S-GRAM: Towards Semantic-Aware Security Auditing for Ethereum Smart Contracts

Han Liu*†
School of Software
Tsinghua University
Beijing, China
liuhan2017@tsinghua.edu.cn

Chao Liu
Peking University
Beijing, China
liuchao_cs@pku.edu.cn

Wenqi Zhao
Ant Fortune Business Group
Ant Financial
Beijing, China
muhan.zwq@antfin.com

Yu Jiang‡
School of Software
Tsinghua University
Beijing, China
jy1989@tsinghua.edu.cn

Jiaguang Sun
School of Software
Tsinghua University
Beijing, China

ABSTRACT

Smart contracts, as a promising and powerful application on the Ethereum blockchain, have been growing rapidly in the past few years. Since they are highly vulnerable to different forms of attacks, their security becomes a top priority. However, existing security auditing techniques are either limited in finding vulnerabilities (rely on pre-defined bug patterns) or very expensive (rely on program analysis), thus are insufficient for Ethereum.

To mitigate these limitations, we proposed a novel *semantic-aware security auditing* technique called S-GRAM for Ethereum. The key insight is a combination of N-gram language modeling and lightweight static semantic labeling, which can learn statistical regularities of contract tokens and capture high-level semantics as well (*e.g.*, flow sensitivity of a transaction). S-GRAM can be used to predict potential vulnerabilities by identifying irregular token sequences and optimize existing in-depth analyzers (*e.g.*, symbolic execution engines, fuzzers *etc.*). We have implemented S-GRAM for Solidity smart contracts in Ethereum. The evaluation demonstrated the potential of S-GRAM in identifying possible security issues.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; • **Security and privacy** → *Software security engineering*; • **Theory of computation** → Program analysis;

*Also with Beijing National Research Center for Information Science and Technology.
†Also with Key Laboratory for Information System Security, Ministry of Education.
‡Correspondence author

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240728

KEYWORDS

Smart contracts, security auditing, language modeling, static semantic labeling

ACM Reference Format:
Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-GRAM: Towards Semantic-Aware Security Auditing for Ethereum Smart Contracts. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3238147.3240728

1 INTRODUCTION

In recent years, smart contracts have been introduced to enable more flexible application scenarios than Bitcoin [16]. Generally, smart contracts are a special form of computer programs that respond to blockchain transactions. However, due to the nature of smart contracts as programs, they are highly vulnerable to various types of security attacks. We use the simplified scenario in Figure 1 to explain the DAO attack in June 2016. Specifically, an attacker identifies a victim contract with a vulnerable function, *i.e.*, `transfer` (Step 1). He or she further deploys a contract to exploit the vulnerability, *i.e.*, `()` fallback function (Step 2). Next, the attacker calls `transfer`. When executing the money transfer operation at line 1 before the balance updating at line 2, `transfer` calls the fallback function (Step 3). The fallback function calls `transfer` again to still more money (Step 4).

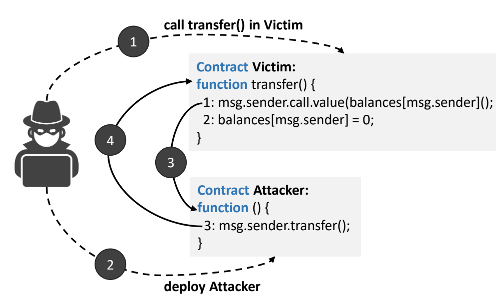


Figure 1: A simplified scenario of DAO attack.

To address the security issues, *rule-based* and *program analysis-based* auditing solutions have been proposed. Unfortunately, the former is limited in handling unknown patterns and the latter can be expensive thus unscalable in practice. In this paper, we focus on the Ethereum blockchain and highlighted the task of *efficient semantic modeling of smart contracts*. However, to fulfill this task is challenging due to the balance between accuracy and scalability. We outlined the main challenges below.

Challenge 1: Model Ethereum Mechanism. Ethereum has defined a set of special mechanisms, *e.g.*, gas system, data storage *etc.*, which must be considered.

Challenge 2: Encode Storage Access. Smart contracts in Ethereum are stateful, with *storage* holding the state data. Finding security issues often requires reasoning about access on storage data. Efficiently encoding the accesses without loosing too much accuracy becomes an important problem.

Challenge 3: Identify Flow Sensitivity. Transactions are flow sensitive, *i.e.*, security-critical operations are vulnerable under specific control flow conditions. Yet, the flow sensitivity analysis can hardly scale.

Semantic-Aware Security Auditing. In this paper, we have proposed S-GRAM, a *semantic-aware security auditing* technique for Solidity smart contracts in Ethereum to address the aforementioned challenges. The key insight behind S-GRAM is a combination of N-gram based language modeling and lightweight static contract analysis. Leveraging S-GRAM to identify security issues enables fast and scalable auditing since the language model is trained only once and the auditing is reduced to calculating probabilities. During the model training, S-GRAM uses lightweight static analysis to generate semantic meta data (*e.g.*, access dependency, flow sensitivity *etc.*) and further help the model learn more semantic regularities, *e.g.*, money transfer is always wrapped by a predicate associated with `msg.sender`. Then, based on a S-GRAM language model, we can predict potential vulnerabilities by identifying irregular contract token sequences. Moreover, S-GRAM can interface to existing in-depth analyzers as an optimization pass. We have implemented S-GRAM and evaluated it on Ethereum contracts. The results demonstrated the potential of S-GRAM in accurately identifying security issues.

2 BACKGROUND

2.1 Ethereum Smart Contracts

A smart contract is a special form of programs at a specific *address* on blockchain. In our setting, we focus on Ethereum smart contracts written in Solidity language [9]. Besides, a contract address includes its own storage (*i.e.*, permanent state data) and a amount of "Ether" balance (*i.e.*, Ethereum cryptocurrency). Moreover, Solidity provides developers with a variety of APIs to implement specific business logic, *e.g.*, send money to some address or retrieve the blockchain information. We summarize the major uniqueness of Solidity smart contracts below.

Data Location Data is stored in different locations. By default, state variables and function local variables are stored in *storage*. Function parameters are stored in *memory*.

Entry Point Every public callable function of a contract is a valid entry point. That is, no explicit main entry exists.

Exception Handling Exceptions cause the execution of a smart contract to stop and all the side effects get reverted. With exceptions manifested by APIs as `call`, `send` and `delegatecall`, a `false` returns and execution continues.

Gas System In Ethereum, every instruction consumes a specific *gas* value. Programmers are allowed to specify a gas limit by explicitly using *gas()* API for a transaction.

2.2 Statistical Language Model

A statistical language model (SLM) is a probability distribution on different sequences of *words*. SLM based techniques have been widely applied in software engineering tasks [11, 13, 17, 20, 21]. Mathematically, for a token sequence $s = t_1 t_2 \cdots t_n$, SLM estimates its probability as a production of a series of *conditional probability*, $P(s) = P(t_1) \cdot \prod_{i=2}^{n} P(t_i|t_1, \cdots, t_{i-1})$. The *N-gram* model is further introduced to approximate the computation by considering only a limited prefix with length N. Therefore, $P(t_i|t_1, \cdots, t_{i-1})$ is reduced to counting the occurrences below

$$P(t_i|t_1, \cdots, t_{i-1}) = \frac{count(t_{i-(n-1)} \cdots t_{i-1}, t_i)}{count(t_{i-(n-1)} \cdots t_{i-1})} \qquad \text{(N-gram)}$$

Based on *N-gram*, to better interpret the probability of a program $s = t_1 t_2 \cdots t_n$ as a sequence of tokens, we use the measurement *perplexity* or its log-transformed version *cross-entropy* [15], which is defined as $H_{\mathcal{M}}(s) = -\frac{1}{n} \log p_{\mathcal{M}}(t_1 \cdots t_n)$. According to the *N-gram* model ($n = k$), the formula amounts to

$$H_{\mathcal{M}}(s) = -\frac{1}{n} \sum_{1}^{n} \log p_{\mathcal{M}}(t_i \mid t_{i-k+1} \cdots t_{i-1}) \qquad \text{(Perplexity)}$$

Given a program, SLM can estimate its perplexity via performing the aforementioned calculation. In the context of smart contracts, we highlighted the possibility of associating statistical regularities (SLM perplexity) with the distribution of bugs. To this end, an SLM should be sensitive to "bug-relevant perplexity" rather than "bug-irrelevant perplexity", which is commonly caused by application-specific data, coding convention *etc.*.

3 SEMANTIC-AWARE SECURITY AUDITING

3.1 General Framework

In this section, we introduce the general framework of S-GRAM, as shown in Figure 2. S-GRAM works in a two-phase manner, *i.e.*, model construction phase and security auditing phase, respectively. In the model construction phase, the input is a large collection of smart contract corpus. Given a contract from the corpus, a *Static Analyzer* performs lightweight analysis to generate semantic metadata, *e.g.*, access dependency and transaction flow sensitivity. Then, a *Tokenizer* parses the contract into a token sequence with the semantic metadata labeled. Next, S-GRAM enables the N-gram based *Training Engine* to train an S-GRAM language model.

In the security auditing phase, the input is a audit target smart contract. Similarly, the contract is parsed into a token sequence by the *Static Analyzer* and the *Tokenizer*. Based on the S-GRAM language model, a *Detector* scans the token sequence to identify "irregular" subsequences, *i.e.*, ones with high perplexity *w.r.t.* the S-GRAM language model, as candidate vulnerabilities [18, 22]. Furthermore, S-GRAM enables a *Ranker* to sort all functions based on a "security

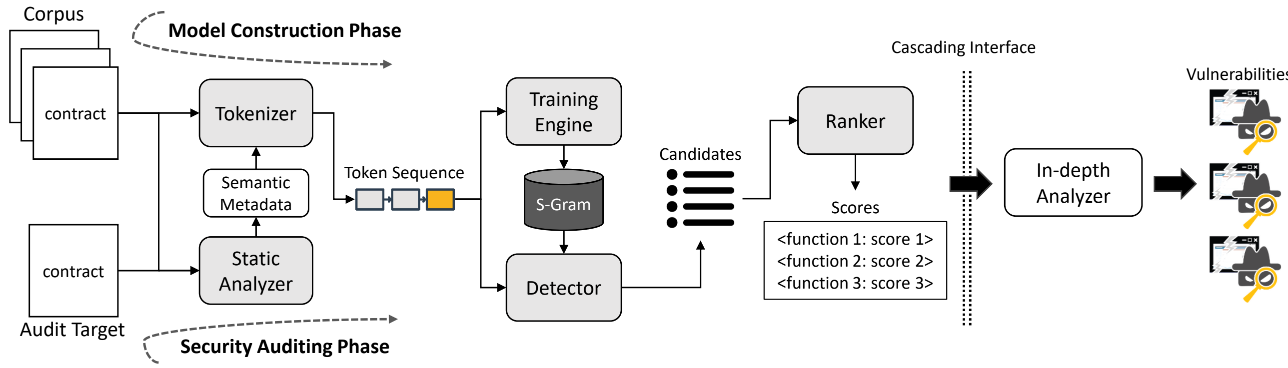


Figure 2: The general S-GRAM framework.

score". With the ranking, S-GRAM can interface to an existing in-depth analyzer in a smarter way, *e.g.*, forcing a symbolic execution engine to execute functions as ranked.

3.2 Semantic Metadata Generation

Given a smart contract, S-GRAM first performs a lightweight static analysis to generate semantic metadat, *i.e.*, storage access dependency and flow sensitivity in our setting. Next, we describe the details of semantic metadata generation using a contract in Figure 3 (we replace brackets with colons to save space).

```
1  contract Reward:
2    uint prize;
3    address owner;
4    modifier costs(uint _prize):
5      require(msg.value <= _prize);
6      _;
7    function Reward(uint _prize):
8      prize = _prize;
9      owner = msg.sender;
10   function update(uint _prize) public { prize = _prize; }
11   function reward(address recv) public costs(prize):
12     require(prize != 0);
13     if(msg.sender == owner) {
14       recv.call.value(prize)();
```

Figure 3: An example Solidity smart contract used for explaining semantic metadata generation

Encode Storage Access. Storage data is persistent across transactions, thus can greatly influence the behavior of smart contracts. In Figure 3, the contract `Reward` has two state variables `prize` and `owner`, which are initialized in the constructor (line 7-9). Analyzing accesses on storage enables effective security auditing of smart contracts. For instance, when seeing a transaction call to the function `reward` with the `prize` storage value to be `10`, a malicious miner can post and prioritize another transaction to the `update` function which sets `prize` to `0` (line 10) and pose a denial-of service (DoS) attack by failing the check at line 12.

However, reasoning about the accesses on `prize` is not easy. Specifically, we must consider its assignment (line 10) and path conditions (line 5 and 12). In S-GRAM, we propose an efficient and abstract way to encode accesses on storage data, which is based on the *transaction dependency* relation (denoted as D_t). More formally, we define an storage access event $e = \langle a, x, t \rangle$, where a is the storage address, $x \in \{W, R\}$ indicates whether the access is a write or read operation, $t \in \{R, L\}$ specifies whether the storage value is dependent on other possible transactions (R, short for *Remote*) or not (L, short for *Local*). Given two storage access events e_1 and e_2, we use F_1 and F_2 to denote two sets of public functions that can reach them, and C_1 and C_2 to represent two sets of path conditions for e_1 and e_2. In Figure 3, `prize` accesses at line 10 and line 12 have $F_1 = \{\texttt{update}\}$ and $F_2 = \{\texttt{reward}\}$. C_1 and C_2 are ϕ. For any pair of storage accesses $e_1 = \langle a_1, x_1, t_1 \rangle$ and $e_2 = \langle a_2, x_2, t_2 \rangle$, we define that e_1 is *transaction-dependent* on e_2 (vice versa), *i.e.*, $\langle e_1, e_2 \rangle \in D_t$ below.

$$
\begin{gathered}
(a_1 = a_2 \wedge (x_1 = W \vee x_2 = W)) \;\wedge \\
(\exists f_1 \in F_1 \wedge f_2 \in F_2,\; f_1 \neq f_2 \;\vee \\
(f_1, c_1) \in C_1 \wedge (f_2, c_2) \in C_2 \wedge f_1 = f_2 \wedge c_1 \neq c_2)
\end{gathered}
$$

Conceptually, if e_1 is dependent on e_2, the storage data accessed by e_1 and e_2 may differ in two transaction scenarios, *i.e.*, $e_1 e_2$ and $e_2 e_1$. In that case, we label the t_1 and t_2 fields of both e_1 and e_2 as R (*Remote*), otherwise L (*Local*). In the contract of Figure 3, the access on `prize` at line 10 is a write operation and the access at line 12 is a read operation. Since they are reached by function `update` and `reward` respectively, they are *transaction-dependent* on each other and marked as *Remote*. The `owner` state variable is only accessed at line 13, thus marked as *Local*.

Identify Flow Sensitivity. Furthermore, S-GRAM identifies critical operations (*e.g.*, storage accesses, money transfer *etc.*) and abstracts their flow conditions as well. To this end, we aim at inferring potential connections between critical operations and their flow conditions, *e.g.*, a secure money transfer is often guarded by a sanity check on the address of the transaction sender.

Given a critical operation s, we use $c_1 c_2 \cdots c_n$ to denote all the flow conditions. We use $A(c)$ to hold a set of addresses involved in c and $O(c)$ to include a collection of operators associated with the storage data in c. Then *w.r.t.* s, the overall address set $\mathcal{A}(s)$ and operator set $O(s)$ include all its flow conditions. In Figure 3, with s to be the call operation at line 14, its $\mathcal{A}(s) = \{\texttt{msg.sender}\}$

and $O(s) = \{$`<=`, `!=`, `==`$\}$. Based on these two sets, S-GRAM can statistically learn a probability distribution on how flow conditions relate to critical operations, *e.g.*, function calls on addresses are more likely to follow address-based flow conditions, transaction-dependent storage accesses always follow equivalence-based (*e.g.*, `!=` and `==`) flow conditions.

3.3 Tokenization

With the semantic metadata generated, S-GRAM performs a tokenization process which parses smart contract code into token sequences. Particularly, the parsing is realized by traversing the abstract syntax tree (AST) of a contract in a type-based manner, *i.e.*, tokens are generated *w.r.t.* specific types of AST nodes. Figure 4 shows the AST[1] of the `reward` function in Figure 3. Each node has a specific type, *e.g.*, `CallExpr`, `BinaryExpr`, `ID` *etc.*. Each node value is a *lexeme* of the contract, *e.g.*, `msg`, `address`, `0` *etc.*.

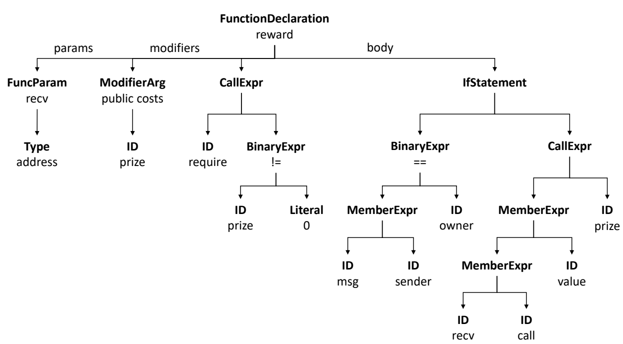


Figure 4: The AST of the `reward` function in Figure 3.

Type-based Tokenization. In S-GRAM, the contract tokenization process is done in a type-based manner so as to capture high-level semantics. Given (e, t) to be an AST node where e is the lexeme value and t is the AST type, the extracted token tk follows the rules described in Table 1.

Specifically, the first group of rules are designed for storage accesses with `UnaryExpr`, `AssignExpr` and `BinaryExpr`. In these cases, S-GRAM generates two tokens, *i.e.*, cf:$O(e_u)$ which contains all the operators in its flow conditions and $t[T(a_u)x_u t_u]$ which attaches access event information. The second group of rules targets at function calls. In cases of `send` and `transfer`, S-GRAM creates a token `call_min_gas`. In terms of `call` and `value` calls, `call_all_gas` is generated since the call will forward all the available gas. Other calls are parsed into `call_normal`. As for arguments of the `value` and `gas` call, we firstly generate a prefix *i.e.*, `eth:` and `gas:` respectively. Then, we use the lexeme for `Literal` arguments and data type for `ID` arguments. Remaining rules are specified to handle other special AST types. To implement the tokenization, we defined a stateful *AST Visitor* in S-GRAM, which traverses the AST and employ the rules to generate corresponding tokens. Based on a large sequence of generated tokens, S-GRAM leverages well-designed *N-gram* toolkits to build the language model and set N empirically.

[1] We use the AST defined in solidity-parser [8]

3.4 Prediction

Vulnerability prediction in S-GRAM is realized via identifying the irregular token sequences in the contract, *i.e.*, with low probability *w.r.t.* the S-GRAM language model. Given a function f, S-GRAM collects all the possible token sequences into a set $T = \{t_1, t_2, \cdots, t_m\}$ and computes a probability $prob_{\mathcal{M}}(t_i)$ for each sequence *w.r.t.* $\mathcal{M}$. Based on a prediction size of K (the maximum length of token sequence considered), we use *Predicted*(T, K, M) to denote the K sequences in T with the least M probabilities, and potential set of vulnerabilities as well. In our evaluation, we set K as a variable and explored how K can affect the efficacy of S-GRAM. In addition, S-GRAM leverages several pre-defined rules to filter false positives. Specifically, for a sequence t containing only scope tokens, *e.g.*, `function_begin` and `function_end`, S-GRAM directly throws it. If t and its subsequence t' are both flagged as potential vulnerabilities, S-GRAM only reports one of them.

3.5 Ranking

Through vulnerability prediction, we can identify a group of potential security issues. In order to interface to in-depth analyzers, S-GRAM generates a ranking on contract functions to help explore contracts more efficiently. Given a function f and language model $\mathcal{M}$, we calculate a probability $prob_f = \frac{1}{n}\sum_{i=1}^{n} prob(t_i)$ where t_i is a token sequence within f. Furthermore, we count the number N_f of token sequences of f that are in the *Predicted* set. Based on $prob_f$ and N_f, S-GRAM computes a security score $Score_f$ for f as a linear function $Score_f = a * prob_f + \frac{b}{M} * N_f$ (a and b are parameters which can be automatically learned via labeled data) and further ranks all functions.

4 EMPIRICAL EVALUATION

4.1 Dataset and Setting

We have implemented S-GRAM into a security auditing tool called Ether★. The S-GRAM language model was trained via the KenLM [2] library. The training set of S-GRAM language model was collected from the `Etherscan` repository [1], including 43,553 deployed open source contracts. The testing set contains 1,500 smart contracts. Evaluation data is publicly available at https://github.com/njaliu/sgram-artifact. We selected Oyente [14] to confirm vulnerabilities. and used the pure N-gram approach (consider only lexemes) as a baseline comparison with S-GRAM.

4.2 Empirical Results

We first investigated the security auditing capability *w.r.t.* different configurations of S-GRAM. Figure 5 showed the number of real vulnerabilities found by Ether★ with prediction size $K = 20$. Regarding different S-GRAM configurations, Ether★ managed to find 3.32 to 6.94 vulnerabilities in the top 20 flagged potential security issues on average. That said, S-GRAM is able to generate a small but effective set of candidate vulnerabilities in practice. In terms of six differently configured S-GRAM models, *i.e.*, $N = 2 \cdots 7$, $N = 5$ performed the best while $N = 2$ is the worst. This can be explained as: *bigram* failed to capture the majority of statistical regularities in smart contracts. Therefore, we constructed S-GRAM language model using *5-gram*.

Table 1: Type-based tokenization rules. $T(a)$ gets the type of the variable pointing to the storage address a.

Value of tk	Rules of tokenization
cf:$O(e_u)$ $t[e][T(a_u)x_u t_u]$	t is `UnaryExpr`, whose access event $e_u = \langle a_u, x_u, t_u \rangle$
cf:$[O(e_l), O(e_r)]$ $t[e][T(a_l)x_l t_l, T_{a_r} x_r t_r]$	t is `AssignExpr` or `BinaryExpr`, whose left and right access events are $e_l = \langle a_l, x_l, t_l \rangle$ and $e_r = \langle a_r, x_r, t_r \rangle$
cf:$[\mathcal{A}(e_c)]$ call_min_gas	t is `CallExpr`, whose call event is e_c and callee is `send` or `transfer`
cf:$[\mathcal{A}(e_c)]$ call_all_gas	t is `CallExpr`, whose call event is e_c and callee is `call` or `value`
cf:$[\mathcal{A}(e_c)]$ call_normal	t is `CallExpr`, whose call event is e_c and callee does not include `require`, `assert` and fit into above
eth:e	t is `Literal`, (e, t) is an argument of `value CallExpr`
eth:$T(e)$	t is `ID`, (e, t) is an argument of `value CallExpr`
gas:e	t is `Literal`, (e, t) is an argument of `gas CallExpr`
gas:$T(e)$	t is `ID`, (e, t) is an argument of `gas CallExpr`
sol:ts	t is `ID`, e is either `now` or `block.timestamp`
sol:unit	t is `Literal` and e is one of the following values: `wei`, `finney`, `szabo`, `ether`, `seconds`, `minutes`, `hours`, `days`, `weeks` or `years`
sig:e	t is `String`, (e, t) is an argument of `call CallExpr`
t	t is `Modifier` or `Literal`, *e.g.*, 100, "token"
e	otherwise

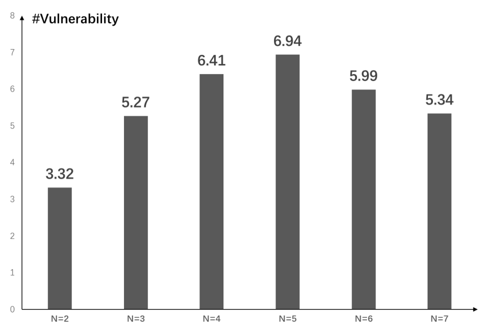
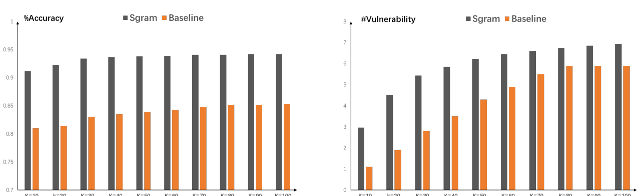


Figure 5: X-axis: value of N. Y-axis: the number of vulnerabilities found by Ether*. Prediction size: $K = 20$.



(a) Auditing accuracy **(b) Number of vulnerabilities**

Figure 6: Comparison between S-GRAM and Baseline.

Next, we further explore how the prediction size K related to the vulnerability detection capability of Ether*, as shown in Figure 6b (left bar). With the increase of prediction size, Ether* managed to find more vulnerabilities. However, the growth rate becomes slower from small to large K values. Moreover, we conducted comparison experiments to compare S-GRAM and the baseline approach, *i.e.*, N-gram based technique. Results are shown in Figure 6. Specifically, Figure 6a displayed the auditing accuracy of both techniques. Under different prediction sizes, S-GRAM achieved an accuracy from 91.2% to 94.2% while the baseline can only climb to 85.3%. Furthermore, S-GRAM outperformed the baseline approach by finding 169.1% more problems.

Table 2: Performance of cascading analysis with Ether*. Time unit: second. Opt: optimization

Contract	LOC	ReGuard	Ether*	Opt
DWorldDeed	2144	23.49	16.10	31.46%
CanReclaimToken	2148	26.11	16.74	35.89%
usingOraclize	2219	31.73	19.23	39.39%
Court	2869	39.08	11.02	71.80%
EtherToken	3257	48.12	9.87	79.49%

4.3 Cascading In-depth Analysis

In the evaluation, we combined Ether* with ReGuard [12], a fuzzer designed for identifying reentrancy vulnerabilities in Solidity smart contracts. Specifically, we used Ether* to rank functions and further optimize transaction sequence generation in ReGuard. Table 2 summarized the performance of with and without Ether*. Using Ether*, ReGuard became more efficient when auditing smart contracts in all cases. Time saved by Ether* spans from 31.46% to 79.49% *w.r.t.* the five contracts picked.

5 RELATED WORK

Smart Contract Analysis. Smart contracts have been attracting increasing research interests during the past several years, especially in the contextof security. Atzei *et al.* investigated known attacks on smart contracts and highlighted a classification of typical bugs and vulnerabilities [6]. Hildenbrandt *et al.* have designed the KEVM for Ethereum with formal semantics in the K language and support for a set of security analysis [10]. To find bugs in smart contracts, Bhargavan *et al.* introduced a general framework which converts smart contracts to the F^* language programs for formal verification [7]. Abraham *et al.* defined the notion of Effectively Callback Free objects so as to enable modular reasoning on callback operations of smart contracts [3]. Luu *et al.* proposed Oyente, a bug finder based on the symbolic execution technique to detect predefined bug patterns [14]. Focusing on reentrancy attacks, Liu *et al.* introduced ReGuard to fuzz testing smart contracts [12].

Statistical Language Models of Code. Software code resembles natural languages. Hindle *et al.* defined the statistical regularities of software as naturalness [11]. Nguyen *et al.* built a language model using *sememes* which carry more semantic information than pure lexemes [17]. While a language model can capture global statistical characteristics, Tu *et al.* proposed a *cache language model* to include local programming patterns that are specific to personal projects [21]. Based on language models, Allamanis *et al.* introduced techniques to learn coding conventions [4]. Raychev *et al.* and Allamanis *et al.* proposed to predict names for both variables, methods and classes [5, 19]. Liu *et al.* highlighted a stochastic technique to optimize program obfuscation based on statistical language models [13]. In the context of debugging and bug finding, Yu *et al.* leveraged the *N-gram* model to improve software fault localization with a special focus on GUI applications [23]. Ray *et al.* investigated the naturalness of buggy code and used the entropy measurement to predict defects in a project [18]. Wang *et al.* further extended the approach via training code at a higher level, *e.g.*, statements and method calls, in order to capture semantic bugs more effectively [22]. Compared to existing techniques, S-GRAM introduced a novel language model for smart contracts which is designed to efficiently capture domain-specific semantics.

6 CONCLUSION

In this paper, we present the S-GRAM semantic-aware security auditing technique for Ethereum smart contracts. Specifically, S-GRAM highlighted the insight that statistical abnormality is very much likely to indicate the existence of vulnerabilities. S-GRAM first performs static semantic metadata generation and type-based tokenization to prepare token sequences and construct a statistical language model. Next, S-GRAM enumerates and ranks all possible token sequences of a contract to be analyzed, then flags those with least probabilities as potential vulnerabilities. We have prototyped S-GRAM as Ether★ and evaluated it on Ethereum contracts. Ether★ achieved an over 90% accuracy in identifying different types of potential vulnerabilities. Furthermore, Ether★ managed to uncover several previously unknown security issues and improved the efficiency of a Solidity fuzzer as well. In the future, we plan to extend S-GRAM on other blockchain ecosystems with different statistical language models.

ACKNOWLEDGMENT

We thank anonymous reviewers for their comments on the paper. We thank Zhiqiang Yang for helping prepare the evaluation datasets. This research is sponsored in part by NSFC under Grant No.: 61527812, National Science and Technology Major Project under Grant No.: 2016ZX01038101, MIIT IT funds (Research and application of TCN key technologies) of China, National Key Technology R&D Program under Grant No.: 2015BAG14B01-02, and China Postdoctoral Science Foundation under Grant No.: 2017M620785.

REFERENCES

[1] [n. d.]. Etherscan. https://etherscan.io/.
[2] [n. d.]. KenLM. https://github.com/kpu/kenlm.
[3] ITTAI ABRAHAM, GUY GOLAN-GUETA, YAN MICHALEVSKY, NOAM RINETZKY, and YONI ZOHAR. [n. d.]. Online Detection of E ectively Callback Free Objects with Applications to Smart Contracts. ([n. d.]).
[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 281–293.
[5] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 38–49.
[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *International Conference on Principles of Security and Trust*. Springer, 164–186.
[7] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, A Rastogi, T Sibut-Pinote, N Swamy, and S Zanella-Beguelin. 2016. Formal verification of smart contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS'16*. 91–96.
[8] ConsenSys. 2018. Solidity Parser in Javascript. https://github.com/ConsenSys/solidity-parser
[9] Ethereum Foundation. 2018. The solidity contract-oriented programming language. https://github.com/ethereum/solidity.
[10] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine.* Technical Report.
[11] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. 837–847.
[12] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *Proceedings of the 40th International Conference on Software Engineering Companion*. IEEE Press.
[13] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Stochastic optimization of program obfuscation. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 221–231.
[14] 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.
[15] Christopher D Manning and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. Vol. 999. MIT Press.
[16] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Consulted* (2008).
[17] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A Statistical Semantic Language Model for Source Code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. 532–542.
[18] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 428–439.
[19] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. 111–124.
[20] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 419–428.
[21] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the Localness of Software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 269–280.
[22] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 708–719.
[23] Zhongxing Yu, Hai Hu, Chenggang Bai, Kai-Yuan Cai, and W Eric Wong. 2011. GUI software fault localization using N-gram analysis. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*. IEEE, 325–332.