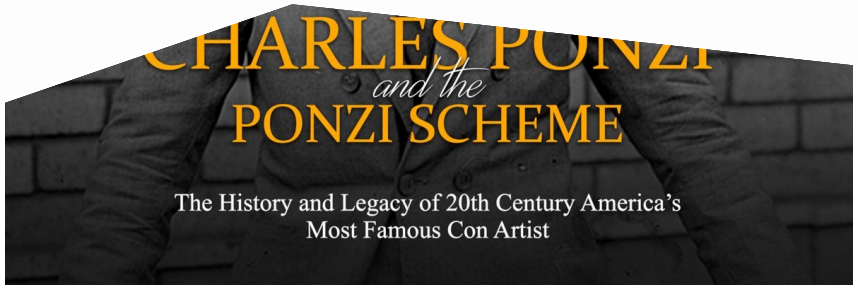
Ponzi schemes

Decrypt

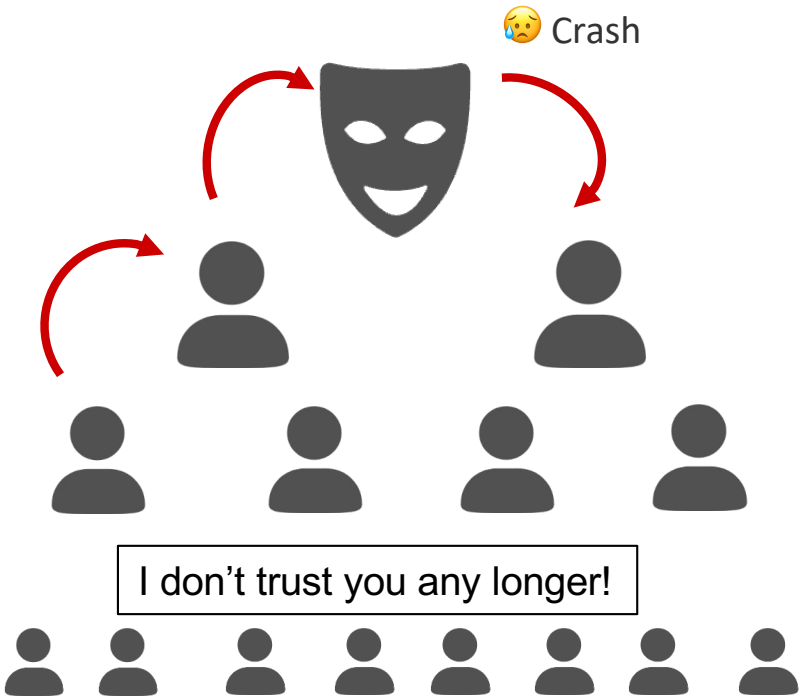
Fairwin: The $125 million alleged Ponzi scheme
eating Ethereum

Blockchain

Millions of p
in 2019



Ponzi Scams



1. Promise the investors a high-yield return
2. Continue the scam by using part of new investment to reward to earlier investors.
3. In return, the rewarded investors would help scam attract new investors.
4. Crash: when there are no enough new investments

Existing Works for Ponzi Scams

- M. Vasek, et. al presented the first empirical analysis of Bitcoin scams [ICFCDS].

- TxML: ML + transaction behaviors features [TxML]
  - Poor scalability.
  - Only popular scams that have lured a number of victims can be identified.
- OPCodeML: ML + instruction frequency [OPCodeML]
  - Lack of interpretability.
  - Be prone to evasion techniques.

[ICFCDS'15] Marie Vasek and Tyler Moore. There's no free lunch, even using bitcoin: Tracking the popularity and profits of virtual currency scams. In International conference on financial cryptography and data security, pages 44–61. Springer, 2015.
[TXML] Eunjin Jung, Marion Le Tilly, Ashish Gehani, and Yunjie Ge. Data mining-based ethereum fraud detection. In 2019 IEEE International Conference on Blockchain (Blockchain), pages 266–273. IEEE, 2019
[OPCodeML] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In Proceedings of World Wide Web Conference, 2018.

Ponzi Scam Classification

```
contract HandoverPonzi {
address public throne ;
uint public price = 1 ether;
uint fee = 0;
 function(){//fallback function
  if(msg.value < price) throw;
  fee += msg.value * 0.1;
  throne.transfer(msg.value*0.9);//reward
  throne = msg.sender;//invest
  price = price * 2;
}
```

Handover-scheme

```
contract ChainPonzi {
  uint balance = 0;
  uint Cursor = 0;
  struct Participant {
    address etherAddress;
    uint payout;
  }
  Participant[] participants;
  function() {
    participants.push(Participant
    (msg.sender, msg.value* 3));
    balance += msg.value ;
    while (balance > participants[Cursor].
                                  payout) {
      uint payoutToSend = participants
                    [Cursor].payout;
      participants[Cursor].etherAddress
            .send(payoutToSend);
      balance = participants
                    [Cursor].payout;
      Cursor += 1;
    }
  }
}
```

Chain-scheme

```
contract TreePonzi {
  struct Participant {
    address inviter;
    address herself;
  }
  mapping(address => Participant) Tree;
  address top = 0xffff..ffff;
  function enter(address inviter) public {
    uint amount = msg.value;
    Tree[msg.sender] = Participant(
          {herself: msg.sender,
          inviter: inviter});
    address next = inviter;
    while (next != top) {
      amount /= 2;
      next.send(amount);
      next = Tree[next].inviter;
    }
  }
}
```

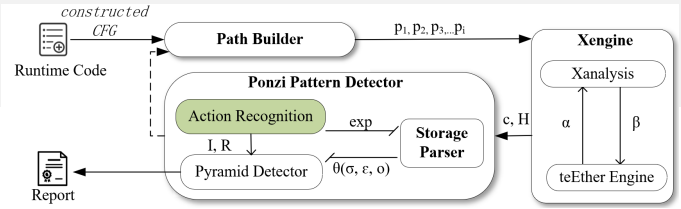
Tree-scheme

```
contract PonziICO {
  uint public total;
  mapping (address => uint)
                public invested;
  mapping (address => uint)
                public balanceOf;
  address[] investors;
  address owner = 0x4f22...1e;
  function withdraw() public{
    uint amount = balanceOf[msg.sender];
    balanceOf[msg.sender] = 0;
    msg.sender.transfer(amount);
  }

  function invest() private{
    uint dividend = msg.value;
    for(uint i=0;i<investors.length;i++) {
        uint amount = dividend * invested[in
                    vestors[i]] / total;
        balanceOf[investors[i]] += amount;
    }
    investors.push(msg.sender);
    invested[msg.sender] = msg.value;
    total += msg.value;
  }
  function() payable{ invest(); }
}
```

Withdraw-scheme

Action Recognition



- Investing action

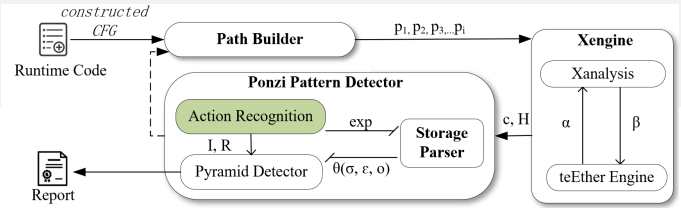
$$i_r = \mathsf{SSTORE}_1(_, \tau_1) \wedge is_r(\tau_1) \wedge i_v = \mathsf{SSTORE}_2(_, \tau_2) \wedge is_v(\tau_2)$$
$$\wedge\ (\{\mu = S[*][8 * i : 160 + 8 * i] \mid i \in \mathbb{N}, i \leq 12\}) \nsubseteq c \wedge ((\lambda > 0 \in c) \vee f \notin sigs)$$

- Rewarding action

$$i = \mathsf{CALL}(_, t, b) \wedge ((t = I_r) \vee ((t = \mu) \wedge (b = I_v)))$$

Refer to our paper for technical details.

Action Recognition



- Handover-scheme

$$tag(I_r) = tag_1 \wedge I_r = R_t \wedge bb(I_{i_r}) \in dcs(bb(R_i))$$

- Chain-scheme

$$tag(I_r) = tag_2 \wedge I_r = R_t$$
$$\wedge\ (tag(R_t^{key}) = tag_1 \vee (type(R_t^{key}) = \text{integer} \wedge is_cycle(\mathcal{G}, bb(R_i))))$$

- Tree-scheme

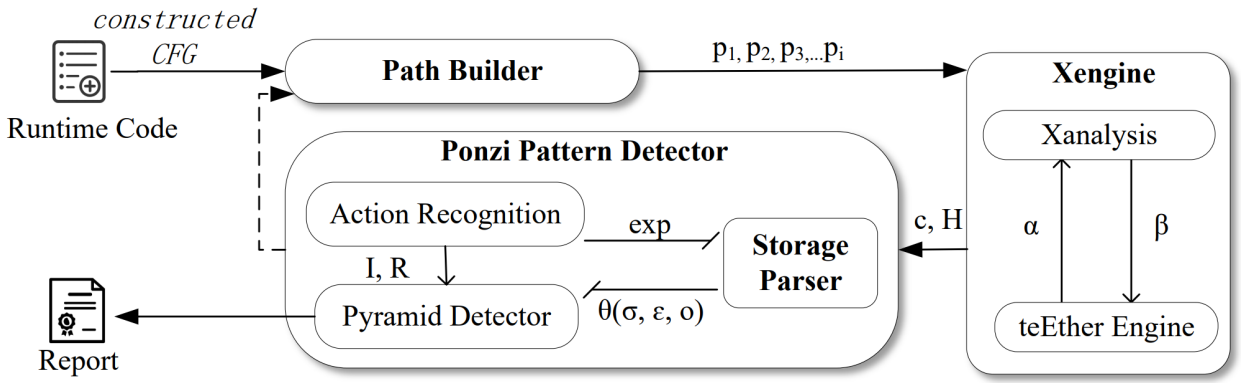
$$tag(I_r) = tag_3 \wedge I_r = R_t \wedge I_r = \theta_{I_r}(_, _, 160)$$

- Withdraw-scheme

$$R_r = \mu \wedge tag(R_b) = tag_2 \wedge I_v = R_b \wedge is_cycle(\mathcal{G}, bb(I_{i_v}))$$

Refer to our paper for technical details.

SADPonzi: The First Symbolic Execution Based Detector for Ponzi Scams in Ethereum (ACM Sigmetrics'21)



The overall architecture of SADPonzi

- Path Builder: generate paths of investing actions and reward actions
- Xengine: build a symbolic context for each path (Z3 SMT)
- Ponzi Pattern Detector: Identify and classify Ponzi schemes

Experiments for SADPonzi

- Benchmark
  - 1,262 non-Ponzi samples from DAppTotal.com
  - 133 Ponzi contracts remarked from M. Bartoletti
- Environment
  - Ubutnu 18.04 with i9-9900 CPU and 64 GB RAM
  - 20 minutes timeout and 30 maximum call depth
- Baselines
  - TxML
  - OPCodeML

Scam Detection Ability of SADPonzi

Overall evaluation results.

Approach	Precision	Recall	F1-measure
TxML [42]	76.7%	57.9%	66.0%
OPCodeML [32]	92.0%	85.2%	88.5%
SADPonzi	**100%**	**100%**	**100%**

- The result of baselines is inline with the evaluation in previous work.
- SADPonzi outperforms existing ML approaches*.

The Prevalence of Ponzi Smart Contracts

- Unique Ponzi smart contracts
  - #616 (extended to 835)
  - 56.05% (468/835) are Chain-scheme
- Txs
  - Over 598.1 K txs records (55% are incoming txs)
- Lose
  - 17 K victims have sent over 100 K ETH

The transactions related to Ponzi Scams.

Type	#Distinct Num	#Extend Num	Investment			
			# Victim	# Transaction	# Ether	USD
Chain	344	468	3,120	227,649	47,228.47	8,330,629.8233
Tree	212	268	14,565	99,612	52,073.8385	9,185,304.3730
Withdraw	13	19	174	824	463.8878	81,825.1690
Handover	47	80	148	2,603	624.6784	110,187.0230
Total	**616**	**835**	**17,870**	**330,688**	**100,390.87**	**17,707,946.38**

Takeaway

- Propose a semantic-aware approach for Ponzi smart contracts detection
- Develop SADPonzi working on Ethereum smart contract bytecode; Outperforms all ML baselines.
- Apply SADPonzi to all 3.4 million smart contracts on Ethereum
- Identify 835 Ponzi scams involving over $17 M

Are Smart Contracts Secure?

Towards Smart Contract Fuzzing on GPUs, S&P’24
Are We There Yet? Unraveling the State-of-the-Art Smart Contract Fuzzers, ICSE’24

Are Smart Contract Secure?

Not Secure Enough!

In 2016, DAO lost $70B [DAO]

In 2021, Poly Network lost $600M [Poly]

In 2024, Munchables lost $63M [NFT]

…

The DAO Attacked: Code Issue Leads to $60 Million Ether Theft

TheDAO, the largest and most visible ethereum project, has reportedly been hacked, sparking a broad market sell-off.

By Michael del Castillo Jun 17, 2016 at 9:00 p.m. Updated Sep 11, 2021 at 8:19 p.m.

The DAO, the distributed autonomous organization that had collected over $150m worth of the cryptocurrency ether, has reportedly been hacked, sparking a broad market sell-off.

A leaderless organization

comprised of a series of smart contracts written on the ethereum codebase, The DAO has lost 3.6m ether, which is currently sitting in a separate wallet after being split off into a separate grouping dubbed a "child DAO".

Ether markets plunged on the news, falling below $13 in trading on the cryptocurrency exchange Poloniex. With ether currently trading at roughly $17.50 per coin, that puts the value of the stolen cryptocurrency at more than $60m.

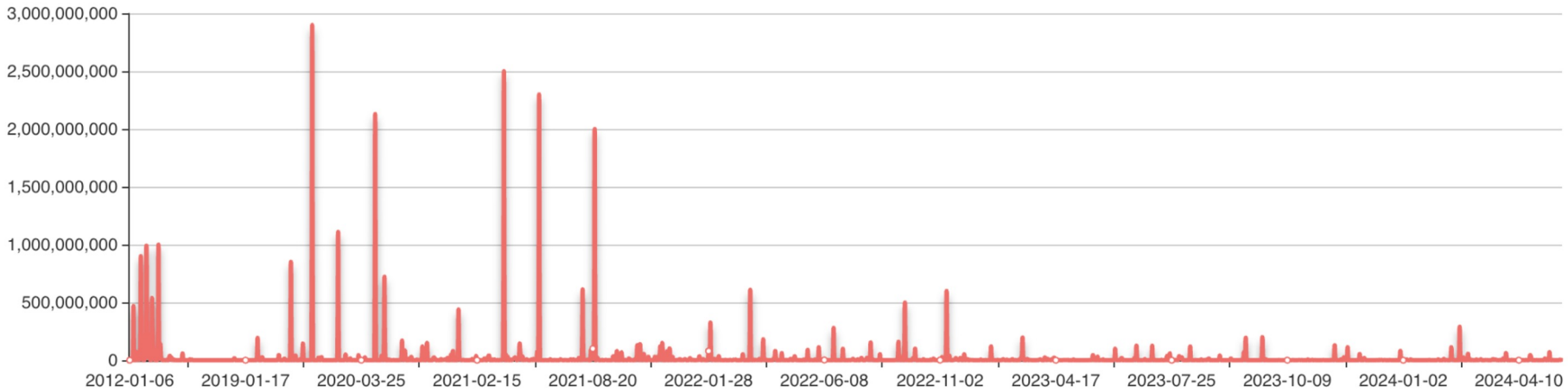
[DAO] https://www.coindesk.com/markets/2016/06/17/the-dao-attacked-code-issue-leads-to-60-million-ether-theft/
[Poly] https://decrypt.co/147059/poly-network-attack-conjures-billions-of-dollars-in-tokens-that-did-not-exist
[NFT] https://www.bloomberg.com/news/articles/2024-03-27/crypto-hit-by-63-million-munchables-game-hack-in-one-of-year-s-biggest-exploits

Financial Loss due to Vulnerabilities

[SlowMist Hacked Statistical]:

Total hack event(s) 1547 ;

The total amount of money lost by blockchain hackers is about $ 32,360,346,407.24 ;



https://hacked.slowmist.io/

Vulnerabilities Detectors

- Smart contracts bug detectors
  - Static tools
  - Oyente[CCS'16]
  - Verismart [SP'20]
  - SAILFISH [SP’22], ...
- Dynamic tools (i.e., Fuzzing)
  - Ityfuzz [ISSTA'23]
  - Smartian [ASE’21], ...

Unravelling Smart Contract Fuzzers (ICSE'24)

- Fuzzing technique provides invalid, unexpected, or random data as input to a target program in order to discover security vulnerabilities or bugs that could be exploited.

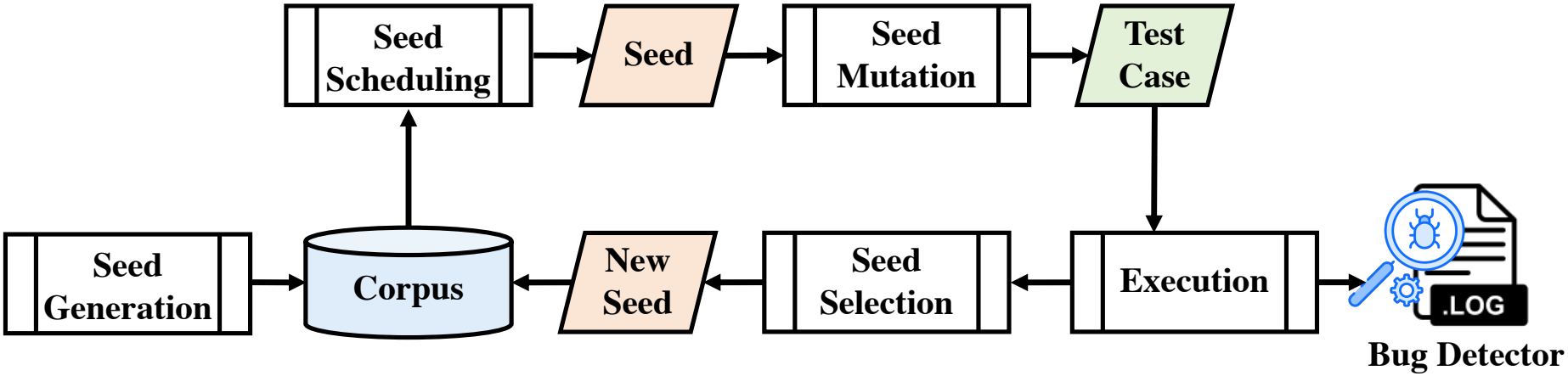


Table 1: Summary of technical details for existing smart contract fuzzers.

Tool			G/M	Arguments Generation						Sequence Generation				Seed Mutation			Seed Scheduling			SE
Name	year	OC		Type	Ran	Dict	Prev	ML	Sym	Ran	Dyn	Static	ML	Args	Env	Seq	Cov	Dis	Bug	
ContractFuzzer [55]	2018	✓	G	✓	✓	✓				✓										
Reguard [66]	2018		M		✓					✓				✓			✓			
ILF [48]	2019	✓	G	✓	✓			✓					✓	✓	✓					
SoliAudit [65]	2019	✓	G	✓	✓					✓				✓	✓					
ContraMaster [93]	2019	✓	M	✓	✓		✓			✓	✓			✓	✓	✓	✓			
Harvey [99]	2020		M		✓		✓			✓				✓		✓	✓	✓		
ETHPlOIT [103]	2020	✓	M	✓	✓		✓					✓		✓	✓		✓			
sFuzz [74]	2020	✓	M	✓	✓					✓				✓	✓		✓	✓		
Echidna [46]	2020	✓	M	✓	✓	✓	✓			✓				✓			✓			
GasFuzzer [22]	2020		M	✓	✓	✓				✓				✓						
Targy [53]	2021		M	✓	✓					✓					✓		✓	✓		
Smartian [40]	2021	✓	M	✓	✓				✓		✓	✓		✓	✓	✓	✓			
SmartGift [109]	2021	✓	G	✓	✓			✓		✓										
ConFuzzius [91]	2021	✓	M	✓	✓	✓			✓	✓	✓			✓	✓	✓	✓			
Beak [104]	2022		M	✓	✓	✓			✓			✓		✓	✓	✓		✓		
xFuzz [102]	2022	✓	M	✓	✓							✓	✓	✓	✓		✓	✓		
EtherFuzz [95]	2022		M	✓	✓	✓				✓				✓	✓	✓	✓			
SynTest-S [75]	2022	✓	M	✓	✓					✓				✓		✓		✓		
RLF [88]	2022	✓	G	✓	✓								✓	✓	✓		✓		✓	
effuzz [54]	2023		M	✓	✓					✓				✓	✓		✓	✓		
IR-fuzz [67]	2023	✓	M	✓	✓							✓		✓	✓	✓		✓		
EF\CF [83]	2023	✓	M	✓		✓				✓				✓	✓	✓	✓			✓
ityfuzz [87]	2023	✓	M	✓	✓					✓				✓	✓	✓	✓	✓		✓

In the **Tool** column, OC: open-sourced; In the **G/M** column, G: generation-based, M: mutation-based; In the **Arguments Generation** column, Type: type-awareness, Ran: randomly generated arguments, Dict: arguments from a pre-prepared dictionary, Prev: values previously encountered or used, ML: arguments obtained through machine learning, Sym: arguments obtained via symbolic execution

In the **Sequence Generation** column, Ran: randomly generated transaction sequences, Dyn: sequences generated through analyzing dynamic data flow, Static: sequences generated by analyzing static data flow, ML: sequences generated through machine learning; In the **Seed Mutation** column, Args: mutating transaction arguments, Env: mutating environment properties, Seq: mutating transaction sequences; In the **Seed Scheduling** column, Cov: fitness by coverage, Dis: fitness by distance, Bug: fitness by vulnerability; The **SE** column indicates whether a standalone EVM is used.

Empirical Evaluations

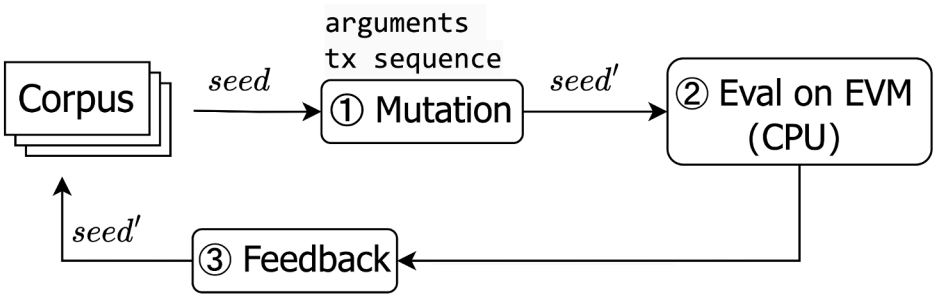
- Use 11 open-sourced fuzzers
- Benchmarks:
  - 2,000 vulnerable smart contracts from previous research datasets and third-party repositories
  - 9 vulnerability types

Findings

- Using standalone EVM (Ethereum Virtual Machine) can significantly improve the fuzzers' throughput.
- The speed of vulnerability detection is largely influenced by fuzzers' throughput.
- The fuzzers' throughput are pivotal for quickly reaching high coverage.

Typical Three Steps in Fuzzing

Fuzz = (①Mutation + ②Eval + ③**Feedback**)^n



A typical workflow of smart contract fuzzing.

Smart contract fuzzing aims at searching test cases (i.e., seeds) that a target smart contract can exercise to explore as many program states as possible.

① Mutation: mutate a seed from the corpus to derive a new one;

② Eval: execute the target smart contract in EVM to evaluate the mutated seed;

③ **Feedback**: **gather feedback metrics to assess the seed's value.**

Existing Fuzzers' Novelty

- Improving ③ feedback, such as
  - code coverage [ASE'21]
  - branch distance [sFuzz]
  - machine learning [ILF]
  - transaction dependency

    [ASE'21], [ConFuzzius], [ISSTA'23].

Comparison of existing smart contract fuzzers.

Tool	OC	Bytecode	Throughput	Novelty
Ityfuzz	✓	✓	33.0K	②③
Smartian	✓	✓	419	③
EF\CF	✓	✓	23.9K	②
sFuzz	✓	✓	512	③
ILF	✓	✓	57	①
Echidna	✓	✗	166	③
Harvey	✗	✗	-	①③
ConFuzzius	✓	✗	107	③

- Using smarter mutation strategies. [ILF] [Harvey]
- Few works focus on faster EVM execution. [EF\CF]

What Is The Major Bottleneck of Fuzzer's Throughput?

EVM Execution

- Results:
  - Ityfuzz and Smartian, spent 83% and 78% of the total time on EVM, respectively.
  - More complex => more time on EVM

Motivation: There are potential opportunities to improve throughput by optimizing ②, particularly when testing complex blockchain projects.

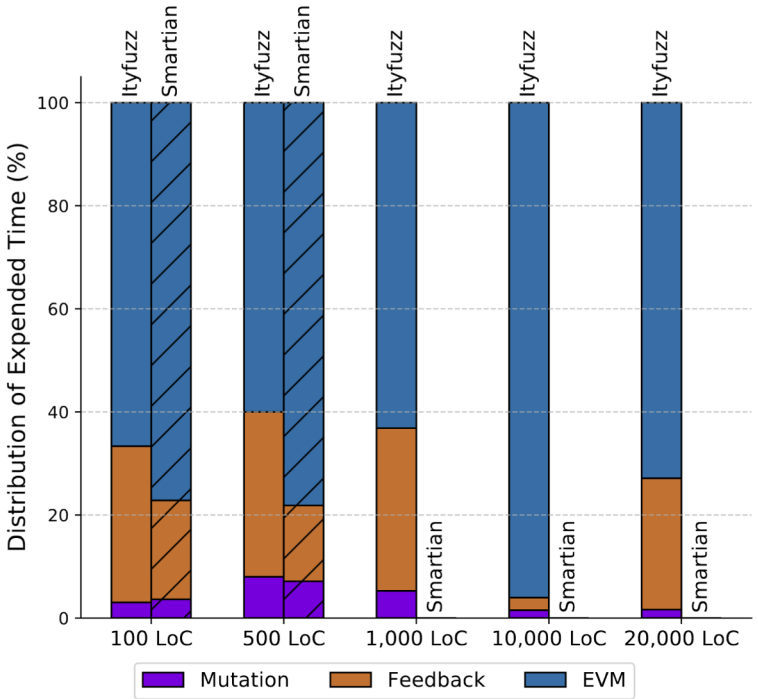


Figure 2. Distribution of time spent by Ityfuzz and Smartian across three typical steps in one fuzzing iteration.

Towards Smart Contracts Fuzzing on GPUs (S&P'24)

- **P1**: no heavy synchronization
  - Fuzzing threads execute the program independently with assigned seeds

Towards Smart Contracts Fuzzing on GPUs (S&P'24)

- **P1**: no heavy synchronization
- **P2**: branch divergence overhead
  - Only a few seeds can eventually hit new paths.
  - It is impossible to skip redundant seeds, because we do not know a seed is redundant until it has been evaluated.

Towards Smart Contracts Fuzzing on GPUs (S&P'24)

- **P1**: no heavy synchronization
- **P2**: branch divergence overhead
  - Only a few seeds can eventually hit new paths.
  - It is impossible to skip redundant seeds, because we do not know a seed is redundant until it has been evaluated.
  - GPU can boost the fuzzing because most seed evaluations involve similar paths.

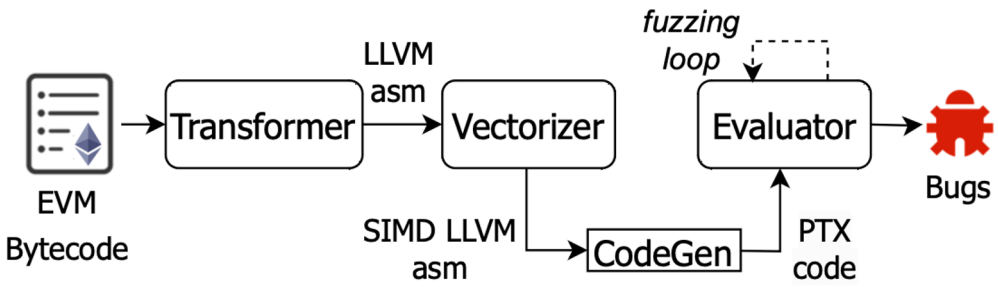
Towards Smart Contracts Fuzzing on GPUs (S&P'24)

- **P1**: no heavy synchronization
- **P2**: branch divergence overhead
- **P3**: smart contracts execution is serial
  - the execution of smart contracts is serial, aligning well with GPU-accelerated fuzzing

Towards Smart Contracts Fuzzing on GPUs (S&P'24)

- **P1**: no heavy synchronization
- **P2**: branch divergence overhead
- **P3**: smart contracts execution is serial
- **P4**: no system calls
  - system calls are not required for executing smart contracts.

Mau Design (S&P'24)



The overview of MAU.

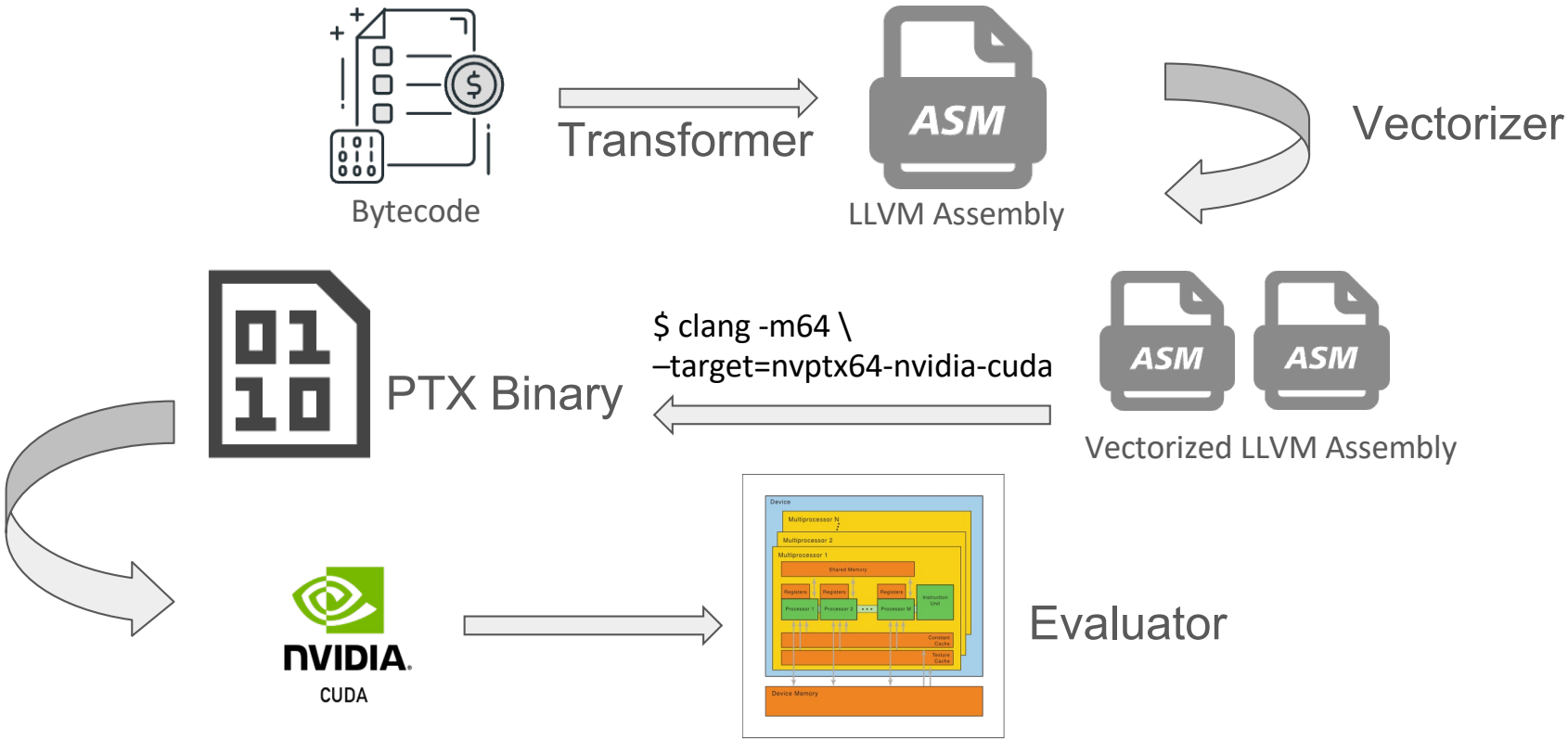
- **Transformer** lifts the EVM bytecode to an LLVM asm.

- **Vectorizer** applies vectorization to the LLVM asm to generate a SIMD program.

- **Evaluator** schedules and evaluates a batch of seeds in parallel.

Translate LLVM ASM to SIMD Application

EVM Bytecode -> LLVM asm -> PTX



EVM Instructions

stack-based language

- stack
- memory
- calldata
- storage
- general calculations
- execution control
- call
- getenv

$$
\begin{aligned}
bytecode &= ins \mid bytecode \\
ins &= \texttt{push}\ c \mid \texttt{pop} \mid \texttt{swap} \mid \texttt{dup} \mid \texttt{op} \mid \texttt{sha3} \mid \texttt{calldata} \mid \\
&\quad \texttt{mload} \mid \texttt{mstore} \mid \texttt{sload} \mid \texttt{sstore} \mid \texttt{call} \mid \texttt{getenv} \mid \\
&\quad \texttt{br} \mid \texttt{bne} \mid \texttt{suicide} \mid \texttt{stop} \mid \texttt{revert}
\end{aligned}
$$

EVM Bytecode Model

Simulated Stack

- μ: the LLVM array to store data
- p: the register to track the stack depth
- μ[p]: the top of the EVM stack

Simulated Stack

- μ: the LLVM array to store data
- p: the register to track the stack depth
- μ[p]: the top of the EVM stack
- thread-safe

PUSH %data

Bytecode

Translate

$$p \leftarrow p + 1; \mu[p] \leftarrow c$$

```
%p = add %p, 1
store %data,(getelementptr
              [1024xi256],[1024xi256]*,@μ, 0, %p)
```

LLVM Assembly

Simulated Memory

- v: the LLVM array for the EVM memory
- thread-safe

MLOAD

$$\mu[p] \leftarrow v[\mu[p]...(\mu[p]+31)]$$

```
%q = load i32,(getelementptr
              [1024xi256], [1024xi256]*, @μ, 0, %p)
%c = load i32, (getelementptr
              [512xi8],[512xi8]*, @v, 0, %q)
store %c, (getelementptr
              [1024xi256], [1024xi256]*, @μ, 0, %p)
```

Simulated Memory

- v: the LLVM array for the EVM memory
- thread-safe

MSTORE

$$v[\mu[p]...(\mu[p]+31)] \leftarrow \mu[p-1];$$
$$p \leftarrow p-2$$

```
%c = load i256, (getelementptr
                [1024xi256], [1024xi256]*, @μ, 0, %p - 1)
%q = load i32, (getelementptr
                [1024xi256], [1024xi256]*, @μ, 0, %p)
store %c, (getelementptr
                [512xi8], [512xi8]*, @v, 0, %q)
%q = sub %q, 2
```

Simulated Calldata

- CALLDATA
  - a byte-addressable array
- Instruction
  - push the calldata buffer from the user input to the stack

Simulated Calldata Vector

- Use LLVM memory operations to emulate the EVM instructions.
- Two-dim vector

CALLDATA
Bytecode

p ← p + 1;
μ[p] ← calldata

Translate

```
%s = getelementptr
        [1024xi256], [1024xi256]* @μ, i32 0,
i32 %p
call void @llvm.memcpy(i8* %s,
        i8 addrspace(1)* %calldata, i64 32, i1
false)
%p = add i32 %p, 1
```

LLVM Assembly

Simulated Calldata Vector

- Use LLVM memory operations to emulate the EVM instructions.
- Two-dim vector

CALLDATA
Bytecode

p ← p + 1;
μ[p] ← calldata

Translate

```
%s = getelementptr
        [1024xi256], [1024xi256]* @μ, i32 0, i32 %p
call void @llvm.memcpy(i8* %s,
       i8 addrspace(1)* %calldata, i64 32, i1 false)
%p = add i32 %p, 1
```

LLVM Assembly

Vectorizer

```
...
%calldata = getelementptr inbounds i8, i8
addrspace(1)* %calldata_vec, i32 %thread_id
...
```

Vectorized LLVM Assembly

Simulated Storage Vector

- Use a hash table as the storage
  - redirect-on-write snapshot
- m: master volume for the storage
- ni: snapshot storage for thread i

```
Function SSTORE(x, y):
    if x ∈ {slot | slot ∈ n_i} then
        n_ix ← y
    else
        n_i ∪ {x : y}
```

```
Function SLOAD(x):
    y ← 0
    if x ∈ {slot | slot ∈ n_i} then
        y ← n_ix
    else
        if k ∈ {slot | slot ∈ m} then
            y ← m_k
    return y
```

Computation Instructions

- μ[p]: LHS
- μ[p−1]: RHS

ISZERO

$\mu[p] \leftarrow \texttt{op}\ \mu[p]$

```
%p = load i32, (getelementptr
                [1024xi256], [1024xi256]* @μ, 0, %p)
%c = load (getelementptr
                [1024xi256], [1024xi256]* @μ, 0, %p)
%z = icmp eq i256 %c, 0
store %z, (getelementptr
                [1024xi256], [1024xi256]* @μ, 0, %p)
```

Hash Instruction

- Keccak256 hash algo in CUDA
- thread-safe
  - only use local registers

Control-flow Instructions

- EVM Jumps:
  - JUMP: jump to μ[p]
  - JUMPI: conditional jump

- Lift EVM jump to table jump
  - i.e., switch
  - over-approximate all possible destinations of the indirect jumps

Translation Rules

EVM Instruction	Translation Rules
`push` c	$p \leftarrow p+1; \mu[p] \leftarrow c$
`pop`	$p \leftarrow p-1$
`dup` c	$p \leftarrow p+1; \mu[p] \leftarrow \mu[p-c]$
`swap` c	$t \leftarrow \mu[p];$
	$\mu[p] \leftarrow \mu[p-c]; \mu[p-c] \leftarrow t$
`mload`	$\mu[p] \leftarrow v[\mu[p]...(\mu[p]+31)]$
`mstore`	$v[\mu[p]...(\mu[p]+31)] \leftarrow \mu[p-1];$
	$p \leftarrow p-2$
`sload`	$\mu[p] \leftarrow \sigma[\mu[p]]$
`sstore`	$\sigma[\mu[p]] \leftarrow \mu[p-1]; p \leftarrow p-2$
`calldata`	$x, y, z \leftarrow \mu[p], \mu[p-1], \mu[p-2];$
	$v[x...(x+z)] \leftarrow d[y...(y+z)]$
unary `op`	$\mu[p] \leftarrow$ `op` $\mu[p]$

EVM Instruction	Translation Rules
binary `op`	$c \leftarrow \mu[p]$ `op` $\mu[p-1]; p \leftarrow p-2;$
	$p \leftarrow p+1; \mu[p] \leftarrow c$
`sha3`	$x, y \leftarrow \mu[p], \mu[p-1]; p \leftarrow p-1;$
	$\mu[p] \leftarrow keccak(v[x...(x+y)])$
`call`	$p \leftarrow p-4; \mu[p] \leftarrow true$
`getenv` env	$p \leftarrow p+1; \mu[p] \leftarrow env$
`bne`	$t, c \leftarrow \mu[p], \mu[p-1]; p \leftarrow p-2;$
	br $c \neq 0, t$, `<fall_through>`;
`br`	$t \leftarrow \mu[p]; p \leftarrow p-1;$
	`switch` t, `<destinations>`;
`stop`	`ret true`
`revert`	`ret false`
`suicide`	move the self balance to $\mu[p]$;
	`ret true`

Code Coverage Instrumentation

- $\mathbf{b_i}$: Bitmap of thread **i**

- **j** = hash(src,dst):
  - edge from one basic block (src) to another one (dst)

$$j = src \ll 1 \oplus dst$$

  - count the branch

$$\mathbf{b_{ij}} = 1$$

- #non-zero bytes in $\mathbf{b_i}$: #code coverage

Code Coverage Counting

- B: incremental bitmap
- B_j: the branch has been explored by any thread yet or not
  - initiate to 0xFF
  - set to zero if i.e., b_{ij} != 0.

$$B_j \leftarrow B_j \& \neg b_{kj}, 0 \leq k < N, \quad i \leq j < i+8, 0 \leq i < \frac{\epsilon}{8}$$

- Interesting?

$$\sum_{i=0}^{\epsilon} B_i \neq 0$$

Experiments for Mau

- Small Benchmark
  - 58 smart contracts with assigned CVEs [SP'20] and an additional 72 curated from Smartbugs [ICSE'20]
- Large Benchmark
  - 400 smart contracts from Ethereum

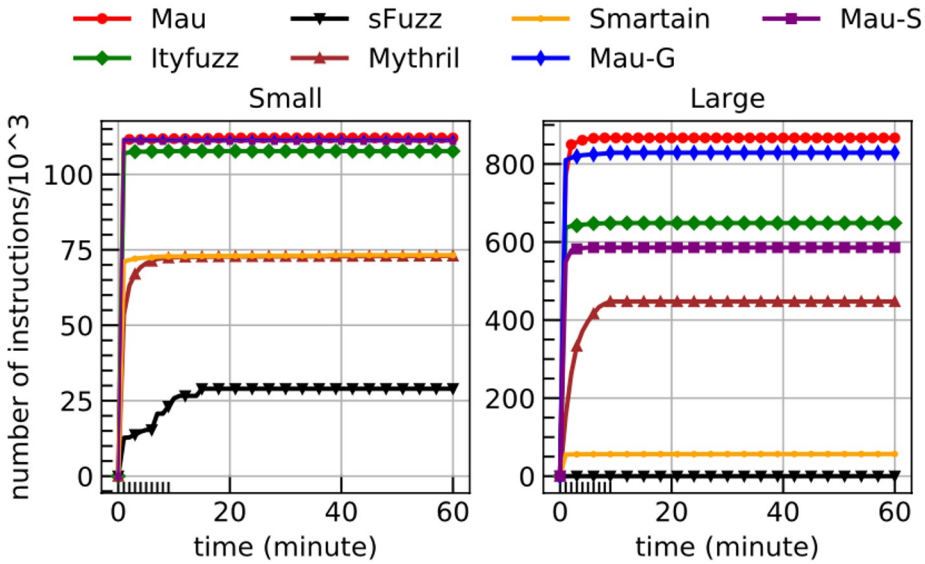
The size of the two benchmarks.

	Cnt	Src/LoC		#Func		Bytecode Size	
		mean	std.	mean	std.	mean	std.
Small	130	136	121.20	10	8.30	4.35 KB	3897.15
Large	400	13099	2961.28	406	117.70	13.95 KB	8328.88

[SP'20] Verismart: A highly precise safety verifier for ethereum smart contracts. SP'20
[ICSE'20] Empirical review of automated analysis tools on 47,587 ethereum smart contracts. ICSE'20

Code Coverage of MAU

- MAU outperformed all baselines in terms of code coverage.
  - It explores 23.36%-371.43% more distinct instructions than different baselines.



Comparison against state-of-the-art tools on both large and small benchmarks.

Bug Detection Ability of MAU

- MAU is more effective than baselines in terms of the accuracy of bug detection.
  - MAU found 1.01X, 2.5X, and 1.65X more bugs than Smartian, sFuzz, and Mythril, respectively.

Table 4: Bug detection on the small benchmark.

Bug	MAU		Smartian		sFuzz		Mythril	
	TP	FP	TP	FP	TP	FP	TP	FP
IB	56	0	53	0	8	0	7	0
BD	10	0	11	0	10	0	8	0
ME	47	0	48	0	29	6	46	0
RE	15	1	19	0	5	20	19	38

Table 5: Bug detection on the large benchmark.

Bug	MAU		Mythril		MAU-G		MAU-S	
	TP	FP	TP	FP	TP	FP	TP	FP
IB	112	0	46	0	104	0	103	0
BD	54	0	28	0	54	0	28	0
ME	108	0	59	0	108	0	99	0
RE	11	3	1	12	10	3	7	3

Throughput of MAU

- MAU is faster than baselines.
  - **162.37K execs/sec** in the large dataset
  - **328.06K execs/sec** in the small dataset
  - surpasses that of SOTA fuzzers, i.e., Ityfuzz , achieving improvements of 8.69x (small dataset) and 15.38x (large dataset), respectively.

MAU's Throughput Over the Number of GPU Cores

- [FSE'20] found that finding new bugs with fuzzing needs an exponential increase in the number of computation resources.
- By **exponentially increasing** the number of GPU cores, the throughput of MAU has a semi-linear increase over the number of GPU cores.

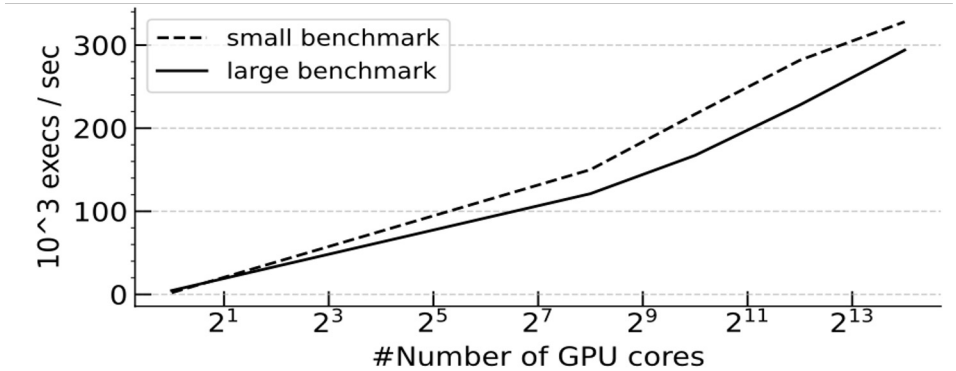


Figure 5: MAU's throughput over the number of GPU cores. The dashed line indicates the throughput when running the small benchmark. The bold line indicates the throughput when running the large benchmark.

[FSE'20]Fuzzing: On the exponential cost of vulnerability discovery. FSE'20

Identified New Bugs by MAU

- Approach:
  - We apply MAU to test all 183,111 real-world smart contracts.
- Results:
  - identified 548 vulnerable projects for stealing cryptocurrency valued at over $172K
  - Two CVE identifiers assigned: CVE-2024-28260 and CVE-2024-28261

The detailed statistics can be found in
https://figshare.com/s/1f2ce70268e6f800470c?file=44708998

Takeaway

- We design and develop MAU, the first GPU-based fuzzer for smart contracts.
- Its throughput surpasses that of SOTA fuzzers achieving improvements of 8.69x and 15.38x, respectively.
- It also detects 1.01-2.50x more bugs and explore 1.03-4.71X more code coverage than baselines.
- Two CVE identifiers assigned
  - CVE-2024-28260, CVE-2024-28261.

Wasm Makes Smart Contracts Safer

EVMBT: A Binary Translation Scheme for Upgrading EVM Smart Contracts to WASM, LCTES'23
WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts, ISSTA'22
WADIFF: A Differential Testing Framework for WebAssembly Runtimes, ASE'23

EVM Disadvantages

- EVM disadvantages
  - no function support
  - DoS
    - limited stack depth (< 1024)
    - expensive instructions: all 256 bits

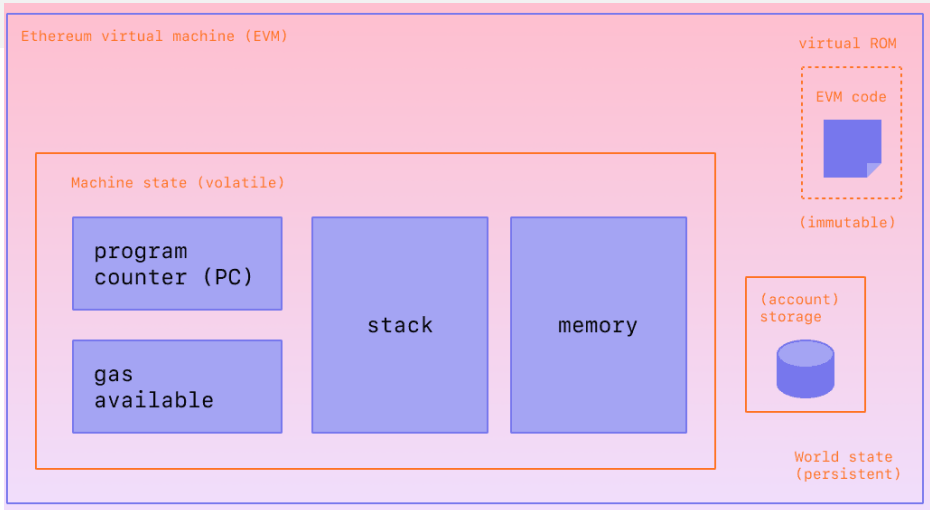


Diagram of the EVM via Ethereum Foundation

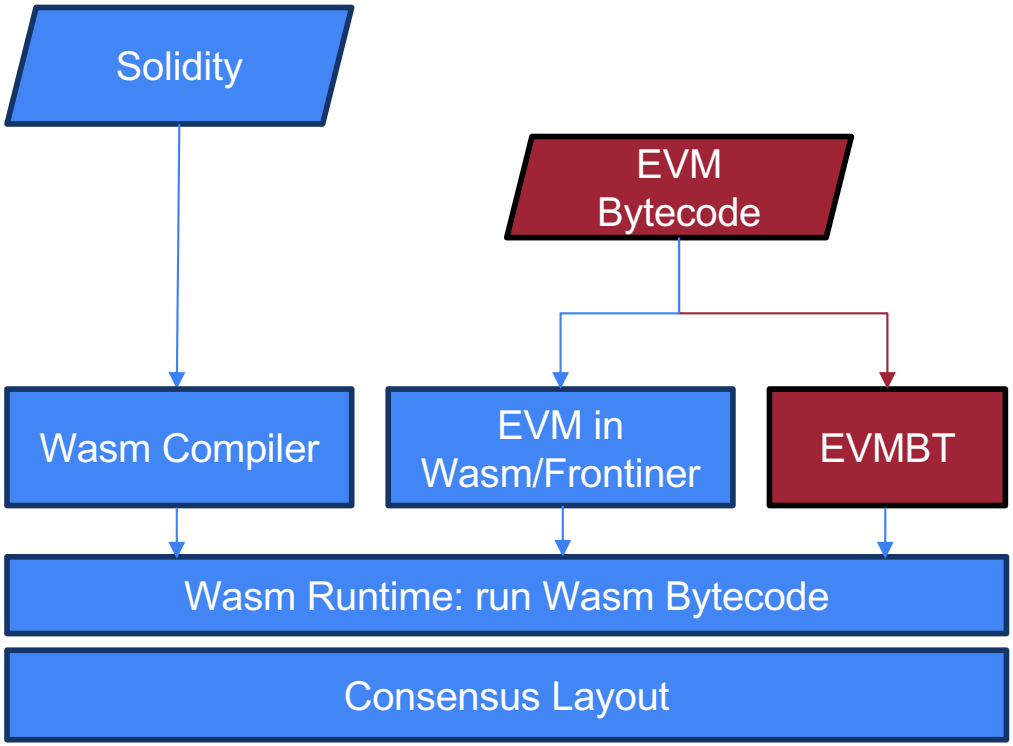
Why using WebAssembly for Smart Contracts?

- WebAssembly (Wasm) for smart contracts
  - Safer (sandbox)
  - More language supports
  - Higher performance
- Migrate ETH smart contracts to Wasm blockchains
  - foster the prosperity of Ethereum
  - inheriting historical transactions.

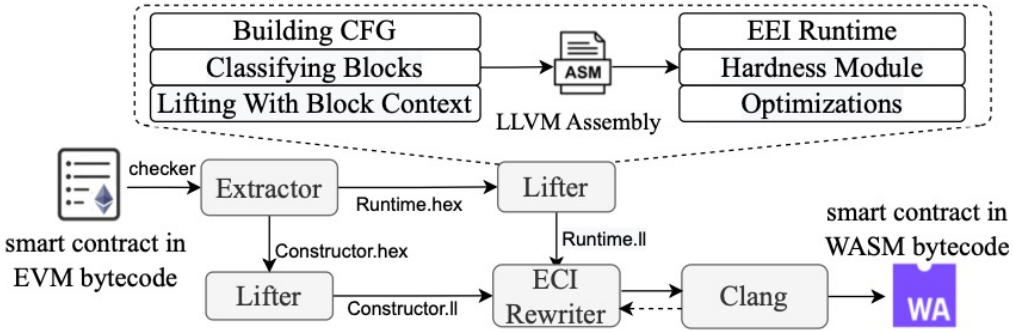
Blockchain	Release Date
EOSIO	Jan. 2018
Polkadot	May. 2020
NEAR	Oct. 2020
Solana	Mar. 2020
Kusama	Oct. 2019
Ethereum 2.0	Dec. 2020

Existing Attempts

- EVM => Wasm smart contract => interpret EVM bytecode
  - NEAR and Polkadot
- Solidity => Wasm bytecode
  - SOLL and Solang
- EVM bytecode => Wasm
  - EVMBT



EVMBT: A Binary Translation Scheme for Upgrading EVM Smart Contracts to WASM (LCTES'24)



The architecture of EVMBT

- **Extractor**: Extract the runtime code and constructor.
- **Lifter**: lift bytecode to LLVM IR (**EEI, less indirect jumps**).
- **ECI-Rewriter**: enable Ethereum deployment.
- **Clang**: Wasm CodeGen.

Translation for Efficient Wasm Code

- Use LLVM memory operations to emulate EVM instructions

PUSH %data; ISZERO

```
%p = add %p, 1;
store %data, (getelementptr
       [1024xi256],[1024xi256]* @μ, 0, %p)
```

Translation for Efficient Wasm Code

- Use LLVM memory operations to emulate EVM instructions

PUSH %data; ISZERO

```
%p = add %p, 1;
store %data, (getelementptr
       [1024xi256],[1024xi256]* @µ, 0, %p)
%c = load (getelementptr
              [1024xi256], [1024xi256]* @µ, 0, %p)
```

Translation for Efficient Wasm Code

- Use LLVM memory operations to emulate EVM instructions

PUSH %data; ISZERO

```
%p = add %p, 1;
store %data, (getelementptr
      [1024xi256],[1024xi256]* @μ, 0, %p)
%c = load (getelementptr
            [1024xi256], [1024xi256]* @μ, 0, %p)
%z = icmp eq i256 %c, 0
store %z, (getelementptr
            [1024xi256], [1024xi256]* @μ, 0, %p)
```

Reduce Redundant LLVM IR

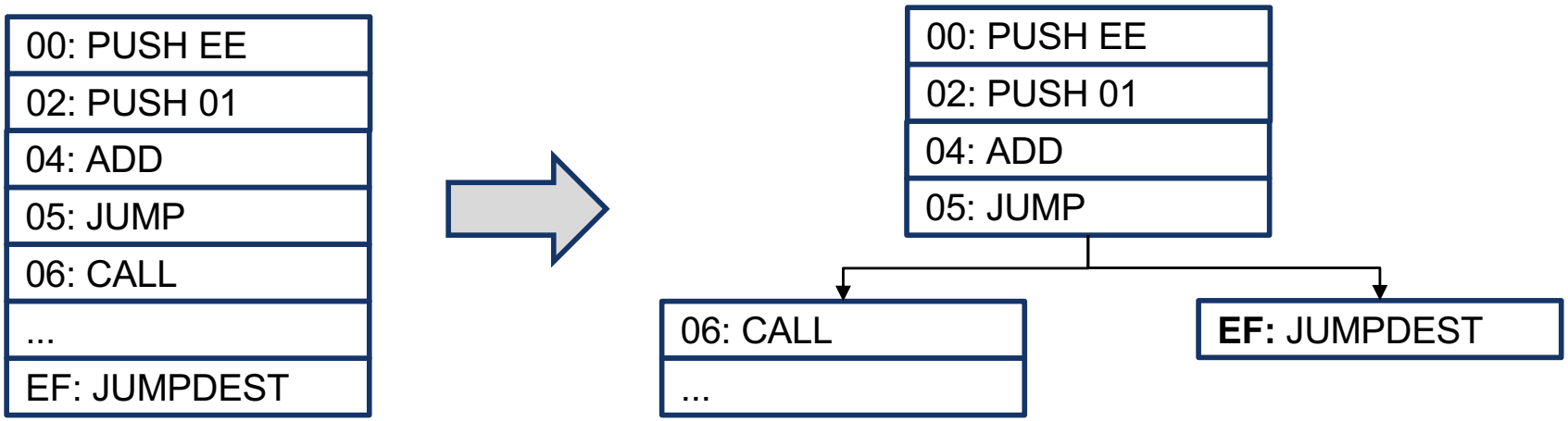
- Infer the operands at compile-time

PUSH %data; ISZERO

```
%z =  icmp eq i256 %data, 0
%p = add %p, 1;
store %z, (getelementptr
       [1024xi256],[1024xi256]* @μ, 0, %p)
```

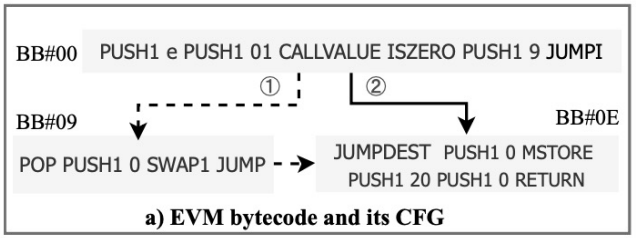
Resolve Indirect Jumps

- Resolve indirect jumps to direct jumps via symbolic execution
  - PUSH EE; PUSH 1; ADD; JUMP
  - JUMPDEST = 0xEF



End-to-End Example

- Cover all EVM instructions



a) EVM bytecode and its CFG

```
function deposit() public returns(bool){
  if (msg.value == 0)
    return false;
  else
    return true;
}
```

b) Original Solidity code

```
1 define void main() "wasm-export-name" {
2 .BB#00:                                   ;; root, eplison=0
3   %sp = load i256, i256* @spPtr
4   store i256 e, i256* getelementptr @mstk[%sp];; PUSH e
5   store i256 1,i256* getelementptr @mstk[%sp+1];;PUSH 1
6   store i256 %sp+2, i256* %spPtr
7   %callvalue = call @getCallValue()          ;; CALLVALUE
8   %cond = icmp eq, i256 callvalue, 0            ;; ISZERO
9   br i1 %cond, label %.BB#09, label %.BB#0E      ;;JUMPI
10
11 .BB#09:                            ;; fall-through, eplison=1
12   %sp = load i256, i256* @spPtr
13   store i256 %sp-1, i256* @spPtr           ;; decrease sp
14   store i256 0, i256* getelementptr @mstk[%sp] ;;record
15   br label %.BB#0E                                ;; JUMP
16
17 .BB#0E:
```

c) LLVM Assembly lifted by Lifter

```
18   %sp = load i256, i256* @spPtr
19   %r = load i256, i256* getelementptr @mstk[%sp]
20   store i256 %r, i256* 0                          ;; MSTORE
21   call void @ethereum.finish(i8* 0, i32 32)    ;; RETURN
22   ret
23 }
```

c) LLVM Assembly by EVMBT (con'd)

```
1 (module
2   (type (;0;) (func))
3   (type (;1;) (func (param i32, i32)))
4   (import "VM" "finish" (func $finish (type 1)))
5   (func $main (type 0)
6     call $ethereum.finish
7   )                          ... ;; code is omitted for clarity
8   (export "main" (func $main)) ;;entry
9 )
```

d) WASM bytecode in text format

An LLVM Assembly example lifted from EVM bytecode.

Experiments for EVMBT

- Eth2 testbed.
  - Ethereum client (geth) + Wasm runtime(Hera)
  - equipped with dual interpreters: EVM + WASM
- Benchmark
  - 2,000 smart contracts and their transactions
  - replay them and validate

Correctness of EVMBT

- Similarity score between $t1$ and $t2$: $\frac{2*M(t_1,t_2)}{|t_1|+|t_2|}$
  - M ($t1$, $t2$) is the number $|t1|+|t2|$ of gestalt pattern matches.
  - $|t|$ is the token amount of t.
- EVMBT can achieve high semantic correctness in translating EVM bytecode.

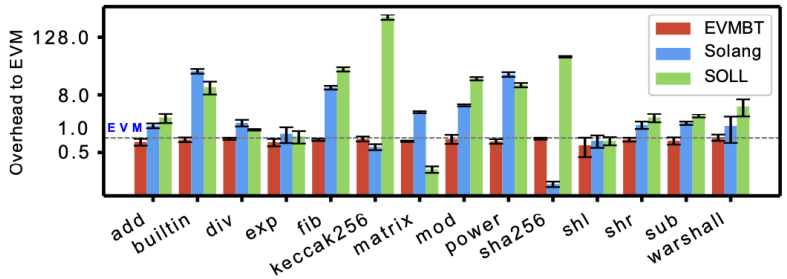
B1 transaction effects.

Tool	Contracts #1,162	Txs #6,854	Same Txs	Score
EVMBT	1,162 (100%)	6,854 (100%)	6,798 (99.2%)	99.998%
SOLL	7 (0.60%)	17 (0.24%)	0 (-)	80.6%
Solang	23 (1.97%)	5 (0.07%)	0 (-)	87.8%

J. Ratcliff and D. Metzener. 1988. Pattern Matching: The Gestalt Approach. In Dr. Dobb's Journal.

Overhead of EVMBT

- 14 synthesis cases sourced from public tests, i.e., Ethereum official tests.
- Gas overhead
  - EVMBT has similar gas efficiency as SOLL (+0.68%) and Solang (+0.15%).
- Time overhead
  - EVMBT achieves a speedup ranging from 1.05x to 1.20x.
  - By contrast, SOLL and Solang introduced 1672.1% and 222.7% overhead.



Time overhead.

Takeaway

- We design EVMBT, a novel EVM2WASM binary translation framework for translating EVM contracts to WASM contracts.
- We conduct extensive experiments to evaluate EVMBT by replaying transactions. The results show that the accuracy of the execution result is much higher than the baselines. EVMBT can generate similar gas-efficient contracts as source code compilers.
- EVMBT is also effective in fixing the vulnerabilities in SmartBugs dataset [39] with small code size overhead (3.4%) and gas overhead (9.3%).

Attack Surfaces in Wasm

- Logical Bugs
- Wasm Runtime Bugs

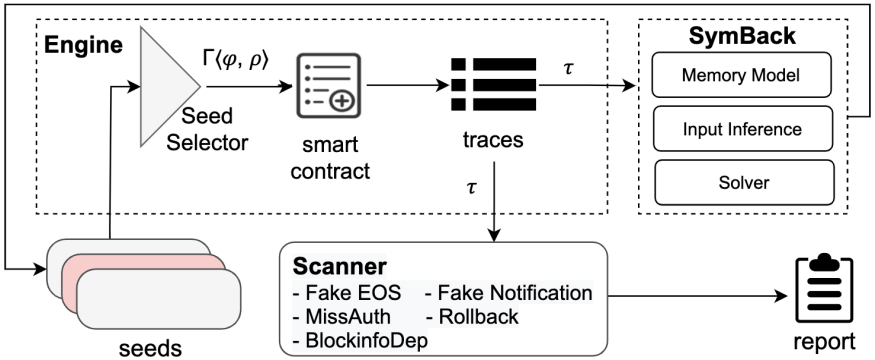
Attack Surfaces in Wasm

- We aim to detect five kinds of commonly-seen vulnerabilities in EOSIO, which cover all vulnerabilities considered in existing studies.

Vulnerability	Root Cause	Consequence
Fake EOS	victim does not check token's issuer	attackers release fake token as EOS
Fake Notification	victim does not check EOS notification message	attackers forge an EOS notification to pretend eosio.token
MissAuth	victim does not verify identification	attackers perform behaviors on behalf of others
Block Dependency	victim uses public block as the random seed	attackers predict the block info
Rollback	victim uses send_inline in *bet-reveal* model	attackers revert execution to deny the transaction

Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020
Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19).

Uncovering Vulnerabilities in Wasm Smart Contracts (ISSTA'22)

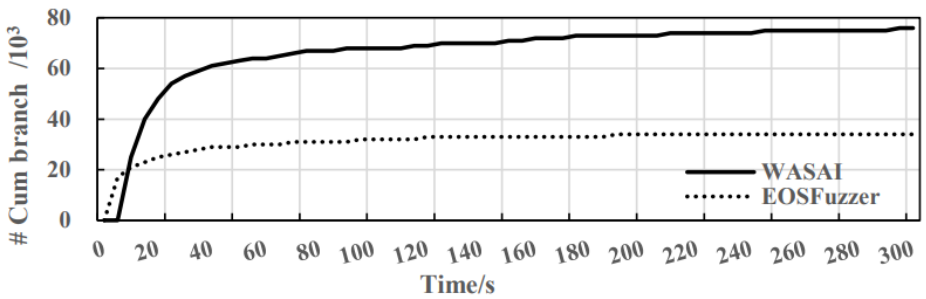


Design of WASAI

- Calling Convention
  - build symbolic variable for input
- Trace Simulation
  - construct and solve constraints
- Seed Schedule
- Vulnerability Scanner

Code Coverage of Wasai

- Dataset
  - 100 real-world smart contracts from the EOSIO blockchain
- WASAI achieves about twice as much code coverage than the baseline.



Branch coverage of WASAI and EOSFuzzer

WASAI's Accuracy of Vulnerability Detection

- Curate a dataset via injecting verified bugs into real-world contracts
  - Fake EOS and Fake Notification: remove the guard code
  - MissAuth: remove/add permission APIs
  - BlockinfoDep and Rollback: generate source code
- Result of WASAI
  - 0 FP and 27 FNs
  - with 100% precision and 98.4% recall

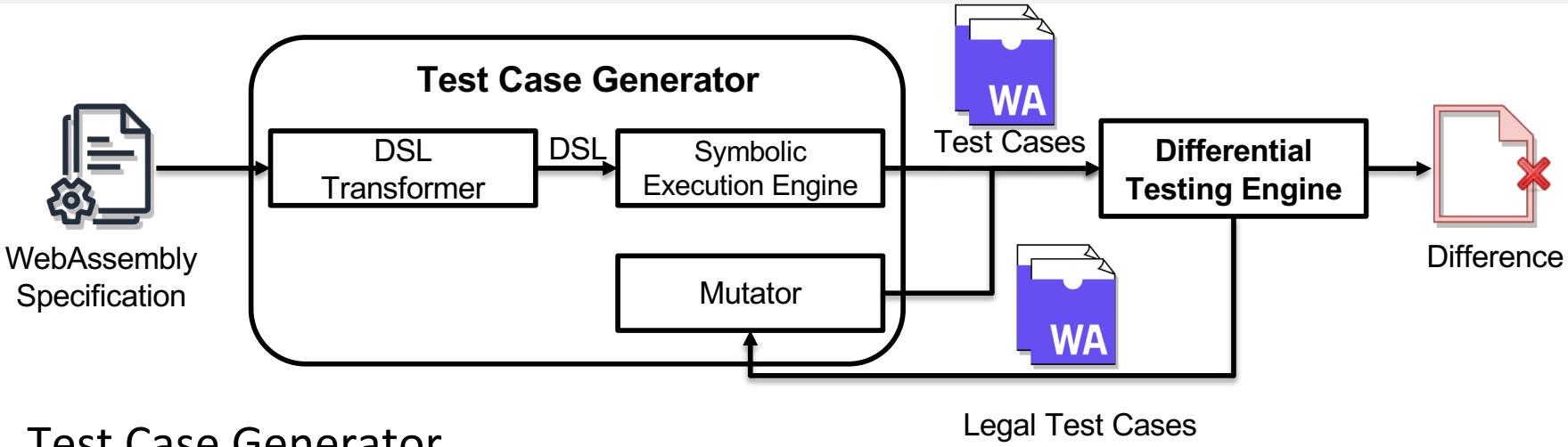
Evaluation results on the ground truth

Types	# Cnt (Vul/Non-Vul)	WASAI			EOSFuzzer			EOSAFE		
		P	R	F1	P	R	F1	P	R	F1
Fake EOS	254(127/127)	100%	100%	100%	90.7%	84.3%	87.3%	98.3%	44.9%	61.6%
Fake Notif	1,378(689/689)	100%	100%	100%	94.9%	78.7%	86.0%	67.4%	98.3%	79.9%
MissAuth	890(445/445)	100%	96.0%	97.9%	-	-	-	100%	38.9%	56.0%
BlockinfoDep	400(200/200)	100%	100%	100%	0%	0%	0%	-	-	-
Rollback	418(209/209)	100%	95.7%	97.8%	-	-	-	50.5%	97.6%	66.6%
Total	**3,340(1,670/1,670)**	**100%**	**98.4%**	**99.2%**	**94.2%**	**63.9%**	**76.1%**	**67.7%**	**75.6%**	**71.4%**

Wild Vulnerabilities Found by Wasai

- 3,881 real-world smart contracts from EOSIO Mainnet
- Results
  - 707 contracts are flagged as vulnerable
  - 58.4% (413) of them are still alive on the Mainnet
  - For the 413 working ones, only 72 of them have been patched
- We identify one zero-day vulnerability with CVE-2022-27134

Differential Testing for Wasm Runtimes (ASE'23)



- Test Case Generator
  - DSL Transformer: transform the specification into the domain specific language (DSL)
  - Symbolic Execution Engine: generate test cases according to the DSL
  - Mutator: generate illegal test cases
- Differential Testing Engine
  - Dump and compare the states of runtimes

Doc to SE

- Identify types

Pop the value $i32.const\ c_2$ from the stack.
Pop the value $i32.const\ c_1$ from the stack.

match the variable part

variable part

Pop (.*) from the stack

fixed content fixed content

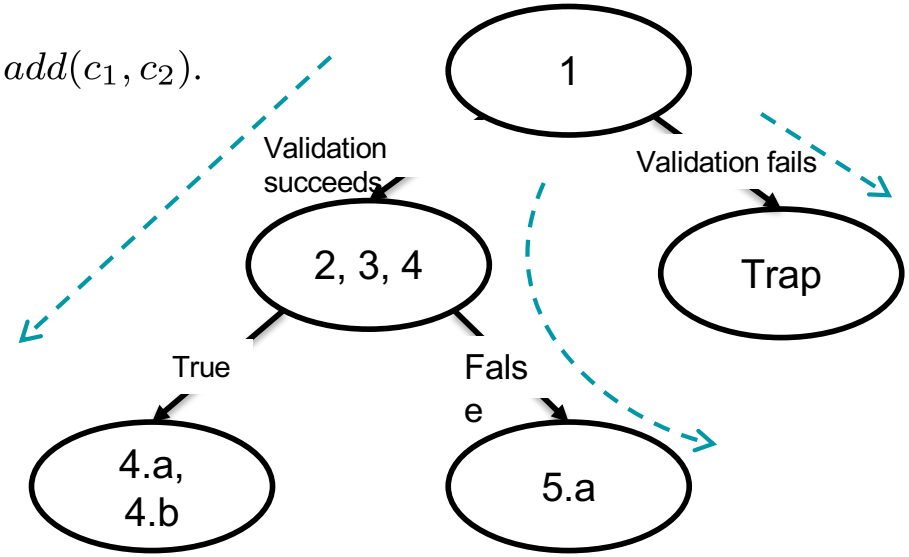
Pop(Alias(c_2))
Pop(Alias(c_1))

match the variable part

Doc to SE

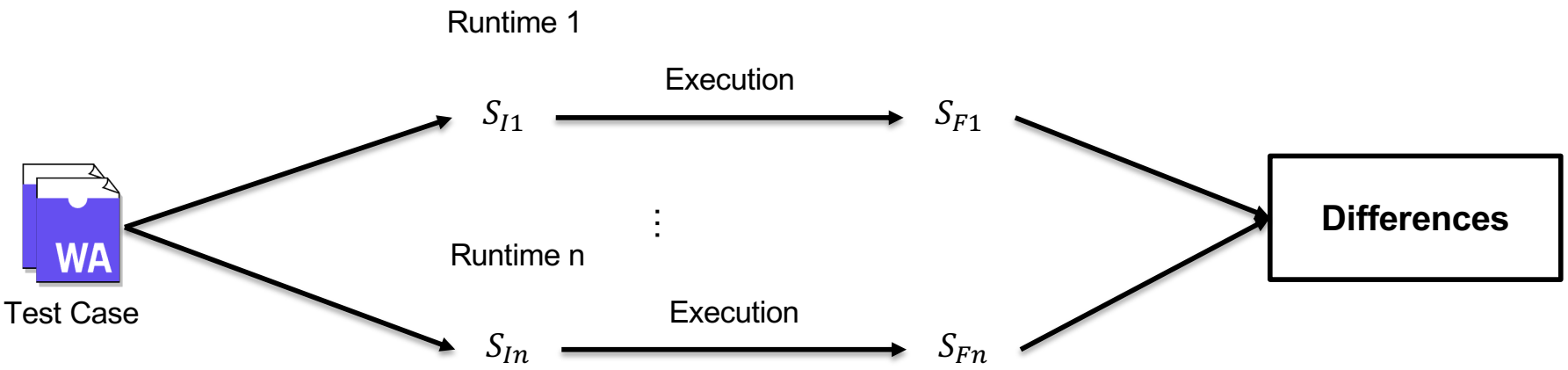
- Identify constraints

1. Assert: due to validation, two values of value type $i32$ are on the top of the stack.
2. Pop the value $i32.const\ c_2$ from the stack.
3. Pop the value $i32.const\ c_1$ from the stack.
4. If $i32.add(c_1, c_2)$ is defined, then:
   a. Let $i32.const\ c$ be a possible result of computing $i32.add(c_1, c_2)$.
   b. Push the value $i32.const\ c$ to the stack.
5. Else:
   a. Trap.



Difference identification

- i.e., linear memory, global variables



Effectiveness of WADIFF

- Runtimes under test
  - wasmer, WAMR-fast, WAMR-classic, WasmEdge, WAVM, wasm3, wasmi
- Generate 124,157 test cases triggering differences
  - among 7 runtimes
  - in about 15 hours
- Detected 21 bugs and 8 of them are confirmed

Takeaway

- Developed the a concolic fuzzer for detecting vulnerabilities in Wasm smart contracts.
- Identified the 0day bug in a real-world smart contract.
- designed an effective differential testing framework for seven Wasm runtimes.
- We can generate sufficient test cases as expected and detect bugs in the wild. In total, 21 bugs are located in all the tested runtimes, with 8 of them being fixed or confirmed by the developers.

Conclusion

- Smart contracts are neither trusted nor secure.
- We design and develop several tools that can effectively detect Ponzi scams, vulnerabilities in the smart contracts.
- Mau is the first GPU-based fuzzer for smart contracts.
- EVMBT is a novel EVM2WASM binary translation framework for translating EVM contracts to WASM contracts.
- WADIFF can detect Wasm runtime bugs in the real world.

References

1. [ISSTA'23] C. Shou, S. Tan, and K. Sen. Ityfuzz: Snapshot-based fuzzer for smart contract. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2023.
2. [ASE'21] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. Cha. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021
3. [SP'20] S. So, M. Lee, J. Park, H. Lee, and H. Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In 2020 IEEE Symposium on Security and Privacy (SP), 2020
4. [ICSE'20] T. Durieux, J. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In Proceedings of the ACM/IEEE 42nd International conference on software engineering (ICSE), Seoul, South Korea, July, 2020
5. [FSE'20] M. B¨ohme and B. Falk. Fuzzing: On the exponential cost of vulnerability discovery. In Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, 2020
6. [TVL]] defillama, "Ethereum TVL", https://defillama.com/chain/Ethereu, 2024
7. [DAO] P. Daian, “Analysis of the dao exploit,” https://hackingdistributed.com/ 2016/06/18/analysis-of-the-dao-exploit/, 2016.

References

8. [Poly] https://decrypt.co/147059/poly-network-attack-conjures-billions-of-dollars-in-tokens-that-did-not-exist
9. [NFT] https://www.bloomberg.com/news/articles/2024-03-27/crypto-hit-by-63-million-munchables-game-hack-in-one-of-year-s-biggest-exploits
10. [sFuzz] T. Nguyen, L. Pham, J. Sun, Y. Lin, and Q. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pages 778–788, 2020.
11. [ILF] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 531–548, 2019.
12. [Echidna] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 557–560, 2020.
13. [Confuzzius] C. Torres, A. Iannillo, A. Gervais, and R. State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In 2021 IEEE European Symposium on Security and Privacy (EuroS&P), pages 103–119. IEEE, 2021.

Thanks for Listening!

Q&A