WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts

Weimin Chen[1], Zihan Sun[2], Haoyu Wang[3],
Xiapu Luo[1], Haipeng Cai[4], Lei Wu[5]

1. The Hong Kong Polytechnic University 2. Beijing University of Posts and Telecommunications Beijing
3. Huazhong University of Science and Technology 4. Washington State University 5. Zhejiang University

WebAssembly and Blockchain

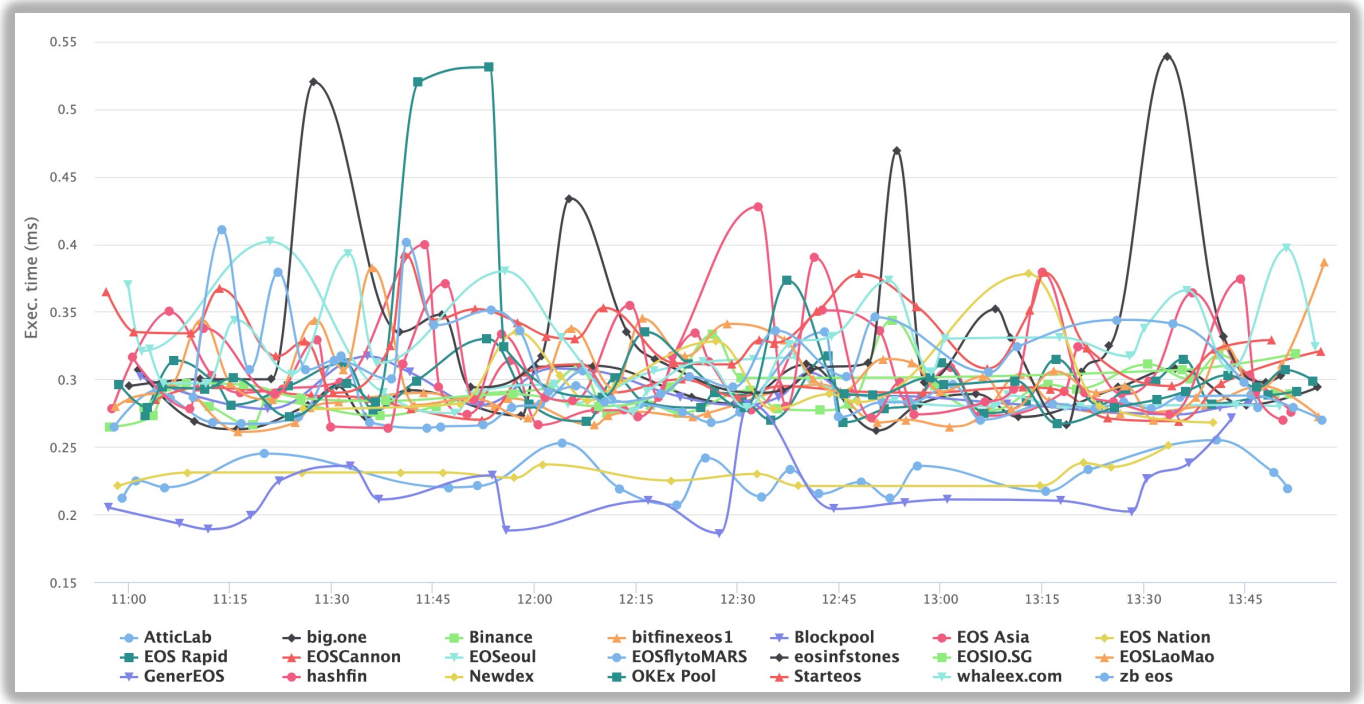
- Why using WebAssembly (Wasm) for Smart Contracts?
  - Open Standards
  - High performance
  - Memory-safe, sandboxed, and platform-independent
  - Many languages available, LLVM support

- How many blockchain platforms have adopted Wasm?

Blockchain	Release Date	Market Capability
EOSIO	**Jan. 2018**	$938,415,321
Polkadot	May. 2020	$6,407,034,521
NEAR	Oct. 2020	$2,450,509,565
Solana	Mar. 2020	$11,957,766,714
Kusama	Oct. 2019	$412,599,922
Ethereum 2.0	Dec. 2020	-

EOSIO

- EOSIO uses a **high-performance** Blockchain Wasm interpreter
- Highest market cap: nearly **18 billion USD** in Apr. 2018
- Over **5.2 Million** accounts
- Our analysis approach for EOSIO smart contracts can be extended to inspect the smart contracts of other Wasm blockchain.



The execution time of custom contracts run by each block producer. The data is from the EOS Mechanics research group.

https://www.alohaeos.com/tools/benchmarks

Outline

0x00 EOSIO Vulnerabilities

0x01 Limitations of Existing Tools

0x02 WASAI

0x03 Evaluation

0x04 Conclusion

Background

eosio.token issues the official token named *EOS*

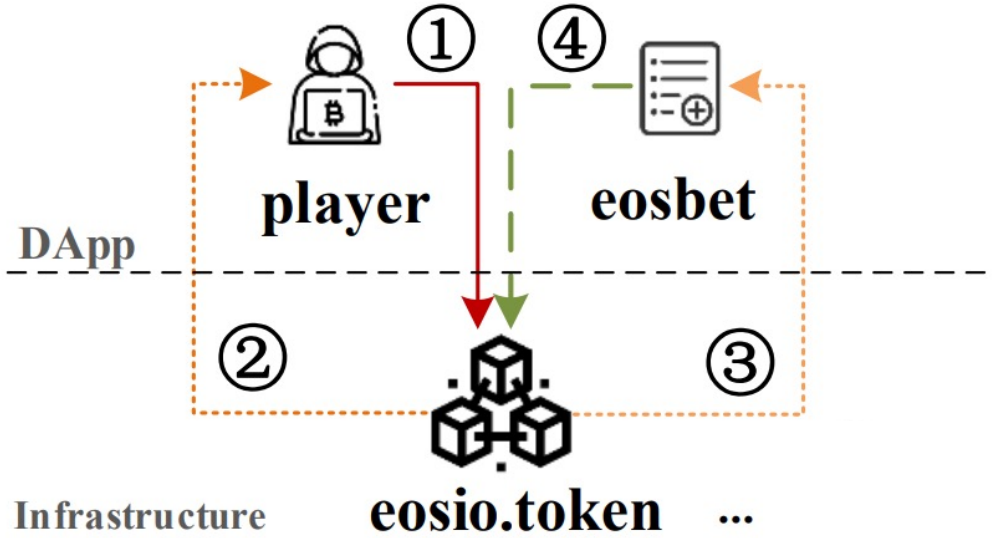
Transferring EOS ===> updating the balance record of **eosio.token**'s database

Background

eosio.token issues the official token named *EOS*

Transferring EOS ===> updating the balance record of **eosio.token**'s database

- Step1. player invokes **transfer@eosio.token** to pay EOS.
- Step2. **eosio.token** forwards the message to notify the player and **eosbet**.
- Step3. **eosbet** interacts with **eosio.token** to reward a lucky user.



Vulnerabilities

- We aim to detect five kinds of commonly-seen vulnerabilities in EOSIO, which cover all vulnerabilities considered in existing studies [29, 30, 61].

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

Outline

0x00 EOSIO Vulnerabilities

0x01 Limitations of Existing Tools

0x02 WASAI

0x03 Evaluation

0x04 Conclusion

Limitations of Existing Tools

- **SE-based static tools (EOSafe [28] and WANA [61])**
  cannot solve complicated constraints and the path explosion issue
- **Dynamic fuzzers (EOSFuzzer [29])**
  cannot feedback the fuzzing to generate effective seeds
  do not analyze transaction dependency
- **Incomplete vulnerability oracles**

Comparison between existing tools and WASAI in bugs detection

Bug	WASAI	EOSFuzzer	EOSafe	WANA
Fake EOS	✓	✓	✓	✓
Fake Notification	✓	✓	✓	✓
MissAuth	✓	✗	✓	✗
Blockinfo Dependency	✓	✓	✗	✓
Rollback	✓	✗	✓	✗
#Total	**5**	**3**	**4**	**3**

Outline

0x00 EOSIO Vulnerabilities

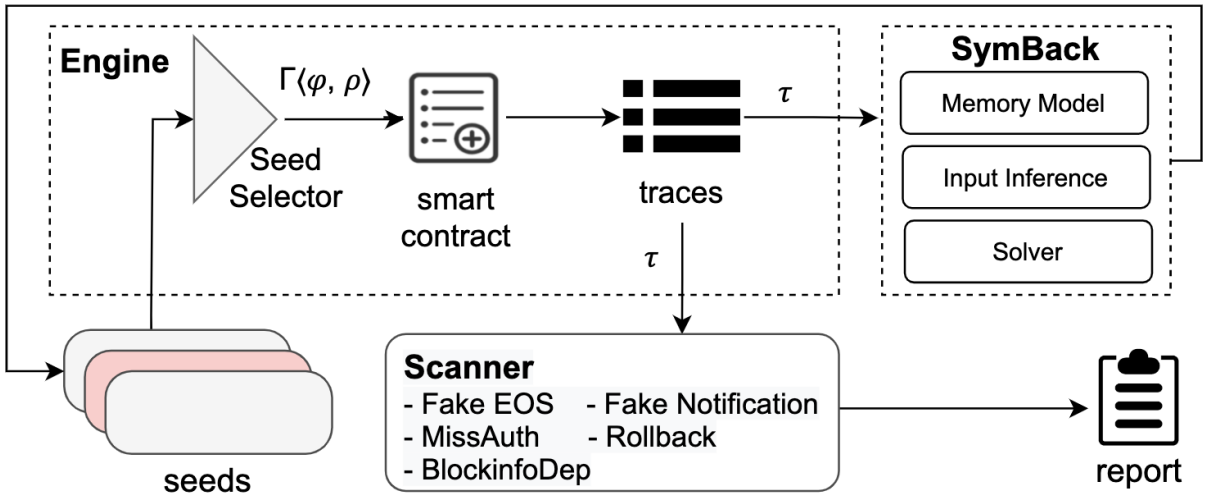
0x01 Limitations of Existing Tools

0x02 WASAI

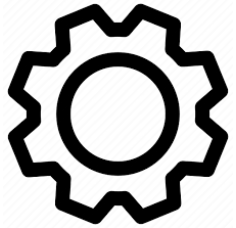
0x03 Evaluation

0x04 Conclusion

WASAI



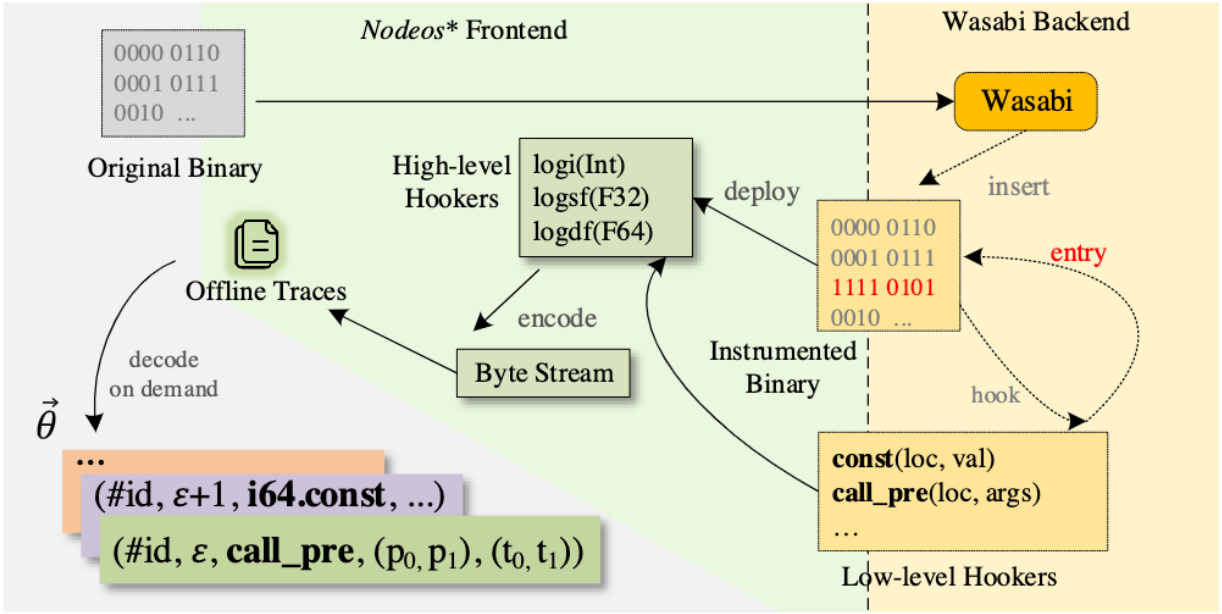
- Instrumentation
- Calling Convention
  - build symbolic variable for input
- Trace Simulation
  - construct and solve constraints
- Seed Schedule
  - analyze data dependency of persistent data to construct transaction sequency
  - send template-based payload.
- Vulnerability Scanner



Challenges #1: Trace Acquisition

WASAI needs to fetch the traces from EOSVM to build symbolic constraints.

- EOSFuzzer instruments Wasm interpreter which supports multi-threads execution.
- EOSFuzzer scarifies the efficiency to execute smart contracts one by one to avoid mixing up the traces from different smart contracts.
- **Solution**
  - contract-level instrumentation
  - Inject low-level hooks to Wasm bytecode
  - hooks execute along with the instructions of smart contracts



Challenges #2: Memory Model

Memory is essential in the execution of Wasm smart contracts.

- There are 23 kinds of memory instructions
- Memory instructions may use a symbolic expression as the address

i32.store (a, 0x0000); i32.store(b, 0xffff);
==> mem[a:a+4] = 0xffff, when a == b

- **Solution**
  - Build a memory model based on byte vectors

$$\triangle.store(\mu_m, addr, size, val) \mapsto \mu_m \quad \triangle.load(\mu_m, addr, size) \mapsto val$$

  - Construct a *symbolic load object* for each load instruction that reads the unsaved memory data.
    - SMT solver will resolve *symbolic load object* by finding a certain address first.

Challenges #3: Calling Convention

We need to map the input data to the functions arguments, for building constraints

- We skip the redundant paths and perform symbolic execution starting from the action function directly.
  - Why?
    - The redundant paths raise **path explosion**.
    - After skip them, we may lose the mapping from the seed to function arguments, and consequently seed mutation will be affected.

- **Solution:**
  - The arguments of a Wasm function are initiated as local registers.
  - We can infer the specific arguments in the Wasm layout of each input data.

Symback

- Observation

  There is a consistent one-to-one mapping between the runtime values and seed parameters in the context of the action function.

$$\mu_{\hat{l}}[i+1] \Longleftrightarrow \vec{\rho}_i \text{ ;when } 0 \leq i \leq \left|\vec{\rho}\right|$$

- We can infer the input for *transfer(name from, name to, asset quantity, string memo)*

ρ	Type	Local	Linear Memory
from	name	$\mu_{\hat{l}}[1]$	-
to	name	$\mu_{\hat{l}}[2]$	-
quantity	asset	$\mu_{\hat{l}}[3]$	$\mu_m[\mu_{\hat{l}}[3] : \mu_{\hat{l}}[3]+8] \leftarrow$ amount item $\mu_m[\mu_{\hat{l}}[3]+8 : \mu_{\hat{l}}[3]+16] \leftarrow$ symbol item
memo	string	$\mu_{\hat{l}}[4]$	$\mu_m[\mu_{\hat{l}}[4]] \leftarrow$ string length $\mu_m[\mu_{\hat{l}}[4]+1 : \mu_{\hat{l}}[4]+\mu_m[\mu_{\hat{l}}[4]]+1] \leftarrow$ string content

Symback

WASAI lifts low-level information in runtime traces to high-level machine states

Enabling Symback to build and solve symbolic constraints for fuzzer feedback.

The operational semantic of Wasm instructions.

Instruction	Machine state changes	Instruction	Machine state changes	Instruction	Machine state changes
const im	$\mu_{\hat{s}}.push(im)$	global.get im	$x = \mu_g[im]; \mu_{\hat{s}}.push(x)$	br	-
drop	$\mu_{\hat{s}}.pop()$	global.set im	$x = \mu_{\hat{s}}.pop(); \mu_g[im] = x$	br_if	$\mu_{\hat{s}}.pop()$
select	$z, x, y = \mu_{\hat{s}}.pop(3); \mu_{\hat{s}}.push(z?y : x)$	load	$\triangle.load$	br_table	$\mu_{\hat{s}}.pop()$
unary	$x = \mu_{\hat{s}}.pop(); \mu_{\hat{s}}.push(op(x))$	store	$\triangle.store$	return	-
binary	$x, y = \mu_{\hat{s}}.pop(2); \mu_{\hat{s}}.push(op(x, y))$	call_pre	$args = \mu_{\hat{s}}.pop(\$); \mu_l.push(args)$	unreachable	-
local.get im	$x = \mu_{\hat{l}}[im]; \mu_{\hat{s}}.push(x)$	call_post	$\mu_l.pop(); rts = \mu_r.pop(); \mu_{\hat{s}}.extend(rts)$	nop	-
local.set im	$x = \mu_{\hat{s}}.pop(); \mu_{\hat{l}}[im] = x$	begin_function	$\mu_s.push(\emptyset)$	memory.grow	$\mu_{\hat{s}}.pop(); \mu_{\hat{s}}.push(4096)$
local.tee im	$\mu_{\hat{l}}[im] = \mu_{\hat{s}}[0]$	end_function	$rts = \mu_s.pop(); \mu_r.push(rts)$	memory.size	$\mu_{\hat{s}}.push(4096)$

Scanner

Scan exploit events

- Fake EOS

$$vul := id_e \in \vec{id}$$

- Fake Notification

$$vul := id_e \in \vec{id} \wedge \vec{\tau} \not\ni (i64.\text{eq}|i64.\text{ne}, (\texttt{fake.token}, \texttt{\_self}))$$

- MissAuth

$$vul := any(\{\underset{0 \to i}{\vec{id}} \cap Auths = \emptyset \wedge id_i \in Effects \mid i > 0\})$$

- Block Dependency

$$vul := \vec{id} \cap \{\#tapos\_block\_prefix, \#tapos\_block\_num\} \neq \emptyset$$

- Rollback

$$vul := \#send\_inline \in \vec{id}$$

Outline

0x00 EOSIO Vulnerabilities

0x01 Limitations of Existing Tools

0x02 WASAI

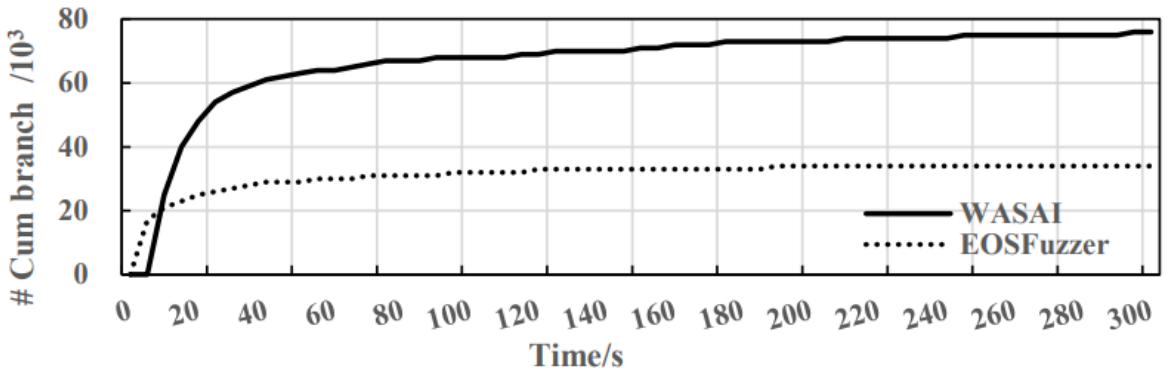
0x03 Evaluation

0x04 Conclusion

RQ1: Code Coverage

- Dataset
  - We randomly select 100 real-world smart contracts from the EOSIO blockchain

- We compare WASAI with EOSFuzzer
  - WASAI obtains over 75,000 distinct branches for all these contracts



Branch coverage of WASAI and EOSFuzzer

Answer: *WASAI achieves about twice as much code coverage than the baseline.*

RQ2: 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%**

Answer: *WASAI outperforms all state-of-the-art techniques on a large-scale benchmark, with 100% of precision and 98.4% of recall.*

RQ3:Robustness of WASAI

Code Benchmark

- WASAI can remain a high accuracy, with 96.6% of precision and 97.9% of recall.
- EOSafe cannot find feasible paths in bug detection.

The impact of code obfuscation

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%	91.4%	92.1%	91.8%	0%	0%	0%
Fake Notif	1,378(689/689)	92.4%	100%	96.0%	94.6%	78.1%	85.5%	67.5%	98.4%	80.0%
MissAuth	890(445/445)	100%	94.2%	97.0%	-	-	-	0%	0%	0%
BlockinfoDep	400(200/200)	100%	100%	100%	0%	0%	0%	-	-	-
Rollback	418(209/209)	100%	95.7%	97.8%	-	-	-	50.4%	97.1%	66.3%
Total	**3,340(1,670/1,670)**	**96.6%**	**97.9%**	**97.3%**	**94.0%**	**64.5%**	**76.5%**	**62.6%**	**59.9%**	**61.2%**

Complicated Benchmark

- WASAI retains 99.9% precision and 92.5% recall.
- EOSFuzzer's random seeds cannot touch deep code.

The impact of complicated verification

Types	# Cnt (Vul/Non-Vul)	WASAI			EOSFuzzer			EOSAFE		
		P	R	F1	P	R	F1	P	R	F1
Fake EOS	190(95/95)	100%	100%	100%	50.0%	100%	66.7%	100%	43.2%	60.3%
Fake Notif	1,178(589/589)	99.6%	83.0%	90.6%	0%	0%	0%	68.1%	99.3%	80.8%
MissAuth	756(378/378)	100%	97.4%	98.7%	-	-	-	100%	40.5%	57.6%
BlockinfoDep	400(200/200)	100%	100%	100%	0%	0%	0%	-	-	-
Rollback	400(200/200)	100%	100%	100%	-	-	-	50%	100%	66.7%
Total	**2,924(1,462/1,462)**	**99.9%**	**92.5%**	**96.0%**	**50%**	**10.7%**	**17.7%**	**67.4%**	**77.6%**	**72.1%**

Answer: *WASAI is more robust than the baselines.*

RQ4:Vulnerabilities in the Wild

- Dataset
  - 3,881 real-world smart contracts from EOSIO Mainnet
  - 991 (25.5%) of them accept EOS.

- Results
  - 707 contracts (71.3%) 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 in *batdappboom* (**CVE-2022-27134**)

Answer: *We reveal the urgency of fixing the vulnerabilities in EOSIO smart contracts.*

Conclusion

- We developed WASAI, the **first concolic fuzzer** for Wasm smart contracts.
- We constructed a **large-scale benchmark** with 3,340 samples. Extensive experimental results demonstrate that WASAI outperforms all state-of-the-art approaches, with an F1-measure of 99.2%.
- We applied WASAI to 991 deployed Wasm contracts. 70% of smart contracts are vulnerable. By the time of this study, over 300 of them have not been patched yet.
- We reported a Fake EOS vulnerability to the EOSIO ecosystem, which was recently assigned a CVE identifier (**CVE-2022-27134**).

Thank you for listening!

Weimin Chen: cswchen@comp.polyu.edu.hk

code: https://github.com/wasai-project/wasai

Benchmark: https://drive.google.com/file/d/1z1rd3o0o6zoYVNcKXpnHWqDLn4EwdcP-/view?usp=sharing