An Improved Gas Efficient Library for Securing IoT Smart Contracts Against Arithmetic Vulnerabilities

JingHuey Khor
University of Southampton Malaysia
No.3, Persiaran Canselor 1,
Kota Ilmu, Educity@Iskandar,
79200 Iskandar Puteri, Johor,
Malaysia
J.Khor@soton.ac.uk

Mansur Aliyu Masama
University of Southampton
Electronics and Computer Science,
University Road,
SO17 1BJ, Southampton,
United Kingdom
ma3u18@soton.ac.uk

Michail Sidorov
Department of Computer Science and Engineering,
Toyohashi University of Technology,
1-1 Tempaku-cho, Toyohashi,
Aichi 441-8580, Japan
michail.sidorov.cl@tut.jp

WeiChung Leong
University of Southampton
Electronics and Computer Science,
University Road,
SO17 1BJ, Southampton,
United Kingdom
wcl1g16@ecs.soton.ac.uk

JiaJun Lim
University of Southampton
Electronics and Computer Science,
University Road,
SO17 1BJ, Southampton,
United Kingdom
jjl1n14@ecs.soton.ac.uk

ABSTRACT
Public blockchains targeting Internet of Things (IoT) are gaining more traction every day with majority of them being built on top of the Ethereum infrastructure. However, a growing number of these blockchains introduces security issues. There are 525 entries already in the Common Vulnerabilities and Exposure database related to Ethereum smart contracts. 479 of them are related to arithmetic errors, which include integer overflow or underflow. This paper, thus, concentrates on analyzing arithmetic vulnerabilities found in existing public blockchains targeted at IoT applications. Furthermore, the performance in terms of security and gas cost of smart contracts is analyzed with and without SafeMath library. In addition, an improved SafeMath library is proposed that has better arithmetic coverage and requires lower gas consumption. Four security tools are used to analyze the arithmetic protection of the improved SafeMath library. The results show that the improved SafeMath library is able to cover 4 more arithmetic operations compared to the original one by using only two common conditions checks and is capable of saving 26 units of gas, which is a significant amount in the long run.

CCS Concepts
• Security and privacy→ Software and application security → Software security engineering

Keywords
Ethereum; IoT; SafeMath; Smart Contract

1. INTRODUCTION
Numerous public blockchains exist focused on running decentralized applications (dApps) for the Internet of Things (IoT). These include, but not limited to IOTA, VechainThor, Waltonchain, Helium, FOAM, etc. Among these blockchains, several ones are built on top of the Ethereum blockchain e.g. Atonomi, OriginTrail, BlockMesh, Fysical, Grid+, PowerLedger, etc. These blockchains are built based on the Ethereum Request for Comments 20 (ERC20), which is the most common standard used for implementing tokens via smart contracts [1].

Smart contracts are usually written in Turing-complete languages, such as Serpent, Solidity, Lisp Like Language (LLP), Viper, etc [2-6]. Among these language, Serpent is a low-level language and lacks safety features [7,8]. Viper is the newest language and is currently in an experimental phase. LLP lacks an easy verification mechanism and is not widespread [9]. Thus, Solidity is the most widely adopted language and is claimed to have the best interoperability with JavaScript application program interface (API) [8]. Therefore, this paper focuses on the analysis of smart contracts written in Solidity.

A reliable and secure dApp for IoT ideally requires a smart contract with minimum or no bugs at all. However, it is common for Solidity developers to make mistakes in writing smart contracts, due to their complexity as a Turing complete language [2]. Since many developers might be familiar with other languages before being introduced to Solidity, they might use certain coding patterns that are harmful if used in Solidity [10]. The bad practices or mistakes in writing smart contracts enable attackers to perform attacks to exploit benefits from it. Several known attacks or bugs in Ethereum smart contracts are race conditions, denial of service, timestamp dependence, integer overflow and underflow, etc [11]. These publicly disclosed vulnerabilities and threats are recorded and stored inside a Common Vulnerabilities and Exposure (CVE) repository. From the CVE, there are 525 entries that relate to Ethereum smart contacts at the time of writing, with CVE-2018-20421 being the most recent and CVE-2017-14457 being the oldest entry. From the total number of entries, 479 entries from them belong to arithmetic vulnerability. This vulnerability occurs when a fixed size variable is coded to store a number that is outside of its data type range, which causes an overflow or underflow condition that

ICSCA 2020, February 18–21, 2020, Langkawi, Malaysia

ACM ISBN 978-1-4503-7665-5/20/02…$15.00
https://doi.org/10.1145/3384544.3384577

gives incorrect results. In order to mitigate integer overflow or underflow in smart contracts written in Solidity, OpenZepplin's SafeMath library has been proposed by developers to be included in a smart contract. From the 479 entries in CVE, 78% of smart contracts perform arithmetic operations without the use of SafeMath library. The remaining 22% do use SafeMath library, however, they are still facing arithmetic issues due to inconsistent usage of the mathematic operations in the contracts. Other than that, all smart contracts directly include arithmetic operations defined in the SafeMath library in their contracts content, instead of using import keyword to import the library.

Therefore, there is a need to understand arithmetic vulnerabilities found in public smart contracts aimed at IoT applications and find possible ways to mitigate them. The following lists the main contributions of this paper:

1. Detailed analysis of arithmetic vulnerabilities found in smart contracts of existing IoT related public blockchains is performed
2. Performance of SafeMath library is analyzed in terms of mathematic operation coverage and gas cost
3. An improved SafeMath library is designed and its security and gas cost performance is analyzed and compared to the original SafeMath library

2. SECURITY ANALYSIS OF IOT SMART CONTRACTS

There are six well known Ethereum smart contracts related to IoT applications as introduced in previous section. However, OriginTrail and Grid+ smart contracts are published as bytecode, which is not readable and therefore is not analyzed in this section. From the remaining four smart contracts as shown in Table 1, PowerLedger is the only smart contract that does not implement SafeMath library for performing arithmetic operations. Although Atonomi, Blockchmesh, and Fysical have included SafeMath library in their smart contracts, all of them are using an outdated version, where *assert()* function is used. If the check using *assert()* function is false, it reverts the changes made. However, even if the changes are reverted, the function has already consumed all of the remaining gas. Gas is a fee that is consumed when transaction is made, and is priced in Gwei. Therefore, gas is wasted although transaction is reverted. Furthermore, these smart contracts check for the same condition twice, one is checked by the smart contracts, and another one is checked by SafeMath library. SafeMath library has provided security check on arithmetic operations, thus it is redundant to check the same condition again in the smart contract. As shown in Figure 1, *transfer()* function in the Atonomi smart contract uses *require()* function to check for the amount to be transferred is smaller or equal to the balance of sender. However, this check is redundant because this *transfer()* function uses *sub()* function from SafeMath library, and *sub()* function itself is doing the same security check.

Table 1. Arithmetic vulnerabilities analysis of IoT tokens

Token	SafeMath library	Check invariants function	Redundant code
PowerLedger	No	-	No
Atonomi	Yes	Assert()	Yes
BlockMesh	Yes	Assert()	Yes
Fysical	Yes	Assert()	Yes

```
function transfer(address _to, uint256 _value)
public returns (bool) {
      require(_to!=address(0));
      require(_value <=balances[msg.sender]);
      balances[msg.sender]-=_value;
      balances[_to]+=_value;
      Transfer(msg.sender, _to, _value);
       return true;
      }
      return false;
}
```

Figure 1. *transfer()* function taken from the Atonomi smart contract

PowerLedger is currently the most active public blockchain for IoT applications, with a total number of 406154 transactions made during the period of 340 days, i.e. since the time the smart contract was deployed up to the day of writing. This means PowerLedger has around 1195 transactions per day. Since this project is getting more popular compared to other IoT aimed blockchains, it was decided to analyze its security in more detail.

PowerLedger is a peer to peer energy market that enables users to trade energy. The transaction number is believed to increase with increased deployment of solar panels. PowerLedger is built using ERC20 token standard, which integrates a set of ERC20 functions within its own contract. These functions include *totalSupply()*, *balanceOf()*, *allowance()*, *transfer()*, *approve()* and *transferFrom()*[1]. Among these functions, *transfer()* function uses one subtraction and one addition operations, and *transferFrom()* function uses two subtractions and one addition operation. Transaction fee can be calculated using the following equation [12]:

$$Transaction\ fee = Gas\ price * Transaction\ gas * Ethereum\ price$$

Among the 40615 transactions, the lowest transaction fee in gas is 21470 gas, which is to perform *transfer()* function. With a gas price of 3e-8 Ether or 30 Gwei, the transaction fee is 6.44e-4 Ether, or equivalent to 0.12 USD, using the exchange rate of 1 Ether equal to 185.98 USD. This represents that a total of 4873.80 USD has been spent to transfer value from one account to another. However, the exchange rate of ETH is constantly fluctuating, at one point reaching approximately 1386 USD for 1 ETH on 12th of January 2018. Therefore, if using this as the exchange rate of ETH to calculate the total transaction cost, it will increase to 36269 USD, which is a significant amount.

Same as other ERC20 based smart contracts, this paper shows that PowerLedger also has two main security vulnerabilities, namely arithmetic vulnerability and unprotected function.

1. Arithmetic vulnerability

PowerLedger smart contract does not use SafeMath library to prevent arithmetic vulnerabilities, but it does include a certain condition check in its code. For example, in order to prevent an integer underflow, *transfer()* function checks if balance of the calling contract is more than the desired transfer value as shown in Figure 2(a). If the calling contract has balance which is more than the desired transfer value, the calling contracts balance will be deducted with the transferred value, and the receiver balance

will be added with the equivalent value. However, there is no security check on the receiver side. Therefore, there might be a possibility where integer overflow occurs, which causes the addition result to be smaller than the original balance. Same applies to *transferFrom()* function in Figure 2(b), it only performs security check for the calling contract, where balance of calling contract must be higher than the transferred value and the allowed transfer value must be higher than the desired transferred value. However, same as *transfer()* function, due to lack of security check on the receiver side, there is a possibility of integer overflow occurring.

2. Unprotected function

In a PowerLedger smart contract, *if()* function is used both in *transfer()* and *transferFrom()* functions to check for calling contracts balance. If calling contract balance is higher than the desired transfer value, it allows the transfer activity to commence and returns true, otherwise it returns false. Therefore, if the desired transfer value is higher than the calling contracts balance, although there is no transfer activity performed, gas is still consumed and wasted to perform the transaction. Therefore, this unprotected function vulnerability causes the loss of funds for end users. The *transferFrom()* function provides better security protection compared to *transfer()* function, where it also checks for the number of allowed transfer tokens. Although PowerLedger encourages end users to use the *transferFrom()* function instead of a *transfer()* function, the transaction record stored on the PowerLedger blockchian shows that *transfer()* function is mostly used. Therefore, instead of using *if()* function, *require()* function should be used to allow the transaction to roll back if *require()* condition is false. In this case, SafeMath library can be implemented to prevent arithmetic vulnerabilities, and at the same time, gas can be saved from a failed transaction.

```
function transfer(address _to, uint256 _value)
public returns (bool) {
if (balances[msg.sender]>=_value){
      balances[msg.sender]-=_value;
      balances[_to]+=_value;
      Transfer(msg.sender, _to, _value);
       return true;
      }
      return false;
}
```

(a)

```
function transferFrom(address _from, address _to,
uint256 _value) public returns (bool) {
if (balances[_from]>=_value &&
allowed[_from][msg.sender]>=_value){
      balances[_from]-=_value;
      allowed[_from][msg.sender]-=_value;
      balances[_to]+=_value;
      Transfer(_from, _to, _value);
       return true;
      }
      return false;
}
```

(b)

Figure 2. PowerLedger smart contract's (a) *transfer()* function (b) *transferFrom()* function

3. PERFORMANCE ANALYSIS OF SAFEMATH LIBRARY

Libraries in Solidity are meant to be deployed only once on the Ethereum blockchain. This is done so that other contracts could use them without the need of re-deployment. In addition, this also prevents code repetition. However, SafeMath library is not deployed on the Ethereum blockchain. This is done to allow its creator, OpenZeppelin, to fix bugs and issue updates. Therefore, in order to use the SafeMath library, it has to be obtained from OpenZeppelin webpage and copied to the intended smart contracts. However, once smart contracts are deployed, they are immutable. As a result, although some improvements are constantly made to the library, the already deployed smart contracts will not benefit from these updates.

Initially, *assert()* function was used in the SafeMath library to check for internal invariants in the contract. Thus, as described in previous section, the unnecessary gas usage makes *assert()* function inefficient. Therefore, SafeMath library was updated with an inclusion of *require()* function. If the check using *require()* function is false, it not only reverts the changes made, but also refunds the unused gas amount. Therefore, *require()* is considered as a more gas-friendly alternative compared to assert().

SafeMath library consists of five functions to provide secure mathematic operations, as shown in Table 2. This includes addition, subtraction, multiplication, division, and modulus. SafeMath library functions have two function parameters, which are the operands of the operations. Each function describes a mathematic operation and then performs a condition check. SafeMath library is using internal functions, which means all these functions are copied to the calling contract without the need of deploying them. In addition, SafeMath library uses pure functions to prevent modification or read from the state. These functions allow a state modification to revert in case of any errors via *require()* function.

Table 2. SafeMath library for 5 arithmetic operations

Description	Operation	Condition
Addition	$c = a + b$	$c \geq a$
Subtraction	$c = a - b$	$b \leq a$
Multiplication	$c = a * b$	$c/a == b$
Division	$c = a / b$	$b > 0$
Modulus	$c = a \% b$	$b != 0$

The deployment cost of a smart contract with the integration of a SafeMath library increases by around 228 gas on average, because the library code increases the smart contract size. However, its transaction cost is reduced by around 228 gas on average with this integration. This reduction is a result of some mathematic operations being outsourced to the SafeMath library. Although the increase of a deployment cost is significant for the integration of SafeMath library in a smart contract, it is paid only once during the deployment of a smart contract. In contrast, transaction cost is charged every time a transaction is performed. Thus, the running cost is lower.

4. IMPROVED SAFEMATH LIBRARY

Although the existing SafeMath library has been widely used in protecting against arithmetic vulnerabilities, the library itself has some disadvantages that include an incomplete coverage of mathematical operations. Therefore, if a new arithmetic operator is required for a smart contract, a new condition check that fulfills the specific arithmetic operation must be created. This might be a tedious task for a developer who has limited knowledge for new arithmetic operations. Therefore, this paper proposes an improved SafeMath library, which is able to provide better coverage of

mathematical operations with two common condition checks for arithmetic operations.

Arithmetic vulnerabilities can be categorized into two main categories, namely integer overflow and integer underflow. Therefore, instead of checking for a specific condition for different operations as shown in the original SafeMath library, the improved SafeMath library just uses two common condition checks to cover more arithmetic operations, as shown in Table 3. For arithmetic operations that produce result higher than or equal to their inputs, the improved SafeMath library will check on the condition where result should be equal or more than the second input, in order to prevent integer overflow. These arithmetic operations include addition, multiplication, exponentiation, and left shift. In contrast, the improved SafeMath library will check on the condition where result should be equal or less than the first input to prevent integer underflow for arithmetic operations that produce result lesser than or equal to their inputs. These arithmetic operations include subtraction, division, modulus, right shift and logarithm. Although it is impossible for integer underflow to occur for modulus and logarithm operations, this condition check still applies to these operations because their results should be less than their first operand. However, for some arithmetic operations, it is possible to get undefined answer when one of the operand is zero, such as division, modulus, logarithm, etc. This issue can be prevented by EVM itself because it does not allow operations that create undefined answers.

Table 3. Improved SafeMath library for 9 arithmetic operations

Description	Condition
$Addition$ $Multiplication$ $Exponentiation$ $Left\ Shift$	$c \geq b$
$Subtraction$ $Division$ $Modulus$ $Right\ Shift$ $Logarithm$	$c \leq a$

5. PERFORMANCE EVALUATION

The performance of the improved SafeMath library is evaluated based on gas consumption and arithmetic protection compared to the original SafeMath library.

5.1 Gas Cost Consumption

Although the improved SafeMath library has general checking conditions for arithmetic operations, it does not require additional gas cost for this change. In order to optimize the gas cost, the improved SafeMath library uses the following code to minimize the gas consumed:

1. Local variable is removed from the function content, but included in the function returns parameter list
2. Return statement is removed from the function

This optimization is able to save 2608 gas cost for deployment and 26 gas cost for transaction compared to the original SafeMath library.

This paper shows that PowerLedger smart contract with the improved SafeMath library integration is able to save 228 gas cost for transaction, which means it requires less 6.84e-6 ETH per transaction. It might seem insignificant, but it costs 51.67 USD less (using the exchange rate of 1 ETH equal to 185.98 USD) for the total of 406154 transactions. The transaction fee of a PowerLedger smart contract with the improved SafeMath library is further reduced by additional 26 gas. This means it saves additional 7.8e-7 ETH per transaction. The savings from the lower transaction cost will become more significant when the number of transactions increases drastically. For example, assuming PowerLedger has the same total transaction number as Ethereum blockchain, which is 577,320,489 for the time of writing this paper, the cost will be 83,748 USD less with the implementation of the improved SafeMath library in the smart contract. Therefore, if the improved SafeMath library is used in a PowerLedger smart contract, it will not only be able to patch the arithmetic vulnerabilities and unprotected function issues, but would also reduce the gas required to execute transactions.

5.2 Arithmetic Security Protection

To analyze arithmetic vulnerabilities of smart contracts, four security tools were chosen to analyze the performance of the improved SafeMath library in detecting arithmetic vulnerabilities. These four tools are SmartCheck [2], Securify [13], Mythril [14], and Oyente [15]. Remix is not chosen in this paper because it is a Solidity compiler that uses static analysis to analyze smart contracts. Therefore, certain tasks such as searching the chain are not possible with this tool. In contrast, Mythril solves the aforementioned Remix's issue using concolic testing, which is a hybrid analysis technique that performs symbolic execution along a concrete execution path [16]. Mythril and Oyente both use symbolic execution methods to analyze the correctness of smart contracts. Symbolic execution is an off-line computation that computes some approximation using formulas to represent program states. Symbolic execution is a powerful technique for discovering bugs; however it tends to result in false negatives because it does not guarantee to explore all program paths. In contrast, Securify is an abstract interpreter that explores all contract behaviors [13]. SmartCheck is an extensible static analysis tool that translates Solidity source code into an Extensible Markup Language (XML) based intermediate representation and checks it against XPath pattern. The coverage of each security tool can be found in their respective papers [13,2,17,14,15].

Existing smart contracts that have arithmetic vulnerabilities found in CVE are analyzed using the aforementioned four security tools, as shown in Table 4. This paper shows that Mythril and Oyente are the best security tools to detect arithmetic issues. Oyente analyzes smart contracts more in depth where it highlights the piece of code where integer overflow or underflow was allowed to happen. Although Mythril is also able to identify arithmetic issues, it might miss some true positives as it is using concolic analysis that examines only a subset of the execution [18].

In order to create a benchmark for this analysis, the aforementioned security tools were used to analyze smart contracts with the integration of the original SafeMath library. This paper shows that with the integration of SafeMath library in smart contracts that have arithmetic vulnerabilities, Mythil and Oyente are unable to detect any arithmetic issues. This is because SafeMath library is able to prevent integer overflow or underflow for these smart contracts. Same applies to the integration of the improved SafeMath library, these security tools are also unable to

detect integer overflow or underflow issues. Therefore, this proves that the improved SafeMath library is able to provide the same arithmetic protection as the original one, with only two general functions that cover at least 9 arithmetic operations.

Table 4. Security analysis of Improved SafeMath library using 4 security tools

Security tool	Original smart contract	With SafeMath library	With Improved SafeMath library
Mythil	Present	Absent	Absent
Oyente	Present	Absent	Absent
SmarkCheck	Not explicit	Not explicit	Not explicit
Securify	Not explicit	Not explicit	Not explicit

6. CONCLUSION

This paper has analyzed arithmetic vulnerabilities of the existing Ethereum smart contracts aimed at IoT applications. PowerLedger, which is the most active token, was found to have integer overflow. Therefore, the original SafeMath library was included in its smart contract. This paper further found that this inclusion allowed to save 228 gas cost for each transaction. However, the original SafeMath library uses 5 functions to cover 5 arithmetic operations and requires different condition check for each of these functions. Therefore, it is a challenging task to extend the original SafeMath library to cover other arithmetic operations. Thus, an improved SafeMath library was designed, presented and tested in this paper. It covers 9 arithmetic operations using only two common check conditions. The proposed library is able to further lower transaction cost by 26 gas compared to the original one. Therefore, the improved SafeMath library is proven to be able to provide a better arithmetic protection for smart contracts aimed at IoT applications with a lower transaction cost.

7. ACKNOWLEDGMENTS

This work was supported by Malaysian Ministry of Higher Education (MOHE) under Grant No. FRGS/1/2018/ICT04/USMC/02/1.

8. REFERENCES

[1] Vogelsteller, F., Buterin, V.. 2015. ERC-20 token standard. Ethereum.

[2] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y. 2018. SmartCheck: Static analysis of Ethereum smart contracts. In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 27 May-3 June 2018, pp. 9-16

[3] Buterin, V.2017. Vyper. DOI= https://vyper.readthedocs.io/en/latest/.

[4] Edgington, B. 2017. LLL Introduction. DOI= https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[5] Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.2015. A programmer's guide to Ethereum and serpent. DOI=https://www.cs.umd.edu/~elaine/smartcontract/guide.pdf.

[6] Ethereum: Solidity: 2016. Introduction to smart contracts. DOI= https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html.

[7] Buterin, V. 2017. PSA: I now consider Serpent outdated tech. DOI = https://twitter.com/vitalikbuterin/status/886400133667201024?lang=en.

[8] Castor, A. 2017. One of Ethereum's earliest smart contract languages is headed for retirement. DOI= https://www.coindesk.com/one-of-ethereums-earliest-smart-contract-languages-is-headed-for-retirement (2017)

[9] Wohrer, M., Zdun, U.: Smart contracts. 2018. Security patterns in the ethereum ecosystem and solidity. In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 20-20 March 2018, pp. 2-8.

[10] Diligence, C. 2018. Ethereum smart contract best practices. DOI=https://consensys.github.io/smart-contract-best-practices/known_attacks/

[11] NCC. 2018. Decentralized application security project - top 10. DOI=https://dasp.co/.

[12] Wood, G. 2019. Ethereum: A secure decentralised generalised transaction ledger Ethereum.

[13] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bunzli, F., Vechev, M. 2018. Securify: Practical security analysis of smart contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada.

[14] Diligence, C. 2018. Mythirl platform: Platform and ecosystem for Ethereum security tools. DOI=https://mythril.ai/files/whitepaper.pdf

[15] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A. 2016. Making smart contracts smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria.

[16] Sen, K., Marinov, D., Agha, G. 2005. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes* **30**(5), 263-272. DOI=10.1145/1095430.1081750

[17] Remix. 2018. Ethereum- IDE. DOI=https://remix.readthedocs.io/en/latest/

[18] Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B. 2018. Vandal: A scalable security analysis framework for smart contracts. CoRR