Seraph: Enabling Cross-Platform Security Analysis For EVM and WASM Smart Contracts

Zhiqiang Yang
zhiqiang@oxhainan.org
Oxford-Hainan Blockchain Research Institute
Hainan, China

Han Liu
liuhan@oxhainan.org
Oxford-Hainan Blockchain Research Institute
Hainan, China

Yue Li
liyue@oxhainan.org
Oxford-Hainan Blockchain Research Institute
Hainan, China

Huixuan Zheng
huixuan@oxhainan.org
Oxford-Hainan Blockchain Research Institute
Hainan, China

Lei Wang
wanglei@oxhainan.org
Oxford-Hainan Blockchain Research Institute
Hainan, China
Shanghai Jiao Tong University
Shanghai, China

Bangdao Chen
bangdao@oxhainan.org
Oxford-Hainan Blockchain Research Institute
Hainan, China

ABSTRACT

As blockchain becomes increasingly popular across various industries in recent years, many companies started designing and developing their own smart contract platforms to enable better services on blockchain. While smart contracts are notoriously known to be vulnerable to external attacks, such platform diversity further amplified the security challenge. To mitigate this problem, we designed the very first cross-platform security analyzer called Seraph for smart contracts. Specifically, Seraph enables automated security analysis for different platforms built on two mainstream virtual machine architectures, *i.e.*, EVM and WASM. To this end, Seraph introduces a set of general *connector API* to abstract interactions between the virtual machine and blockchain, *e.g.*, load and update storage data on blockchain. Moreover, we proposed the *symbolic semantic graph* to model critical dependencies and decoupled security analysis from contract code as well. Our preliminary evaluation on four existing smart contract platforms demonstrated the potential of Seraph in finding security threats both flexibly and accurately. A video of Seraph is available at `https://youtu.be/wxixZkVqUsc`.

CCS CONCEPTS

• **Security and privacy** → *Domain-specific security and privacy architectures.*

KEYWORDS

smart contracts, connector API, symbolic semantic graph

ACM Reference Format:
Zhiqiang Yang, Han Liu, Yue Li, Huixuan Zheng, Lei Wang, and Bangdao Chen. 2020. Seraph: Enabling Cross-Platform Security Analysis For EVM and WASM Smart Contracts. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382157

1 INTRODUCTION

The blockchain technology has been undergoing a rapid growth in recent years via promising to enable traceable transactions in a decentralized network without a trusted third-party. As a form of blockchain programs or scripts, smart contracts [17] have been gaining an increasing popularity across different application domains as well, *e.g.*, supply chain finance, insurance, cryptocurrency *etc.*. In order to achieve better blockchain services, many companies introduced their own smart contract platforms.

```
function transfer(address _s, address _r, uint256 _val) {
  ...
  balances[_s] = balances[_s].sub(_val);
  balances[_r] = balances[_r].add(_val);
  emit Transfer(_s, _r, _val);
}
```

(a) `transfer` function on Ethereum

```
function transfer(string _s, string _r, uint256 _val) {
  ...
  Entry entry0 = table.newEntry();
  entry0.set("account", _s);
  entry0.set("asset_value", int256(_s_val - _val));
  int cnt = table.update(_s,entry0,table.newCondition());
  ...
}
```

(b) `transfer` function on FISCO-BCOS

Figure 1: Smart contracts on Ethereum and FISCO-BCOS

Commonly, these platforms share similar infrastructure but manifest unique features at the same time. For example, while both Ethereum [17] and FISCO-BCOS [1] use the Ethereum Virtual Machine (EVM) runtime, they adopt different design for blockchain

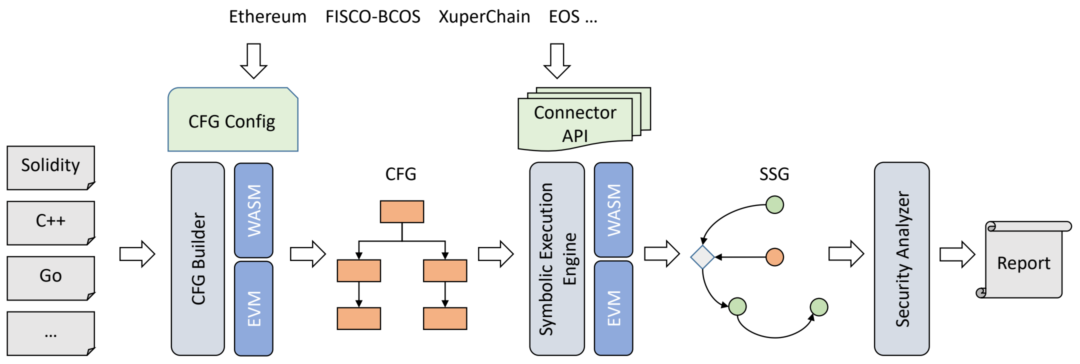


Figure 2: The general workflow of cross-platform security analysis for smart contracts.

storage. From the security perspective, such diversity raises new challenges for smart contracts, which are notoriously known to be vulnerable to external attacks [11, 13, 16]. That is, despite that those platforms have similar designs, it is still non-straightforward to create security analyzers for them, *i.e.*, adapting a tool of platform A for B commonly involves low-level code refactoring across many components, which is a non-trivial work in practice. We further summarize concrete challenges as below.

Challenge 1: VM-Blockchain Interaction. One of the major differences among smart contract platforms is the way virtual machines interact with their blockchains, *e.g.*, load and update storage data on blockchain. As shown in Figure 1, two `transfer` functions are defined on Ethereum and FISCO-BCOS, respectively. Particularly, Ethereum stores data in *state variables* (*i.e.*, `balances` in this case), thus storage accesses are realized via variable assignments (line 3 and 4 in Figure 1a). On the other hand, FISCO-BCOS adopts the table design where data is stored in a database. In that case, load and update on a storage are realized via table operations, as line 4-6 in Figure 1b. Although the contracts are manipulating storage data in both cases, analyzing them is non-straightforward since we need to handle storage access instructions for Ethereum but function calls for FISCO-BCOS.

Challenge 2: Security Analysis Process. Moreover, the current design of security analyzers is highly platform-specific. That is, the analyzer on one platform can hardly be directly applied in another. The reason behind is the lack of a general form of semantic representation for smart contracts where different types of security analysis can be implemented. While this is a long-lasting topic in the community of programming language, it is relatively little studied in the contexts of smart contracts, leaving room for adaptation and further optimization.

The Seraph Solution. To overcome challenges above, we designed and developed Seraph, the very first cross-platform security analyzer for blockchain smart contracts based on EVM and WASM runtime. Specifically, Seraph highlighted a set of *connector API* to abstract the interaction between virtual machines and their host blockchains. Furthermore, we proposed the *symbolic semantic graph* (SSG) as a general and lightweight representation for critical smart contract logics and dependencies. Compared to traditional program dependency graphs, SSG is able to model blockchain semantics and capture a large classes of smart contract vulnerabilities in an efficient and accurate way. In the preliminary evaluation, we applied Seraph on four smart contract platforms in the literature and managed to uncover security threats for all of them.

2 CROSS-PLATFORM SECURITY ANALYSIS

2.1 Overview

The general workflow of Seraph is shown in Figure 2. Specifically, Seraph accepts smart contracts written in different high-level programming languages as inputs, *e.g.*, Solidity, C++, Go *etc.*. Additionally, Seraph requires specifying the target smart contract platform, *e.g.*, Ethereum. Based on the platform, the user is allowed to configure the structure of control flow graph (CFG), *e.g.*, labeling low-level instructions as control flow operators. Then, the input smart contracts are compiled and transformed into an EVM or WASM CFG depending on which infrastructure is used in the platform. Furthermore, Seraph leverages symbolic execution [10] to systematically explore the CFG. Particularly, we implement a group of *Connector API* (§2.2) for the target platform in order to abstract VM-blockchain interaction. The goal of symbolic execution is to generate *symbolic semantic graph* (SSG, explained in §2.3) as a lightweight representation for the input smart contracts.

Next, the security analysis for the input smart contracts is automated and based on SSG. Specifically, the analysis task (*e.g.*, detecting integer overflow) is converted to a *multi-path graph query*, *i.e.*, search for multiple paths in SSG. Lastly, a security report is generated for the analysis.

2.2 Connector API

As aforementioned, the *Connector API* is defined to abstract the interaction between a virtual machine and its host blockchain. Generally, Seraph provided two types of APIs, *i.e.*, environment APIs and state APIs, respectively. Environment APIs are used to compute information of the blockchain environment. Figure 3a shows an implementation of the `get_block_hash` API for Ethereum. Specifically, the API retrieves a hash value for a given block with number `_num`. In the implementation of the API, we compute the difference between a symbolic value `Hb` (current block number) and `_num`.

Then, symbolic constraints are generated depending on the difference. According to Ethereum, if the difference is less than 256, a concrete block hash is generated. Here, we used a symbolic value to represent the hash. Otherwise, the hash is set to 0.

```
def get_block_hash(_num):
  gap = Hb - block_num
  solver.add((h == HASH_ + _num && gap < 256) \
    || (h == 0 && gap >= 256))
  stack.push(h)
  return h
```

(a) `get_block_hash` API on Ethereum

```
1 def update_storage(key, value):
2   if key in list(storage.keys()):
3     storage[key] = value
4     ssg.update(key, value, meta)
```

(b) `update_storage` API on FISCO-BCOS

Figure 3: Two instances of connector APIs

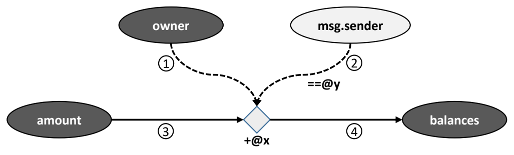
On the other hand, Figure 3b demonstrates an implementation of the `update_storage` state API, which is designed to load or update storage data on blockchain. Specifically, the API takes as inputs a key-value pair. Given an existing key, Seraph first updates the storage data (line 3) and further the dependency in the symbolic semantic graph (line 4), which will be explained later.

2.3 Symbolic Semantic Graph

The design of *symbolic semantic graph* (SSG) is based on an observation that the majority of security issues in smart contracts can be encoded as simple data dependency problems, *i.e.*, whether data A is dependent on data B. SSG is designed to model critical dependency information in a concise manner.

```
function reward(uint8 amount):
  if(owner == msg.sender)
    balances[msg.sender] += amount;
```

(a) A simple Solidity function on Ethereum



(b) SSG of Figure 4a

Figure 4: An example of symbolic semantic graph

Figure 4a shows an Ethereum Solidity function `reward`, which increases the balance of contract `owner`. Furthermore, the SSG of `reward` function is shown in Figure 4b. We defined three types of graph nodes, *i.e.*, Value Node (dark ellipse), Environment Node (light ellipse) and Flow Node (diamond). Value node models storage data on blockchain. Environment node indicates environment information of blockchain itself. Flow node is a connection point. In addition, SSG has two types of edges, *i.e.*, flow edge (solid line) and control edge (dashed line). While flow edges represent data dependency (the value of `balances` is dependent on the value of `amount`), control edges refer to control dependency (`msg.sender` affects the execution of `balances` update). Both nodes and edges can have attributes. In this case, the flow node has an attribute `+@x`, indicating that the data dependency is involved in an addition at program counter `x`. Similarly, `==@y` indicates the control edge derives from a equivalence check at program counter `y`. Compared to traditional program dependency graph (PDG) [7, 8, 15], SSG interprets blockchain semantics and discards program orders to reduces state space. Based on SSG, security analysis is encoded as *multi-path graph queries*. For instance, to detect potential integer overflow in the contract, we could ask Seraph to search for crossing paths of flow and control edges, *e.g.*, path 2 and 3 ⟶ 4. Although the addition introduces a potential overflow, the execution is controlled by a sanity check on `msg.sender` thus leads to no overflow.

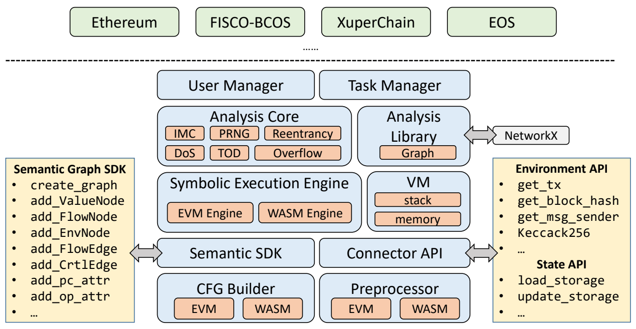
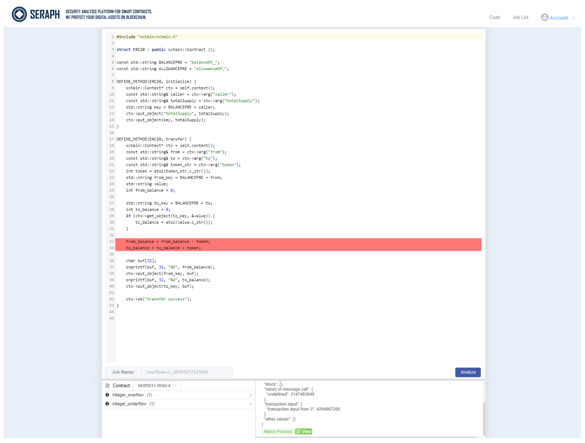


Figure 5: The architecture of Seraph.

3 DESIGN OF SERAPH

Architecture. The current version of Seraph is a web platform with front- and back-end services. The back-end architecture of Seraph is shown in Figure 5. Specifically, at the top layer are user and task management modules, which are designed to manage registered users and analysis tasks submitted to Seraph. The rest of the components are used to enable cross-platform security analysis as discussed above, including preprocessor (compile, parse, disassemble *etc.*), CFG builder, connector API, symbolic execution engine for EVM and WASM, and core analyzers. We created the Semantic SDK to help construct intermediate representation (*i.e.*, SSG) for security analysis. For example, `add_ValueNode` in the SDK is designed to insert a value node to an existing SSG. Moreover, we have also integrated an Analysis Library to encapsulate the implementation of SSG. We developed Seraph in Python and used Z3 [2] as the SMT solver. In terms of the analysis library, NetworkX was adopted to construct SSG structures.

Main Functionalities. Figure 6 shows a screenshot of Seraph. Currently, the target users include both individual smart contract developers and blockchain service providers. Three main functionalities are provided, *i.e.*, user registration, task management and


SERAPH
SECURITY ANALYSIS PLATFORM FOR SMART CONTRACTS.
WE PROTECT YOUR DIGITAL ASSETS ON BLOCKCHAIN.
Code
Job List
Account
Job Name:
Analyze
Contract:
Attack Process
View

```
#include "xchain/xchain.h"

struct ERC20 : public xchain::Contract {};

const std::string BALANCEPRE = "balanceOf_";
const std::string ALLOWANCEPRE = "allowanceOf_";

DEFINE_METHOD(ERC20, initialize) {
    xchain::Context* ctx = self.context();
    const std::string& caller = ctx->arg("caller");
    const std::string& totalSupply = ctx->arg("totalSupply");
    std::string key = BALANCEPRE + caller;
    ctx->put_object("totalSupply", totalSupply);
    ctx->put_object(key, totalSupply);
}

DEFINE_METHOD(ERC20, transfer) {
    xchain::Context* ctx = self.context();
    const std::string& from = ctx->arg("from");
    const std::string& to = ctx->arg("to");
    const std::string& token_str = ctx->arg("token");
    int token = atoi(token_str.c_str());
    std::string from_key = BALANCEPRE + from;
    std::string value;
    int from_balance = 0;

    std::string to_key = BALANCEPRE + to;
    int to_balance = 0;
    if (ctx->get_object(to_key, &value)) {
        to_balance = atoi(value.c_str());
    }

    from_balance = from_balance - token;
    to_balance = to_balance + token;

    char buf[32];
    snprintf(buf, 32, "%d", from_balance);
    ctx->put_object(from_key, buf);
    snprintf(buf, 32, "%d", to_balance);
    ctx->put_object(to_key, buf);

    ctx->ok("transfer success");
}
```

Job Name: overflow.cc_20191213121056

Contract: b60f5031-050d-4

integer_overflow (1)

integer_underflow (1)

```
"block": {},
"return of message call": {
  "undefined": 2147483649
},
"transaction input": {
  "transaction input from 3": 4294967295
},
"other values": {}
}
```

Figure 6: A screenshot of Seraph.

security analysis. The platform works in a push-button manner. After a user logs in and pushes the analysis button, a task is submitted to Seraph for job scheduling. Once the analysis is finished, users can explore detailed information, including an end-to-end exploit.

4 PRELIMINARY EVALUATION

Table 1 describes the preliminary detection accuracy on Ethereum contract benchmarks [4, 5]. Compared to analyzers in the literature, *i.e.*, Oyente [13], Securify [16] and Mythril [3], Seraph achieved a higher accuracy over four types of vulnerabilities.

Table 1: Detection accuracy on Ethereum. IO: Integer overflow. PRNG: Psuedo Random Number Generator. IMC: Insecure Message Call. $*$: Unsupported.

Threats	Oyente	Securify	Mythril	Seraph
IO	64.29%	$*$	85.71%	92.86%
DoS	$*$	40.00%	$*$	100.00%
PRNG	$*$	$*$	33.33%	66.67%
IMC	66.67%	66.67%	33.33%	83.33%

Table 2: Detection of integer overflow on four platforms.

Contract	Platform	Detected?
`overflow_eth.sol`	Ethereum	✓
`overflow_bcos.sol`	FISCO-BCOS	✓
`overflow_xchain.cc`	XuperChain	✓
`overflow_eos.cc`	EOS	✓

Table 2 shows cross-platform analysis for integer overflows. The evaluation data is available at `https://njaliu.github.io`.

5 RELATED WORK

Security problems of smart contracts have been widely discussed in recent years [11–13, 16]. Luu *et al.* highlighted four types of vulnerabilities for smart contracts [13]. Tsankov *et al.* proposed a verification technique [16], which transforms Ethereum smart contracts into Datalog logics [6]. Permenev *et al.* further presented their solution to verify smart contracts in a inductive manner [14]. In addition to security problems, Liu *et al.* proposed a statistical approach to identify potential code smells as well [12]. While this family of works mainly focused on Ethereum smart contracts, Seraph aims at enabling cross-platform security analysis to address more real-world concerns. Furthermore, the symbolic semantic graph (SSG) is related to program dependency graphs (PDG) which were widely discussed in the community of programming languages [7–9, 15]. Particularly, SSG can be considered as a lightweight variant of PDG which defines general blockchain semantics.

6 CONCLUSION

In this work, we highlighted the Seraph tool, a cross-platform security analyzer for blockchain smart contracts based on the EVM and WASM runtime. Specifically, Seraph enables a general abstraction for the interaction between virtual machines and their host blockchains via *connector API*. Moreover, Seraph introduced *symbolic semantic graph* as a lightweight representation for security analysis. In the preliminary evaluation, we applied Seraph in analyzing smart contracts on four existing blockchain platforms.

7 ACKNOWLEDGMENTS

Lei Wang is supported by National Key Research and Development Program of China No. 2018YFB0803400 and No. 2019YFB2101601.

REFERENCES

[1] 2019. FISCO-BCOS. https://fisco-bcos.org/.
[2] 2019. Microsoft Z3 SMT Solver. https://z3.codeplex.com/.
[3] 2019. Mythril. https://github.com/ConsenSys/mythril.
[4] 2019. Not so smart contracts. https://github.com/crytic/not-so-smart-contracts.
[5] 2019. SWC Registry. https://swcregistry.io.
[6] Thomas Eiter, Georg Gottlob, and Heikki Mannila. 1997. Disjunctive datalog. *ACM Transactions on Database Systems (TODS)* 22, 3 (1997), 364–418.
[7] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[8] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.
[9] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and enforcing security guarantees via program dependence graphs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 291–302.
[10] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
[11] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: finding reentrancy bugs in smart contracts. In *ICSE (Companion)*. ACM, 65–68.
[12] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for Ethereum smart contracts. In *ASE*. ACM, 814–819.
[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.
[14] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2019. Verx: Safety verification of smart contracts. *Security and Privacy* 2020 (2019).
[15] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 49–61.
[16] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *arXiv preprint arXiv:1806.01143* (2018).
[17] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014).