# Verification of Ethereum Smart Contracts: A Model Checking Approach

Tam Bang, Hoang H. Nguyen, Dung Nguyen, Toan Trieu, and Tho Quan

***Abstract*—*Ethereum smart contracts* based on *blockchain* technology are powerful and promising applications that provide a global platform for exchanging cryptocurrencies and public services. This technology are garnering a huge impact and is widely adopted in the current times as it can transform the way we transfer and exchange value by passing the need for a middleman and reducing cost. These *smart contracts also* represent a basis for true ownership of digital assets and a wide range of decentralized applications. Besides this, since *Ethereum* and its *smart contracts* are a publicly accessible, unchangeable and distributed platform, they are extremely vulnerable to various forms of attack, with their security becoming a top priority. However, current security-verifying programs tend to provide many technical details which are pretty hard for normal people to understand briefly. To tackle this problem, we designed a process aiming to mitigate these limitations, with our key insight being a combination of *semantic* structure analysis and *symbolic execution* on *control-flow graph*s (CFG for short). This article proposes a new approach for auditing *Ethereum smart contracts*, applying this technique would benefit both average *users without any technical knowledge* and security experts as well.**

***Index Terms*—*Ethereum smart contracts*, *semantic* structure analysis, *symbolic execution*, *control-flow graph*.**

## I. Introduction

In recent years, technologies and computer ecosystems have evolved tremendously, which cause many positive impacts on modern societies. From the Internet of Things to artificial intelligence and also *blockchain* technology, they have shown to be applicable in many fields including financial industries [1], cross-industry [2] and public sector [3]. In addition to this, *blockchain* seems to be one of the most disruptive technologies because its mechanics are likely to have more influence on high-tech industries over the next few years [4].

The *blockchain* is not a new technology; however, this technology has gained a great effect in this decade. This is a huge step forward in decentralized systems and distributed applications. "It's about thinking about the current architectural landscape and strategies to move immutable distributed databases" [4]. To eliminate the need for trusted third parties, *blockchain* was developed to work on a peer-to-peer network that implements its peers to agree on the trading transactions.

While the first generation of *blockchain* was designed only to solve cryptocurrencies problems, *Ethereum*, one of the most popular current systems, focuses on implementing decentralized computing approaches [5]. One new prominent of these reliable platforms is to enable *smart contracts*, which can automatically execute on the *blockchain* and enforced by the consensus protocol [6]. Accordingly, *smart contracts* are likely to apply in a wide range of fields including ownership of copyrights, financial instruments, document existence and asset tracking for the Internet-of-Things [7].

The increased adoption of *smart contracts* demands strong security guarantees. Unfortunately, it is challenging to create *smart contracts* that are free of security bugs. As a consequence, critical vulnerabilities in *smart contracts* are discovered and exploited every few months [8], [9]. Moreover, we have to require not only the security but also the correctness of executions, to keep *smart contracts* more secure. In fact, adversaries may take advantage of undocumented methods and exploit potential bugs as well as vulnerabilities in the contracts, which can cause harm to users. One of the most successful attacks is “The DAO” in June 2016, which exploited the “call to unknown functions” and “reentrancy” vulnerabilities and managed to steal from a contract around $50M at the time of the attack [10]. More precisely (in Fig. 1), an attacker identifies a victim contract with a vulnerable function, i.e., transfer (function used to send “Ether”) at step 1. He or she will deploy a contract to exploit the vulnerability, which is the fallback function (step 2). Then the attacker call transfer. When executing the money transfer operation at line 1 before updating the balance at line 2, transfer calls the fallback function (Step 3). The fallback function calls transfer again to still more money (Step 4). [11]. More recently, $31M worth of ether was stolen due to a critical security bug in a digital wallet contract [12].

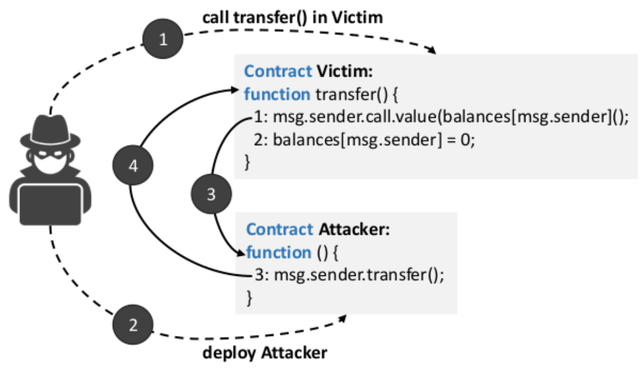
Hence, verifying smart contract behaviors and solving security issues are extremely crucial and challenging when *blockchain* technologies evolve with much diversity across their ecosystems.

The next sections of this paper are organized as follows. Section II describes states of the art about *smart contracts* audit procedures. Section III recalls some basic concepts of *smart contracts*. Section IV proposes our works to verify *Ethereum smart contracts*. Section V illustrates the execution of our audit procedures. More extensive experiments, knowledge about AST, CFG and *symbolic*

Manuscript received October 11, 2019; revised March 5, 2020.

The authors are with the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Vietnam (e-mail: {bnbaotam, nhhoang.sdh19, 1652119, 1652614, qttho}@hcmut.edu.vn).

*execution* also report in this section. Section VI outlines some challenges related to the application of *blockchain* and *smart contracts* for auditing. Finally, Section VII concludes the paper and proposing avenues for future research.



```
Contract Victim:
function transfer() {
  1: msg.sender.call.value(balances[msg.sender]();
  2: balances[msg.sender] = 0;
}

Contract Attacker:
function () {
  3: msg.sender.transfer();
}
```

Fig. 1. A simplified scenario of DAO attack [11].

## II. Related Work

*Smart contracts* are how things get done in the *Ethereum* ecosystem. When someone wants to get a particular task done in *Ethereum*, they initiate a smart contract with one or more people.

Smart contract security audits are fundamentally the same as the regular code audit, which is meticulously investigating code to explore security flaws and vulnerabilities before the code is publicly deployed.

Many decentralized applications, which are centered around *Smart contracts*, have implemented a variety of software tools to aid in the auditing practice. These tools, such as automated code-checking for vulnerabilities, may be used as a supplement, but should not replace the formal auditing process. One option, as mentioned previously, is Mythril [13], which can be used for detecting uint overflows and underflows. Another tool is Etherscrape[13], used here to scrape live *Ethereum* contracts for reentrancy bugs when send() is being used [13]. Or, Securify can analyses security violations of contracts on a bytecode level through *semantic* inference [14], whereas SmartCheck parses the contract language for lexical and syntax analysis [15].

However, a simple smart contract with no business logic costs around $4000. More complicated and advanced *smart contracts* can go from 50,000$ all the way up to 100,000$. Plus, if that's not enough, there will usually last 4 weeks and then it takes 8 weeks for the auditing process to be completed [16]. Moreover, if the user are non-technical individuals, they may not understand the results of smart contract audit even if they read the report carefully.

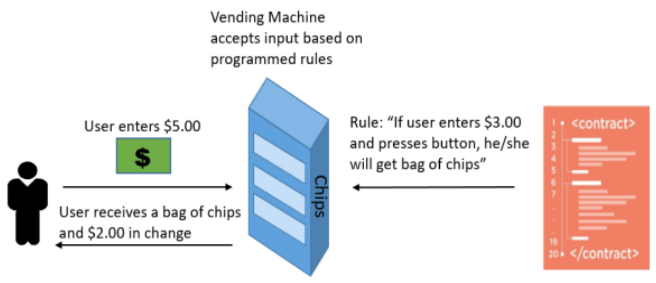

Vending Machine accepts input based on programmed rules
User enters $5.00
$
User receives a bag of chips and $2.00 in change
Chips
Rule: "If user enters $3.00 and presses button, he/she will get bag of chips"
<contract>
</contract>

Fig. 2. Szabo's example [6].

## III. Preliminaries

In this section, we assume that the reader is familiarised with the *blockchain* concept and how it operates.

### A. First Declaration of Smart Contract

Szabo introduced *smart contracts* for the first time as a "computerized transaction protocol that executes the terms of a contract" [6]. He suggested translating contractual clauses into code, and embedding them into a property that can self-enforce them, so as to minimize the need for trusted intermediaries between transacting parties [17]. Fig. 2 shows an example of *smart contracts* using the Vending machine. A kid has 5$ and tries to buy a bag of chips, so he puts his money to the Vending machine. The Machine programmed with some rules and it meets the requirement in this situation. So he receives a bag of chips with his change.

### B. Ethereum Smart Contracts

Although this was an innovation in the early 1990s, *smart contracts* did not thrive during that period, as an authorized trusting third party was necessary to monitor the terms and the execution of the encoded contracts, which poses the risk that a contracting party may not meet its contractual obligations [18]. With *blockchain* technology, the implementation of *smart contracts* becomes achievable and responsibilities are distributed to the participating nodes [5].

A smart contract is a special form of programs at a specific address on *blockchain* technology. They are self-executing with specific instructions written on their code which gets executed when a certion condition is made. In our framework, we use *Ethereum smart contracts* written in Solidity language due to *Ethereum* is the most popular blockchain platform for creating *smart contracts* [19]. A contract address also includes its own storage (i.e., state data) or an amount of "Ether" balance (i.e., *Ethereum* cryptocurrency). Moreover, Solidity supports a variety of APIs to implement specific business logic for developers, e.g., transfer money to some address or get the *blockchain* information. Fig. 3 illustrates a home buying between two people using *Ethereum smart contracts*.

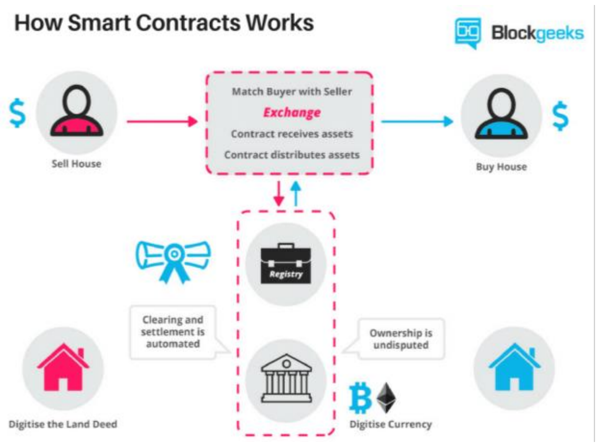


Fig. 3. How *smart contracts* work [16].

## IV. Proposed Method

Investigating security issues underlying the design and implementation of such decentralized *smart contracts* is our primary goal. We also focus on providing a clear understanding of audit information for even non-tech users. As shown in Section I, our formal approach consists mainly of four primary steps:

1) We first define several states which are understandable by common logic to express *semantic*s in implementation processes. After executing a processing call, the states representing core implementations of *smart contracts* are able to be gained through traversing abstract syntax tree (AST).

2) For code scripts of *Ethereum smart contracts*, we organize relevant Solidity contract classes involved and construct call graphs of all these classes and the CFG for particular methods.

3) Next, through CFG information, we identify potentially suitable methods by using *symbolic execution* techniques and slice these graphs to store relevant statements while maintaining their connectivity in those sliced graphs. Then, data-flow and control-flow analysis of the methods based on these graphs is performed to produce likely dependence relations among the objects and function calls needed for invoking each of these methods.

4) Finally, we compare directly the states achieved from step 1 and other states gained by invoking *symbolic execution* on the CFGs from step 3. If these states are fully compatible, this contract is more liable to be trusted. If not, it could be considered an unreliable contract and should not be deployed on any *blockchain*.

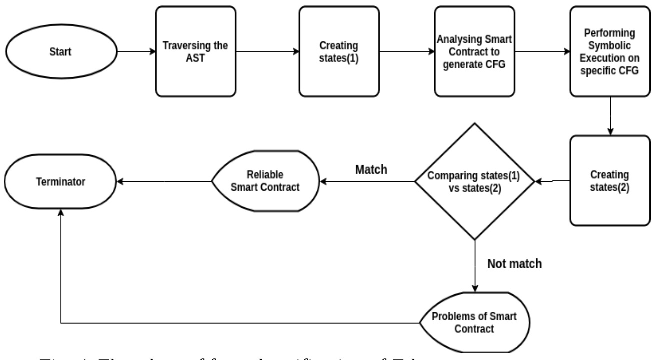

Start
Traversing the AST
Creating states(1)
Analysing Smart Contract to generate CFG
Performing Symbolic Execution on specific CFG
Creating states(2)
Comparing states(1) vs states(2)
Match
Reliable Smart Contract
Terminator
Not match
Problems of Smart Contract

Fig. 4. Flowchart of formal verification of *Ethereum smart contracts*.

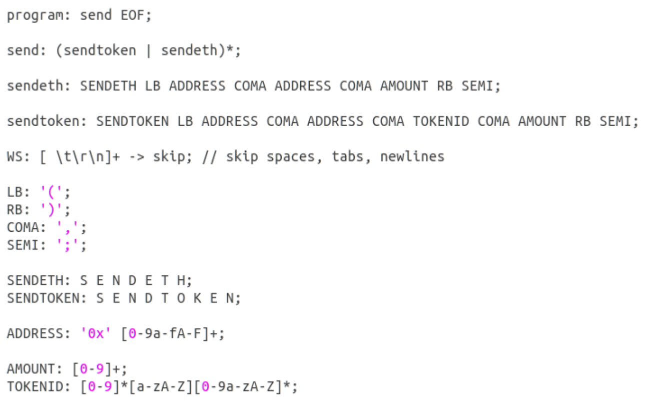
## V. Formal Verification of *Ethereum Smart Contracts*

In this section, we attempt to further clarify four steps of verifying *Ethereum smart contracts* were mentioned in Fig. 4. Besides, some useful libraries such as Antlr4 [20], [21] are also utilized for our detailed implementation

### A. Context-Free Grammar & AST Traversal

First of all, we define the internal structures of a program which includes various syntaxes. Based on these kinds of structures, we generate parse trees representing syntactic structures of a string according to some Context-Free Grammar. Then, abstract syntax trees of the parse trees, which are removed some tokens for faster compilation time, are able to be built.

Secondly, we attempt to scan all addresses appearing in the contract and store temporarily the initial states before traversing to generate any new state. In addition to this, some Depth First Search algorithms can be applied to traverse the abstract syntax trees (program→functions→parameters) and a new state could be generated through any change of each statement of a corresponding address. Our final result is a set of all states of a structure, and this result is able to be used for comparison in the following steps.

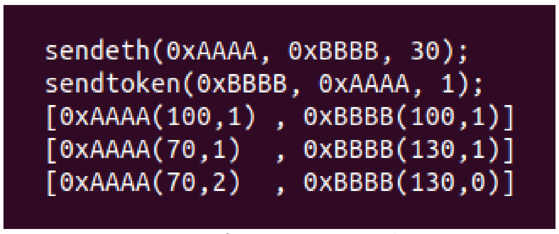


```
program: send EOF;

send: (sendtoken | sendeth)*;

sendeth: SENDETH LB ADDRESS COMA ADDRESS COMA AMOUNT RB SEMI;

sendtoken: SENDTOKEN LB ADDRESS COMA ADDRESS COMA TOKENID COMA AMOUNT RB SEMI;

WS: [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines

LB: '(';
RB: ')';
COMA: ',';
SEMI: ';';

SENDETH: S E N D E T H;
SENDTOKEN: S E N D T O K E N;

ADDRESS: '0x' [0-9a-fA-F]+;

AMOUNT: [0-9]+;
TOKENID: [0-9]*[a-zA-Z][0-9a-zA-Z]*;
```

Fig. 5. Context-Free Grammar of two structures: sendeth & sendtoken.

For instance, we develop a system to trade tokens and ether. Accordingly, we first define some syntaxes as AMOUNT, which only have numerical value, or a string begin with “0x” and follow that is another string include number, characters from “a” to “f” (also capital character) is called ADDRESS. Based on that, we specify two structures of our system, the first structure is called sendtoken(<sender>, <receiver >,<amount>); and the second one is sendeth(<sender>, <receiver>, <amount>); Each structure of parameters can describe as follows:

- Sender and receiver are addresses of senders or receivers on *Blockchain*.
- An amount is a number of token or ether.
- Sendtoken send an amount of token from sender to receiver.
- Sendeth send an amount of ether from sender to receiver.



```
sendeth(0xAAAA, 0xBBBB, 30);
sendtoken(0xBBBB, 0xAAAA, 1);
[0xAAAA(100,1)  , 0xBBBB(100,1)]
[0xAAAA(70,1)   , 0xBBBB(130,1)]
[0xAAAA(70,2)   , 0xBBBB(130,0)]
```

Fig. 6. Executing process for generating states by traversing AST.

In this example, we have two initial states:

1) 0xAAAA(100,1): 0xAAAA has 100 ether and 1 token.
2) 0xBBBB(100,1): 0xBBBB has 100 ether and 1 token.

After executing sendeth(0xAAAA, 0xBBBB, 30), the 0xAAAA’s balance decrease 70 ether while the amount of ether of 0xBBBB address increase from 100 to 130. Then, the final state indicate that 0xBBBB transfer 1 token to 0xAAAA’s account by invoking sendtoken(0xBBBB, 0xAAAA, 1).

## B. Generating CFG of Smart Contracts

In computer science, a *control-flow graph* is a representation of all paths which might be traversed of a program during its execution. Many static analysis techniques approach to optimize and verify the *semantics* of programs through these graphs.

In detail, Slither is the first open-source framework written in Python 3, and it supports static analysis for Solidity and visualizes detail information of *smart contracts*. Furthermore, it also provides APIs to write custom analyses easily. Thus, we aim to use this framework to generate *control-flow graphs* of Solidity *smart contracts* in DOT format, which is able to visualize via GraphViz [22] or to convert to PNG image files.

Considering the example in Section V.A, in this step, we build *smart contracts* describing a way to exchange Ether and token through Ethereum smart contracst. However, to simplify the problem, we suppose to initialize two objects A and B with 100 ether and 1 token each account. The sendeth and sendtoken methods perform to switch the amount of ether from A to B, and the amount of token from B to A.

```
pragma solidity ^0.5.8;

contract SimpleContract {
    uint etherA = 100;
    uint etherB = 100;
    uint tokenA = 1;
    uint tokenB = 1;
    bool check = false;

    function sendeth(uint amountE) public returns (string memory) {
        if(etherA > amountE) {
            etherA = etherA - amountE;
            etherB = etherB + amountE;
            check = true;
            return "Success";
        }
        else
            return "Fail";
    }

    function sendtoken(uint amountT) public returns(string memory) {
      if(tokenB > amountT && check = true) {
            tokenB = tokenB - amountT;
            tokenA = tokenA + amountT;
            return "Success";
        }
        else
            return "Fail";
    }
}
```

Fig. 7. SimpleContract performs sendeth and sendtoken functions.

After scanning through the source code, we generate corresponding *control-flow graphs* of every vital method of this smart contract, for instance:

For more details, the sendeth function first checks the condition. If it passes, the process will move to excute the following expressions, and then return “Success” when it is done. On the other hand, this process will stop after returning “Fail”.

## C. Symbolic Execution

*Symbolic execution* is a means of analyzing a program to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

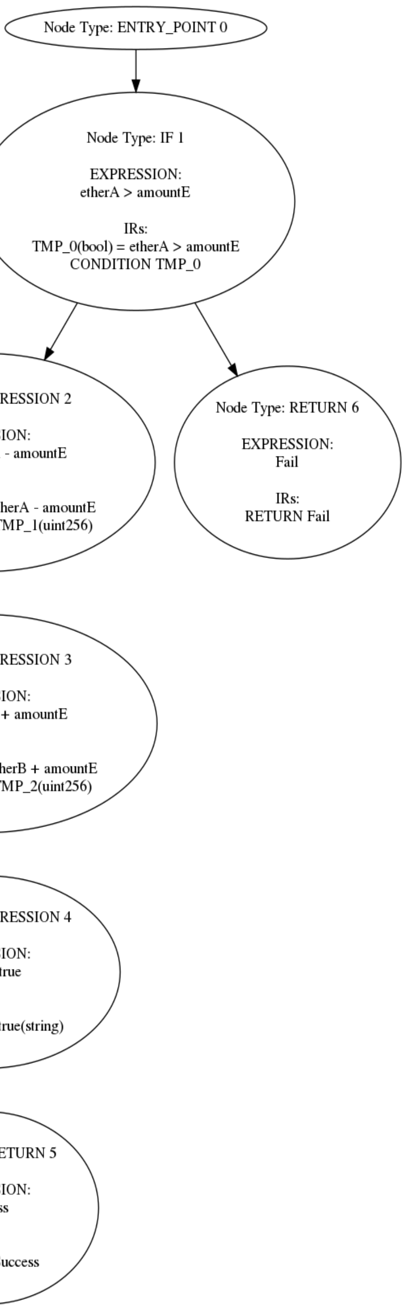
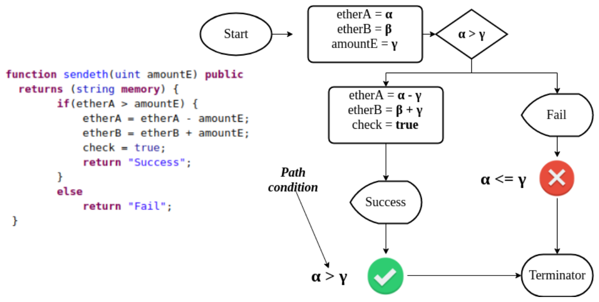


Fig. 8. CFG of sendeth method.

To give a clear example, in SimpleContract, the CFG of “sendeth” function (similar to “sendtoken”) corresponding to it in Fig. 3, reads in values and returns Fail if the amountE is greater than etherA.

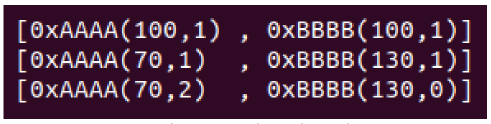


```
function sendeth(uint amountE) public
  returns (string memory) {
        if(etherA > amountE) {
            etherA = etherA - amountE;
            etherB = etherB + amountE;
            check = true;
            return "Success";
        }
        else
            return "Fail";
 }
```

Fig. 9. *Symbolic execution* with sendeth function based on CFG.

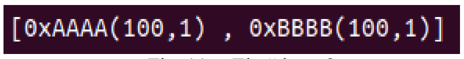
During *symbolic execution*, symbolic state maps variables to symbolic values (e.g., amountE assigned to $\alpha$, etherA to $\beta$ and etherB to $\theta$). When reaching the if statement, α, β could take any value, and *symbolic execution* can, therefore, proceed along both branches, by "forking" two paths. Each path gets assigned a copy of the program state at the branch instruction as well as a path constraint. In this example, the path constraint is $\beta > \alpha$ for the then branch and $\beta \leq \alpha$ for the else branch. Both paths can be symbolically executed independently. When paths terminate (e.g., as a result of return Fail or simply existing), *symbolic execution* computes a concrete value for $\alpha$, $\beta$ by solving the accumulated path constraints on each path.

In this example, the constraint solver would determine that in order to reach the statement: return "Fail", α would need to be smaller than β. In addition, all of the above procedures can be followed through CFG. Conclusively, after performing *symbolic execution* on CFG with amountE = 30 and amountT = 1, we have some of the following states:



```
[0xAAAA(100,1) , 0xBBBB(100,1)]
[0xAAAA(70,1)  , 0xBBBB(130,1)]
[0xAAAA(70,2)  , 0xBBBB(130,0)]
```

Fig. 10. "Then" branch.



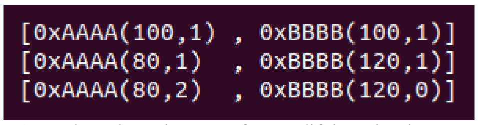
```
[0xAAAA(100,1) , 0xBBBB(100,1)]
```

Fig. 11. "Else" branch.

### D. Comparing States

We compare the results from step 1 and step 3 (all states that can be achieved by the CFGs, which help us do not skip any cases). Assuming a token's price is 30 ether and just focus on the "then" branch because the states of the "else" branch are similar to the initial states (Fig. 6 and Fig. 11). Thus, there are two possibilities:

- User of 0xAAAA want to buy 1 token from user of 0xBBBB, so 0xAAAA address pays 30 ether and then receive 1 token. Following that, this smart contract executed exactly the same as the formal logic, so it is reliable and deployable on a public *blockchain*.
- On the other hand, considering a new situation, the smart contract is modified by erasing line 25 of its source code. Then, the results of the *symbolic execution* are completely different, especially in the second and the third state (Fig. 6 and Fig. 12). In this case, even if 0xAAAA sent 30 ether to 0xBBBB, the smart contract would not return any token, and the owner of address 0xAAAA would also receive nothing. Thus, this smart contract is more likely to be unreliable and should not be deployed on any *blockchain*.



```
[0xAAAA(100,1) , 0xBBBB(100,1)]
[0xAAAA(80,1)  , 0xBBBB(120,1)]
[0xAAAA(80,2)  , 0xBBBB(120,0)]
```

Fig. 12. "Then" branch states after modifying SimpleContract.

## VI. Challenges

"There will be further bugs, and we will learn further lessons; there will not be a single magic technology that solves everything". - Vitalik Buterin [5]. So, there are many challenges to overcome in the adoption of the auditing methods, such particular problems are:

- A set of special mechanisms which should be taken into account in the *Ethereum* network, for instance, gas & data storage, identify flow sensitivity, and exception handling.
- The privacy and security of *blockchain* technologies.
- The scalability of the *blockchain* and the flexibility of audit procedures.
- The impact of verifying processes on the user's decisions.

We believe that the continued integration of *blockchain*s could bring new business models and assist us in improving the existing audit systems. Furthermore, a prominent mechanism could also be implemented for handling most of the above challenges.

## VII. Conclusion and Future Works

In this research, we propose a new process for auditing *Ethereum smart contracts*. First, our process generates states by using AST traversing with some syntaxes to describe formal logic that everyone can understand easily. Then we create CFGs and perform *symbolic execution* on them. After that, we get some other new states and start comparing it with the first ones to give conclusions. Certainly, applying this technique would benefit both average *users without any technical knowledge* and security experts as well. For the case of average users, they can scan new contracts before transferring to ensure that their cryptocurrencies are not diverted to any adversary address. In contrast, security experts exploit our study to quickly investigate the suspicious breaches inside *smart contracts* of the *Ethereum* platform. Further work could be conducted on discovering unknown vulnerabilities or integrating them with dynamic analysis. Furthermore, we can build applications based on *blockchain* technology and apply our approach to verify *smart contracts* before users agree to use them.

## Conflict of Interest

The authors declare no conflict of interest.

## Author Contributions

Prof. Tho Quan developed the theoretical formalism, Prof. Tho Quan and Nguyen H. Hoang supervised the work. Tam Bang worked out almost all of the technical details. Tam Bang, Hoang H. Nguyen processed the experiment data, performed the analysis and wrote the paper with input from all authors. Tam Bang, Dung Nguyen, Toan Trieu contributed to the design and implementation of the research to the analysis of the results. All authors had approved the final version.

## Acknowledgment

This research is funded by Ho Chi Minh University of Technology (HCMUT) under Grant To-KHMT-2019-07 and supported by Vietnam Blockchain Corporation. We also thank Prof. Huynh Tuong Nguyen at the Ho Chi Minh City University of Technology for providing valuable feedback on this paper.

## References

[1] K. Wang and A. Safavi. (2017). Blockchain is empowering the future of insurance. [Online]. Available: https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/

[2] Ethlance. (2017). [Online]. Available: http://ethlance.com/

[3] A. Irrera, "Northern trust uses blockchain for private equity record-keeping," *Reuters*, 2017.

[4] R. Modi, *Solidity Programming Essentials: A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain*, Birmingham: Packt Publishing, 2018.

[5] B. V. Ethereum, "A next generation smart contract & decentralized application platform," White Paper, 2014.

[6] N. Szabo, *Smart Contracts*, 1994.

[7] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, p. 1, 2016.

[8] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer, "An in-depth look at the parity multisig bug," *Ethereum Parity Wallet Security*, 2017.

[9] P. Bylica. (2017). How to find $10M just by reading the blockchain. [Online]. Available: https://medium.com/golem-project/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95

[10] D. Siegel, "Understanding the DAO hack for journalists," *CoinDesk*, 2016.

[11] H. Liu, C. Liu, W. Zhao *et al.*, "Towards semantic-aware security auditing for Ethereum smart contracts," in *Proc. the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 814–819.

[12] H. Qureshi, "A hacker stole $31m of ether - how it happened, and what it means for Ethereum," *Free Code Camp*, 2017.

[13] ConsenSys, "Security tools," *Ethereum Smart Contract Best Practices*, 2018.

[14] P. Tsankov, A. Dan, D.-C. Dana *et al.*, "Securify: Practical security analysis of smart contracts," in *Proc. the ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[15] S. Tikhomirov, E. Voskresenskaya, *et al.*, "Smartcheck: Static analysis of Ethereum smart contracts," in *Proc. 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[16] Blockgeeks. (2018). Why are smart contract security audits so important? [Online]. Available: https://blockgeeks.com/smart-contract-security-audits/

[17] N. Szabo. (1997). The idea of smart contracts. [Online]. Available: http://szabo.best.vwh.net/smart_contracts_idea.html

[18] T. I. Kiviat, "Beyond bitcoin: Issues in regulating blockchain transactions," *DukeLaw Journal*, vol. 65, pp. 569–569, 2015.

[19] Ethereum Foundation. (2018). The solidity contract-oriented programming language. [Online]. Available: https://github.com/Ethereum/solidity

[20] Terence Parr. (2018). ANTLR 4 (Another Tool for Language Recognition). [Online]. Available: https://github.com/antlr/antlr4

[21] Crytic. (2018). Slither, the solidity source analyzer. [Online]. Available: https://github.com/crytic/slither

[22] AT&T Labs Research. (1991). Graphviz - graph visualization software. [Online]. Available: https://www.graphviz.org/7

**Tam Bang** received the B.E. in computer science in 2018 from Ho Chi Minh City University of Technology. He became a master student in computer science at Ho Chi Minh City University of Technology (HCMUT), Vietnam in 2018. His current research interests include *blockchain* and big data. He also has experience in AI.

**Hoang H. Nguyen** received his bachelor's degree in electronics and telecommunications from Ho Chi Minh City University of Science (HCMUS), Vietnam in 2013. He became a master student in computer science at Ho Chi Minh City University of Technology (HCMUT), Vietnam in 2014. He was a visiting student for one year in the School of Information Systems, Singapore Management University. His current research interests include programming languages, program analysis, and mobile security. He also has experience in *blockchain* security development.

**Tho Quan** is an associate professor in the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Vietnam. He received his B.Eng. degree in information technology from HCMUT in 1998 and received Ph.D. degree in 2006 from Nanyang Technological University, Singapore. His current research interests include formal methods, program analysis/verification, the *Semantic* Web, machine learning/data mining and intelligent systems. Currently, he heads the Department of Software Engineering of the Faculty. He is also serving as the vice dean of Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology.

**Dung Nguyen** is currently studying for the B.E in computer science in Ho Chi Minh City University of Technology. His current research interests are blockchain and software development.

**Toan Trieu** is currently studying for the B.E in computer science in Ho Chi Minh City University of Technology. His current research interests are security and software development.