

석사학위논문
Master's Thesis

이오스 자원관리 시스템 취약점 분석

Who Spent My EOS? On the (In)Security of Resource
Management of EOS.IO

2020

이상섭 (Lee, Sangsup)

한국과학기술원

Korea Advanced Institute of Science and Technology

석사학위논문

이オス 자원관리 시스템 취약점 분석

2020

이상섭

한국과학기술원

전산학부

이오스 자원관리 시스템 취약점 분석

이상섭

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2019년 12월 17일

심사위원장 김 용대 (인)

심사위원 류석영 (인)

심사위원 손수엘 (인)

Who Spent My EOS? On the (In)Security of Resource Management of EOS.IO

Sangsup Lee

Major Advisor: Yong-Dae Kim
Co-Advisor: Soo-el Son

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Daejeon, Korea
December 17, 2019

Approved by

Yong-Dae Kim
Graduate school of information security

The study was conducted in accordance with Code of Research Ethics¹.

¹ Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS	이상섭. 이오스 자원관리 시스템 취약점 분석. 전산학부 . 2020년.
20173410	27+iv 쪽. 지도교수: 김용대, 손수엘. (영문 논문)
	Sangsup Lee. Who Spent My EOS? On the (In)Security of Resource Management of EOS.IO. School of Computing . 2020. 27+iv pages. Advisor: Yong-Dae Kim, Soo-el Son. (Text in English)

Abstract

EOS is a popular cryptocurrency, whose market cap is over seven billion USD. Its ecosystem operates in the EOS.IO system, which is devised to speed up the slow transaction rate of previous blockchain technologies. Whereas many previous studies have investigated the security issues of Bitcoin and Ethereum, the security of EOS.IO has thus far drawn little attention despite its popularity. Even the studies that have addressed the security of EOS and its underlying blockchain system mostly focused on implementational bugs in the core of the EOS.IO system or in smart contracts, rather than addressing the fundamental problems stemming from the EOS.IO design.

To address this void in the previous literature, we investigate the design architecture of EOS.IO. Based on this investigation, we introduce four attacks whose root causes stem from the unique characteristics of EOS.IO, including intentionally slowing down the block creation time—which can disrupt the essential functions of its blockchain and incapacitate the entire EOS.IO system. In addition, we find that an adversary can partially freeze the execution of a target smart contract or maliciously consume all the resources of a target user with crafted requests. We report all the identified threats to the EOS.IO foundation, one of which is confirmed to be fatal. Finally, we discuss possible mitigations against the proposed attacks.

Keywords EOS, EOS.IO, Resource, Security, Vulnerabilities

초록

이오는 시가총액이 70억 달러가 넘는 인기있는 암호 화폐이다. 이오는 이전 다른 블럭체인의 느린 거래속도를 보완하고자 고안 되었으며 이오스 생태계는 이오스.아이오 시스템에서 작동한다. 하지만 이오스의 많은 인기에도 불과하고 이전의 연구들은 비트코인 및 이더리움의 보안 연구에 초점이 맞추어져 있으며, 협업에서의 이오스 보안의 경우에도 스마트컨트랙트의 구현상의 오류에만 주로 초점이 맞추어져 있다. 이 논문에서는 이전의 이러한 연구의 공백을 해결하기 위해 이오스닷.아이오의 설계적 취약점을 조사하고, 이 조사를 바탕으로 이오스닷 아이오의 고유한 특성을 이용한 4가지 공격을 소개한다. 본 논문에서 소개할 공격은 블럭체인의 필수 기능을 못 사용 하게끔 만들 뿐만 아니라 블럭 생성시간을 의도적으로 느리게 하여 전체 이오스닷아이오 시스템이 작동하지 않을 수 있다. 또한 공격자는 대상 스마트 컨트랙트의 실행을 부분적으로 중지 시키거나 대상 사용자의 모든 리소스를 악의적인 요청으로 모두 소모 시킬수 있다. 확인된 모든 위협은 이오스닷 아이오의 설립자에게 보고되었으며 그중 하나는 치명적인 버그로 명명되어 즉각 패치되었다. 이 논문에서는 위 4가지 공격을 소개하고 공격에 대한 완화 방법에 대해 논의한다.

핵심 날말 이오스, 이오스닷아이오, 리소스, 보안, 취약점

Contents

Contents	i
List of Tables	iii
List of Figures	iv
Chapter 1. Introduction	1
Chapter 2. Background	3
2.1 EOS.IO overview	3
2.2 EOS.IO smart contract	4
2.3 Resources of EOS.IO	4
2.4 EOS resource payment	5
Chapter 3. Attack Model	6
Chapter 4. Attacks	7
4.1 Block delay attack	7
4.2 DoS by draining EOS resources	10
4.2.1 SCP CPU-drain attack	10
4.2.2 SCP RAM-drain attack	11
4.2.3 RAMsomware attack	13
Chapter 5. Mitigation	15
5.1 Preventative measures against attacks	15
5.2 Shifting the payment responsibility	15
5.3 Fine-grained permission control	16
Chapter 6. Scalability evaluation	17
6.1 Crawling EOS Smart Contract	17
6.2 Regulating binary format ABI to JSON format ABI	17
6.3 EOSRAY	18
6.4 Analyzer	18
6.5 CFG recovery	18
6.6 Reachability Checker	19
6.7 Define Sensitive Functions	19
6.8 Limitation and Evaluation	19

Chapter 7. Related work	21
7.1 Security analysis of blockchain systems	21
7.2 Security analysis of EOS.IO	21
Bibliography	24
Acknowledgments in Korean	26
Curriculum Vitae in Korean	27

List of Tables

4.1	Estimated financial loss by the block delay attack	9
4.2	The amount of EOS-CPU and EOS-RAM consumed in the SCP CPU-drain attack	11
6.1	Estimated insecure smart contracts	19

List of Figures

2.1	Architecture of EOS.IO	3
4.1	Overview of the block delay attack.	8
4.2	The amount of EOS-RAM consumed by the SCP RAM-drain attack.	12
4.3	The estimated EOS-RAM loss by the RAMsomware attack.	13
6.1	E-SCV Architecture	17

Chapter 1. Introduction

Recently, *cryptocurrency* has attracted a great deal of attention due to its rapidly growing market cap. Bitcoin, often called the first cryptocurrency, reached a market cap of over 100 billion USD in 2019, and more than 2K cryptocurrencies have emerged worldwide, thus representing a combined value of over 200 billion USD [6]. This surging interest in the cryptocurrency has also attracted the attention of both industry and academia for its underlying *blockchain* system. Consequently, many studies have been conducted to analyze and improve the core technology of each blockchain system [32, 43, 7, 10, 38, 27, 30].

The security of the blockchain is another important issue for which vast research has been conducted. These studies focused on analyzing double spending [19], network-related attacks [14, 2], illicit transactions [26], transaction malleability [9, 1], selfish mining [36, 22, 13, 12, 44], or smart contracts [28, 21, 18, 41, 42, 40]. However, most of these studies have only focused on the two major cryptocurrencies; Bitcoin and Ethereum.

EOS.IO is an emerging blockchain system, whose cryptocurrency, EOS, possesses a market cap of over seven billion USD [6], making it one of the top five cryptocurrencies worldwide.¹ There have been only a few studies on its security [35, 17, 25, 4, 34]. The EOS.IO foundation has been incentivizing researchers through a bug bounty program, but most of the bugs found so far are related to implementation, such as software vulnerabilities that an adversary can exploit to perform arbitrary code execution [35], or logical bugs that cause a smart contract to deviate from the objectives of its developers [17, 25, 4]. However, as the design of EOS.IO is markedly different from that of Bitcoin or Ethereum in managing system resources, there may exist distinctive and unexplored threats that stem from the resource management policies of the EOS.IO system.

To address the lack of research in this area, we chose EOS.IO as the objective of this study and analyzed its security. We first studied the characteristics of the EOS.IO system. Note that some parts of the EOS.IO whitepaper [10] are outdated or ambiguous; hence, we had to manually analyze the source code to figure out its actual implemented semantics.

To speed up the slow transaction rate of previous blockchain systems, EOS.IO adopted a *Delegated Proof of Stake (DPoS)* algorithm [24], which delegates the consensus on blocks and execution of the smart contracts to the representative nodes, called Block Producers (BPs). This design decision of using DPoS entails problems of sharing computational resources among a small number of BPs. To effectively manage resources and prevent resource abuse, EOS.IO asks a participant to obtain a resource capacity, such as computational power, network bandwidth, and storage [10]. Even though this new means of resource management can effectively support the representative nodes in executing smart contracts, it raises new concerns about added assets that can negatively impact the security of the system [16].

Any attack on resource managements that leads to a denial of service (DoS) is a critical security threat. Many emerging blockchain systems strive to achieve high Transactions-Per-Second (TPS) rate, which enables the provision of a short-latency service for their users and service providers. Incessant attacks that successfully undermine the availability or TPS of blockchain services break fundamental requirements that blockchain service users expect. For instance, stock transaction services or payment services at retail stores demand a short latency to process financial transactions. Exacerbating transaction time directly affects the quality of these services, thus contributing to users choosing other blockchain

¹ We checked this at the time of the submission.

services.

Therefore, we investigated resource-related threats that exploit the design architecture unique to EOS.IO. We manifested the capabilities of smart contract providers (SCPs) as well as their service users and defined an EOS.IO adversary who is able to conduct the same level of privileged operations as these participants do. Under this adversary model, we identified four possible attack scenarios: block delay, SCP CPU-drain, SCP RAM-drain, and RAMsomware attacks. In these attacks, an adversary intentionally incapacitates the entire EOS.IO system with a crafted smart contract, an adversary depletes or drains the resources of a smart contract provider so that this victim no longer provides the service, and a malicious smart contract provider surreptitiously replaces the original smart contract with another one to steal user assets.

In order to address these attacks, we analyze their causes and propose several mitigations. These mitigations include simple preventative measures along with suggestions for systematic changes of EOS.IO. We believe that the proposed mitigations can prevent the attacks described in this paper as well as potential future threats.

In summary, our contributions are as follows:

- We present the *block delay* attack, a novel attack that exploits a transaction scheduling policy of the EOS.IO system. This attack is able to delay all transactions in the system, thus resulting in the DoS.
- We introduce two new attack methods: *SCP CPU-drain* and *SCP RAM-drain* attacks. These attacks exploit resource management policies unique to the system, thus disrupting services from a target smart contract provider.
- We present the *RAMsomware* attack scenario that abuses controversial design decisions of the system. The attack allows locking EOS-RAM resources of a target user, which enables an attacker to ask a ransom in exchange for releasing the locked EOS-RAM resources.
- We demonstrate the feasibility of each attack and estimate the impact of each proposed attack scenario based on experiments conducted in a local EOS.IO system.
- We propose preventative measures as well as a redesign of the EOS.IO system to mitigate the proposed and potential threats.

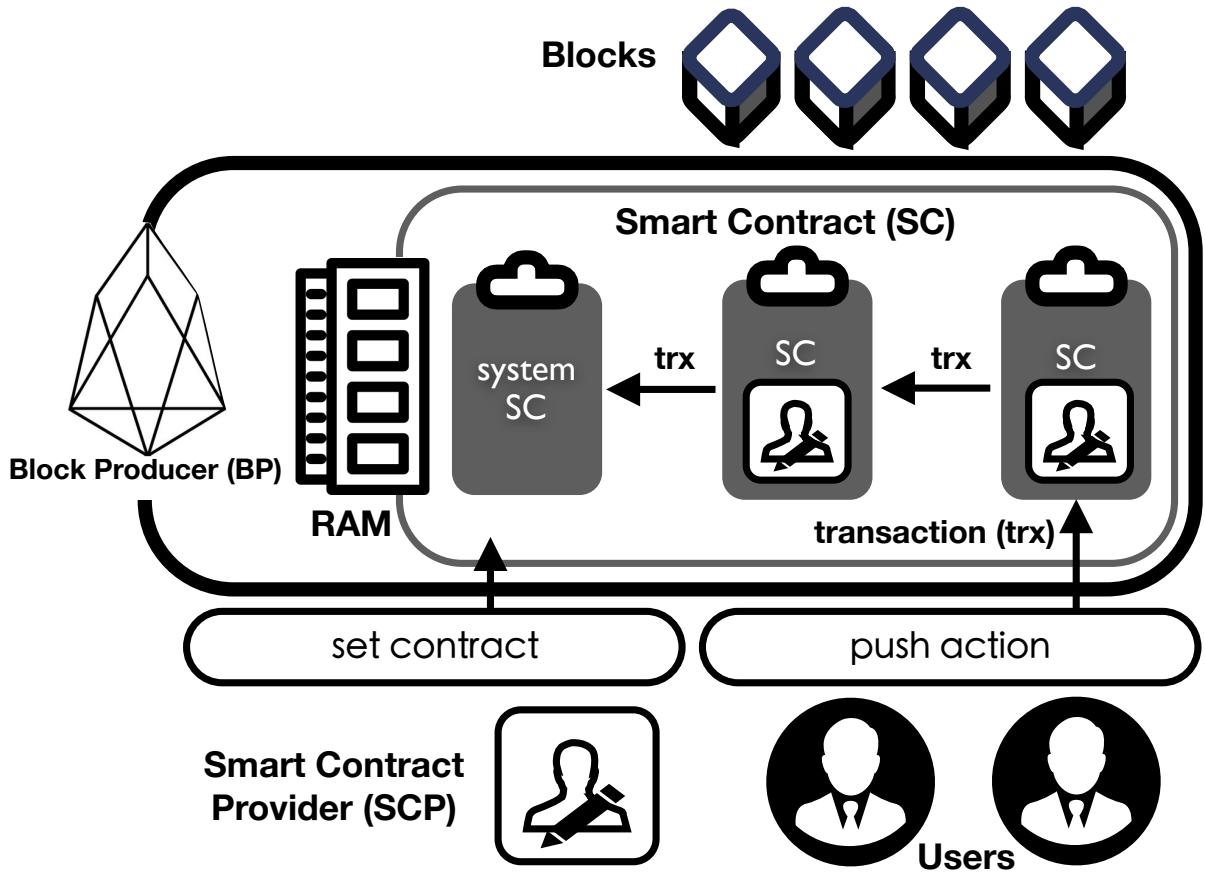


Figure 2.1: Architecture of EOS.IO

Chapter 2. Background

2.1 EOS.IO overview

EOS.IO is a blockchain system for a cryptocurrency called EOS. In this section, we describe the main characteristics of EOS.IO. As some parts of its whitepaper [10] are outdated and ambiguous, we interpret them based on our judgment by carefully analyzing the EOS.IO source code. Figure 2.1 illustrates a simplified overview of the EOS.IO ecosystem. The main difference between EOS.IO and the other blockchain systems lies in its consensus algorithm, which is the core of the blockchain system. Unlike the other blockchain systems of the top cryptocurrencies in the market [6], EOS.IO adopted *Delegated Proof of Stake (DPoS)* [24] as its consensus algorithm. DPoS delegates the role of a node in a blockchain system to a small number of representative nodes, known as *Block Producers (BPs)*. Among the nodes in the EOS.IO network, 21 BPs are selected through a voting procedure; these BPs are responsible for deciding whether newly created blocks can be attached to the current chain. With DPoS, EOS.IO is able to significantly speed up its transaction rate.

Another feature of EOS.IO is its ability to execute a *smart contract (SC)*, which is essentially a program developed by a user. It is similar to those supported by the Ethereum platform [43]. In this

paper, we call the developers of SCs as *Smart Contract Providers (SCPs)*. In the EOS.IO system, if a user wants to execute an SC, s/he can send a *transaction* to one of the BPs in the EOS.IO network. This transaction consists of several *actions* that specify a target SC and the parameters for the execution of the SC. When the BP receives the transaction, it executes the requested SC after fetching the SC from the EOS.IO chain and creates a new EOS.IO block that stores the execution results. Then, the new block is propagated to the other BPs through a broadcasting procedure.

2.2 EOS.IO smart contract

An SC often refers to a computer program that executes on a blockchain system or a distributed ledger. Many blockchain ecosystems, including Ethereum and EOS.IO, support the execution of a given SC for various purposes, including banking, gambling, initial coin offerings (ICOs), and trading in the marketplace [8].

The design of Ethereum, the most popular blockchain system that supports SCs, demands that every SC in their system should be unchangeable; thus, it allows no revision not even for updating the source code or fixing vulnerabilities in the SCs. Therefore, SCPs, in practice, dispose of their SCs and create new ones in order to patch inherent bugs.

On the other hand, EOS.IO allows SCPs to modify their SCs. However, this design decision entails another problem that SCPs are able to change the semantics of their SCs, which are then executed by users without any active notification of the change. Therefore, users have few options but to completely trust SCPs to execute their SCs while understanding that these SCs may be changed at any time.

The option still remains for users to analyze an SC to understand its semantics before executing the SC; however, SCs in the EOS.IO system are binary programs compiled in the form of *Web Assembly* [5], which makes the understanding of their semantics difficult. Furthermore, SCPs are under no responsibility to open the source code of their SCs. According to one of the EOS.IO explorers called EOS Park, only 10% of SCPs release their source code publicly [11]. These characteristics of EOS.IO make it difficult for users to analyze each SC, further contributing to users having no choice, but to trust the goodwill of SCPs.

2.3 Resources of EOS.IO

Every transaction and execution involving SCs in the EOS.IO system consumes resources. By design, the system abstracts its resources into three items: computational power, network bandwidth, and storage. For convenience, in this paper, we call these as EOS-CPU, EOS-NET, and EOS-RAM, respectively. When a user directs a transaction to run an SC, the user should already have enough resources for the consumption by the SC, called *transaction costs*. Therefore, the platform asks participating SCPs and users to purchase EOS-RAM or to stake EOS-CPU and EOS-NET. Here, *staking* refers to an action of allocating a certain number of EOS tokens (i.e., EOS cryptocurrency) to reserve BP resources.

EOS.IO adopted this mechanism to effectively manage the constrained resources of the BPs and prevent resource abuse. More specifically, when a user stakes EOS tokens, EOS.IO distributes the capacity of the resources to a user proportional to the tokens staked by this user. For example, if a user staked 1% of the total staked EOS tokens, then the user is allowed to use 1% of the total resource capacity as well. Staked tokens do not disappear but are marked as staked in the system. That is, a user can unstake his/her tokens anytime, and the tokens become available after three days. At the time

of unstaking, the staked EOS-CPU and EOS-NET are also released. On the other hand, the tokens of the purchased EOS-RAM are not returned. EOS-RAM is storage for the data used within an SC, and the user can refund EOS-RAM only if the data used in the target SC is removed. Therefore, a user should consider using his/her resources wisely.

2.4 EOS resource payment

The underlying design philosophy for EOS.IO is *Receiver Pays*, denoting that SCPs should pay for the usage of their SCs by users. More specifically, a user only pays for the initial transaction cost, while the SCP pays for the other costs of the subordinate transactions which come within the first SC. Consequently, the costs of running SCs become the business costs for which SCPs should be responsible. SCPs stake their EOS-CPU and EOS-NET for the users who execute their SCs, and the amount of staked resources strongly affects the number of processed transactions from users.

However, it is burdensome for those SCPs who require a large amount of EOS-CPU and EOS-NET to execute their SCs. In such cases, SCPs can delegate the resource usage to users. Even though EOS.IO enables the SCPs to delegate the transaction costs to the users, it requires an additional permission, called the `eosio.code`, from the users. If this permission is given, an SCP can control all the resources of a user, including EOS tokens. As a result, since most users would not want to give their permission to an SCP, many SCPs follow the Receiver Pays model by paying for the business costs during the execution of their SCs.

Chapter 3. Attack Model

We assume an EOS.IO adversary who utilizes the same functionalities of the EOS.IO system as SCPs and their service users do. That is, the adversary is able to implement and manage her own SCs, or to execute other SCs. The adversary can also entice victims to send a transaction that triggers the execution of a crafted SC. Furthermore, the adversary is capable of updating the crafted SC anytime to leverage the misplaced trust of the victims who only examined the previous SC before the update. One goal of the adversary is to deplete available resources of SCPs, such as EOS-CPU, EOS-NET, or EOS-RAM, thus causing the denial of service (DoS) for these SCPs. The adversary also aims to cause delays in block generation in BPs, because such delays significantly undermine the service availability of the EOS.IO system. Consider SCs for stock trading or banking services, which demand a short latency for task completion. In such services, any system delay can potentially lead to a significant financial loss as transactions that should be processed within a short time cannot be processed. Furthermore, the adversary could also plunder the resources of the victims and demands a ransom.

Note that our adversary model is similar to those used in the previous studies that exploit SC bugs [28, 21, 18] in Ethereum or cause double spending in Bitcoin with spam transactions [37], except that the adversary is able to update the crafted EOS.IO SC.

Chapter 4. Attacks

We present four novel attacks against the EOS.IO system. The block delay (§4.1) attack undermines service availability of BPs, thus contributing to block generation delays in the EOS.IO system. The SCP CPU-drain (§4.2.1) and SCP RAM-drain (§4.2.2) attacks deplete the resources staked for the execution of SCs, including EOS-CPU and EOS-RAM, thus resulting in the denial of SC services. The RAMsomware (§4.2.3) attack exploits the controversial EOS design decision that allows SC updates. The adversary starts by obtaining the sensitive permission with her benign SC code and then changes it later to deplete the EOS-RAM of the victims. The adversary may then ask a ransom for releasing victims' EOS-RAM. We emphasize that all the presented attacks harness attack vectors unique to EOS.IO.

In this section, we describe each attack and demonstrate its feasibility with an attack experiment that measures its potential impact on EOS.IO.

Experimental setup. We conducted experiments to measure how much loss each attack can cause in EOS.IO. We used a machine running 64-bit Ubuntu 16.04.3 LTS with Intel Core i7-8700 CPU at 3.20 GHz (12 cores), 32 GB of RAM, and 200 GB SSD. For the testing environment, we prepared two different versions of EOS.IO on the machine: version 1.1.1, where we discovered the block delay attack, and version 1.7.3, which is the latest stable version to confirm the patch for the block delay attack and to test the feasibility of other attacks. Each version of EOS.IO is initialized with default system SCs that manage the resources of EOS.IO and the execution of other SCP-provided SCs. Note that we have never conducted the presented attacks against the actual EOS.IO system in production for ethical and respectable research.

Collected SCs. We collected 3,212 SCs from 3,600 accounts in the EOS.IO main network and installed them in our testing environment.

4.1 Block delay attack

The block delay attack causes a delay in the production of blocks in a BP by generating an unacceptable number of spurious transactions. These spurious transactions elicit misbehaviors in the BP when scheduling them, thus usurping opportunities that should have been used for processing valid transactions. This entails huge financial losses for participants in EOS.IO by delaying the transactions that require immediate completion. We note that this attack targets all SCs running in BPs, not a particular SC, which makes the attack even more critical.

A DoS on any blockchain system is indeed a critical security threat, as many emerging blockchain systems strive to provide a high-availability service with a high TPS. Because the EOS.IO system only depends 21 representative BPs to process worldwide transactions, the DoS on a single BP may cause a significant TPS degradation. We first describe the EOS.IO block generation procedures and then explain how the block delay attack exploits the procedures.

Block generation procedures. The EOS.IO system has two internal policies; 1) a BP should generate a block within every 0.5 seconds; 2) a BP limits its resource usage for processing transactions within a block. Specifically, EOS.IO limits the execution time as well as the amount of EOS-CPU and EOS-NET for processing transactions within a single block.

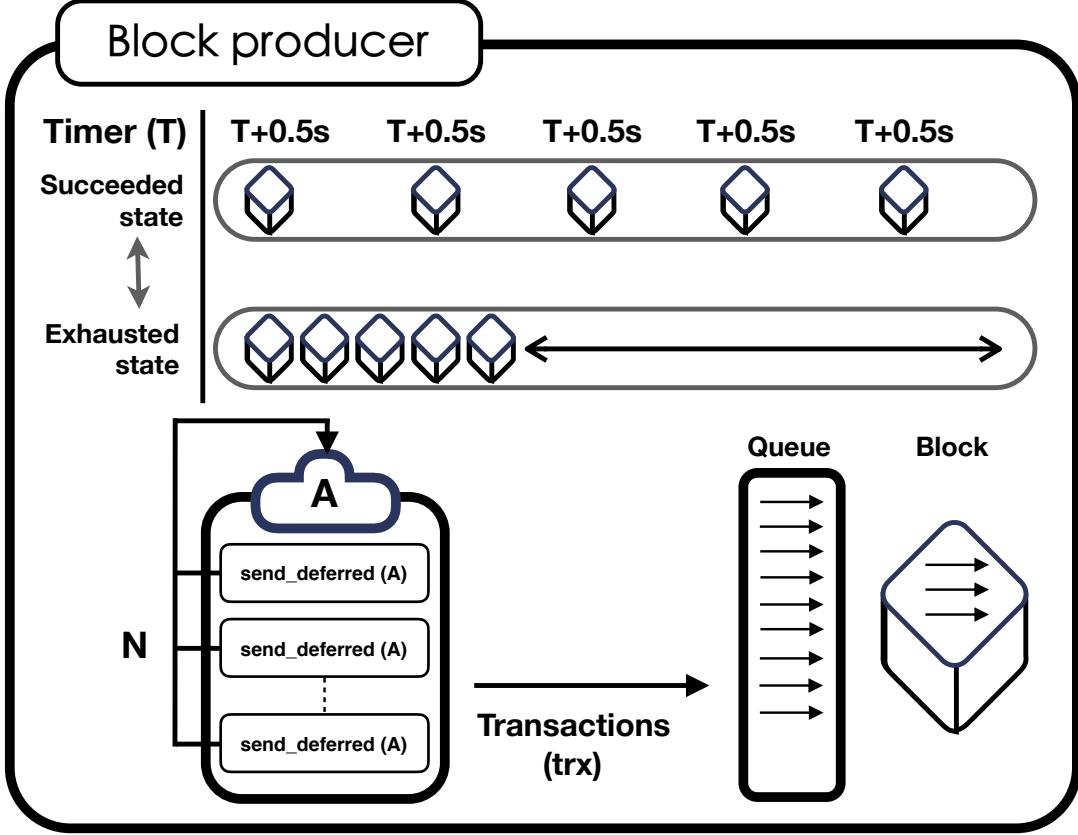


Figure 4.1: Overview of the block delay attack.

To enforce these block generation policies, EOS.IO has four internal states, including *succeeded* and *exhausted* states. We only explain these two states relevant to initiate the block delay attack. In the succeeded state, a BP generates a block for every 0.5 seconds. When a BP completes given transactions earlier than 0.5 seconds, it waits further and then propagates the resulting block to other BPs. If the resource usage for processing transactions within a block is over a specified threshold, it switches its state to the exhausted state and generates as many blocks as possible without any waiting time to maximize its computation resources.

A block generation delay arises when the exhausted state is changed back to the succeeded state after a large number of blocks are generated in the exhausted state. This delay stems from the time gap between the timestamp specified in a block and the actual time when the block is generated. For each block, EOS.IO specifies a timestamp with the multiplication of the fixed time interval (0.5 seconds) with the number of all generated blocks in the entire EOS.IO system that precede the current block. That is, a BP in the exhausted state could create a block of which timestamp indicates a future time, when the BP creates more than one block within 0.5 seconds. When the state is changed back to the succeeded state, the BP attempts to generate a new block with a future timestamp. It then pauses its block generation until the real time matches the timestamp of this new block.

Attack conditions. In order to intentionally invoke such a block generation delay, the attack should 1) change the succeeded state to the exhausted state, 2) make the BP to generate a large amount of blocks in the exhausted state, and then 3) change the BP back to the succeeded state.

By carefully analyzing the EOS.IO system, we found that an adversary could easily conduct the

Table 4.1: Estimated financial loss by the block delay attack

Block Count	Attacker			Victim		
	Time [†] (min)	EOS-CPU (min)	EOS-NET (MiB)	Cost [‡] (EOS)	Delay (min)	Loss [*] (EOS)
376	0.92	1.23	16.13	480	2.05	40,802
704	2.06	2.32	34.72	910	3.56	70,856
1,106	3.02	3.65	50.82	1,426	5.67	112,851
1,471	4.00	4.85	65.53	1,894	7.46	148,478
1,840	5.04	6.07	79.69	2,368	9.12	181,518

* We estimated the loss of EOS.IO from the total volume of traded EOS tokens in April 2019.

† As this attack creates a large number of transactions, it was difficult to precisely control the attack duration.

‡ The attack cost is estimated by multiplying the total staked EOS tokens by the ratio of the used EOS-CPU and EOS-NET to their total capacity.

block delay attack by abusing two EOS.IO features: deferred transactions, and smart contract updates.

In general, a BP executes a requested transaction right away, however, a deferred transaction is scheduled later for its execution [10]. The original purpose is to move computations into different shards or to create a long-running process for continuance transactions. To initiate a deferred transaction, an SC invokes `send_deferred()`, which makes a new asynchronous transaction on a BP. Suppose an adversary implements a malicious SC that recursively invokes multiple deferred transactions of itself. The execution of this SC causes an exponentially increasing number of crafted transactions, as Figure 4.1 illustrates.

When a BP receives such a large number of transactions, its resources eventually runs out and the BP changes its state to the exhausted state.

After producing numerous transactions, an adversary changes the crafted SC to a totally different one. This change renders all the queued transactions from the original SC invalid because those transactions still attempt to call methods in the original SC which no longer exists. Therefore, the BP in the exhausted state marks the spurious transactions as invalid and immediately process them, thereby generating a large amount of blocks. Consequently, these spurious blocks generates a huge gap between the timestamp of the last block and the real time. When the BP comes back to the succeeded state, the BP pauses its further block generation. Consider that a BP generates each one of 100 blocks for 0.2 seconds in the exhausted state, thus taking 20 seconds ($0.2 * 100$). The timestamp of the last block must be set to 50 seconds ($0.5 * 100$), thereby producing the time gap of 30 seconds. As a result, the BP should delay its next block generation for 30 seconds when the state changes back to the succeeded one. As the adversary is able to manage the number of blocks with the deferred transaction, she can impose an arbitrary delay time, resulting in the DoS of the BP under the attack.

We conducted experiments to measure the extent of the losses that this attack can cause to the EOS.IO system. We created a malicious SC that internally makes six deferred transactions that invoke the SC itself, thus causing every single transaction to invoke the execution of the same SC six times.

Table 4.1 summarizes the experimental results. The block count describes the total number of blocks created during the attack. The *Attacker* columns describe the time and EOS.IO resources consumed

during the attack. Note that the exact number of required EOS tokens can vary depending on the total number of staked EOS tokens (see §2.3). Here, we calculated them using the required values of EOS-CPU and EOS-RAM in the real EOS.IO system at the time of the submission. However, since the EOS tokens are returned after unstaking EOS-CPU and EOS-NET as described in §2.3, the cost of the attack can be considered as zero. The *Victim BP* columns show the time delay of the BPs and the expected number of EOS tokens that are not traded due to the corresponding delay, respectively. We estimated the loss based on the number of actual EOS tokens conducted in the main EOS.IO network. The equations for calculating the delay time (T_{delay}) and the estimated loss (E_{loss}) are shown as below:

$$E_{min} = E_{April}/((60 \text{ mins}) * (24 \text{ hrs}) * (30 \text{ days}))$$

$$T_{delay} = (T_{limit} * N_{blocks}) - \sum_{i=1}^{N_{blocks}} T_{block_i}$$

$$E_{loss} = E_{min} * T_{delay}$$

E_{April} represents the number of EOS tokens actually signed in April 2019, which was 859,821,765, and E_{min} represents its one-minute average; T_{limit} is the time limit for executing each transaction in EOS.IO, which is 0.5 seconds; N_{blocks} is the number of blocks created, and T_{block_i} is the execution time of i-th block during the attack.

The consequences of the attack are severe since a single adversary running the attack for 5 minutes with 2,368 EOS tokens causes a financial loss of 181,518 EOS as well as undermines the service availability for nine minutes. Note that the cost of the attack is zero since the resource of the adversary spent for the attack is returned in the very next day. In addition, coordinated attacks targeting many SCs can certainly cause the DoS at all BPs. We reported this bug to the EOS.IO foundation, and they marked it as one of the most critical bugs.

The EOS.IO developers patched this bug by adding a timestamp check for each EOS.IO block generation. This patch makes a BP to compute the expected time for a newly generated block and checks whether the difference between this expected time and the real time is within 0.5 seconds. If so, the BP generates the next block without any delays. Otherwise, it waits until the real time reaches the expected time. This check prevents a block generated in the exhausted state from having a future timestamp.

4.2 DoS by draining EOS resources

This section presents two DoS attacks: SCP CPU-drain and SCP RAM-drain attacks. Both attacks aim to undermine the service availability of a target SCP by draining staked EOS-CPU and EOS-RAM, respectively.

4.2.1 SCP CPU-drain attack

We present a novel attack that depletes all the EOS-CPU staked for the execution of a target SC, thus making this SC unavailable for further execution. The goal of the adversary is to render the target SC unavailable for any BP by depleting all the EOS-CPU staked by the SC owner in exchange for the EOS-CPU staked by the adversary. We refer to this attack as an *SCP CPU-drain* attack.

An SC often invokes `send_deferred()` that generates a deferred transaction for the execution of another SC. By design, this execution of an SC requires EOS-CPU from 1) the user who initiated the

Table 4.2: The amount of EOS-CPU and EOS-RAM consumed in the SCP CPU-drain attack

Attack Count	Attacker		Victim SCP	
	EOS-NET (KiB)	EOS-CPU (ms)	EOS-NET (KiB)	EOS-CPU (ms)
	1	0.137	0.146	3.562
10	1.329	1.485	3.555	4.336
20	2.655	2.938	3.549	8.352
30	3.980	4.474	3.544	12.53
50	6.626	7.422	3.534	20.74
100	13.21	15.23	3.509	41.19

transaction or 2) the SCP of the original SC. However, spending EOS-CPU belonging to a user requires the consent of this user that grants the `eosio.code` permission [10] to the SC. Because this permission is so sensitive that it allows the SC to manage the EOS tokens of the user, users are naturally hesitant to grant this permission. Therefore, SCPs often use their staked EOS-CPU for deferred transactions to facilitate the wide adoption of their SCs.

When the staked EOS-CPU of an SCP is exhausted, the corresponding SC becomes unavailable for 24 hours.

The SCP CPU-drain attack exploits this feature, by continuously consuming the EOS-CPU of the victim SCP. As a result, the attack is cost-effective if the total use of the staked EOS-CPU of the victim SCP is greater than that of the adversary. The adversary could also speed up the attack by creating an SC that recursively calls itself as well as the target SC similar to the block delay attack.

To demonstrate the tangible threat of the SCP CPU-drain attack, we sampled one of the vulnerable SCs¹ among the SCs that we collected from the main EOS.IO network. In order to do this, we manually analyzed a few SCs to check if a `send_deferred()` function call exists at the beginning of the SC without a proper user validation.

Table 4.2 shows the experimental results. During the experiment, we varied the number of calling the target SC from 1 to 100.

When we conducted the attack once, the consumed EOS-CPU of the victim SCP (11.50 ms) reached almost three times as much as that of the adversary (4.136 ms). The gap in EOS-CPU between the attacker and the victim SCP increases almost at an almost constant rate. This result demonstrates the feasibility of the SCP CPU-drain attack that an adversary can intentionally dry off the victim using only a small EOS-CPU amount.

4.2.2 SCP RAM-drain attack

We present another threat that allows an adversary to deplete the EOS-RAM of a victim SCP by inserting spurious data, which results in the DoS for the SCP. We refer to this attack as an *SCP RAM-drain* attack.

Compared to other blockchain systems, EOS-RAM is another unique component of EOS.IO that abstracts the available data storage for executing an SC. Unlike EOS-CPU, SCPs are able to choose either

¹We intentionally omitted the SC name in the SCP CPU-drain attack.

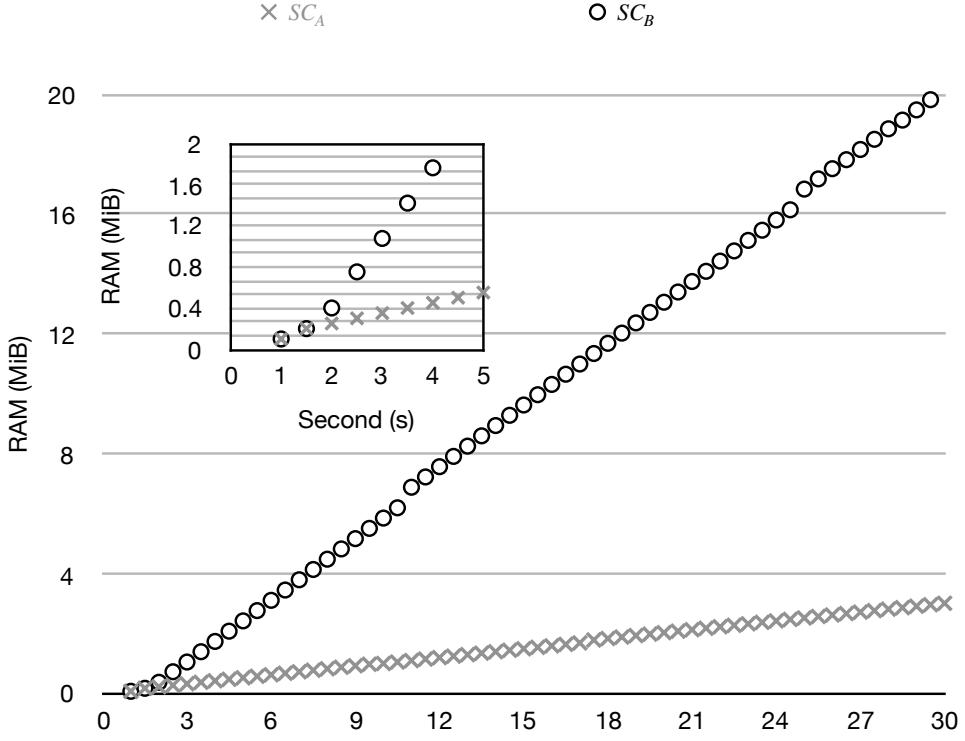


Figure 4.2: The amount of EOS-RAM consumed by the SCP RAM-drain attack.

themselves or other users who execute their SCs to pay for the cost of storing user data without any permission.

In addition, EOS.IO stores data in the form of a key-value pair, as in general database systems. Thus, it is recommended that one should properly create a unique key for each data insertion to prevent storing duplicated data. When an SCP chooses to pay for his/her SC instead of the user without such consideration, his/her SC can become vulnerable.

That is, an adversary can keep storing spurious data to EOS-RAM by exploiting this vulnerable SC until there is no remaining EOS-RAM purchased by the SCP, which eventually causes the DoS of the vulnerable SC.

Among the collected SCs, we manually searched ones that use the EOS-RAM of SCPs and implement no measure to prevent the same user from storing unlimited data. We experimented one vulnerable SC ² as a case study to show the feasibility of the SCP RAM-drain attack.

To conduct the attack, we implemented two additional SCs and compared their efficacy. SC_A sends the vulnerable SC a transaction that stores spurious data at the EOS-RAM staked by the SCP of the SC. It then invokes itself via one `send_deferred()` call. SC_B does the same as SC_A , however, it invokes `send_deferred()` twice, thus executing two instances of the SC at once.

Figure 4.2 shows the experimental results. As the graph shows, the slope of SC_B is much steeper than that of SC_A . It only took 30 seconds to deplete 20 MB for SC_B . The actual efficacy of the SCP RAM-drain attack can differ depending on how much of EOS-RAM is consumed and how the execution logic is implemented; however, this is only a matter of time.

Once the EOS-RAM of the attacked SCP is exhausted, the function for the SC to use EOS-RAM becomes no longer available. Here, the SCP can operate the SC again by purchasing additional EOS-RAM; however,

²We intentionally omitted the SC name in the SCP RAM-drain attack.

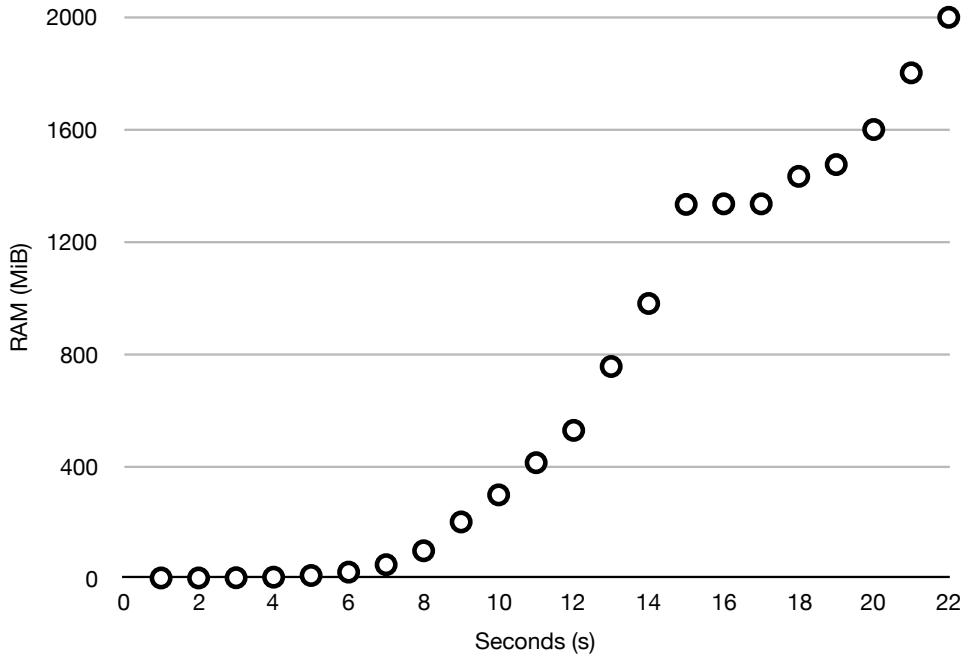


Figure 4.3: The estimated EOS-RAM loss by the RAMsomware attack.

this is only a temporary measure. To sort out this problem, SCPs should add a proper defense mechanism by modifying the source code of the SC and have to delete all the EOS-RAM tables which may contain valuable user data.

4.2.3 RAMsomware attack

We present the *RAMsomware* attack, a new threat that enables the adversary to take possession of the resources of victims including EOS tokens, EOS-CPU, and EOS-RAM, leveraging misplaced trust in SCs. This attack takes advantage of the fact that the SC of the adversary for executions can be different from the SC for which a victim granted the `eosio.code` permission for privileged operations. This is a classic time-of-check to time-of-use (TOCTOU) attack.

The adversary begins the attack by uploading a benign SC. This benign SC requests the `eosio.code` permission from users for further execution. As described in §2.4, users do not want to grant their permission to SCPs in general. In order to get the `eosio.code` permission, the adversary can publish the source code of the benign SC and obtain a safety inspection; there exist third-party services that test the security of the submitted SCs [11]. The adversary then promotes this benign SC on his/her own website along with the safety inspection results. A victim checks the published code as well as the inspection results and decides to grant the permission to this SC. Later, the adversary switches the SC to a malicious one, but the victim does not recognize this change. Thus, the victim is highly likely to run this changed SC. However, the EOS.IO system does not warn or notify the victim of any code changes made by the adversary. This attack stems from the design decision of EOS.IO that allows an SCP to update and revise their SC without notifying users of such SC changes.

One attack scenario is to lock the EOS-RAM of victims for ransom. Assuming that the adversary is capable of obtaining the `eosio.code` permission from victims, the attack SC can call itself repeatedly via `send_deferred()` while storing spurious data using the EOS-RAM of the victim until they are completely

exhausted. Unlike EOS-CPU or EOS-NET which are restored every 24 hours, the EOS-RAM will be locked until the adversary releases their usage. Therefore, the adversary can ask a ransom in exchange for returning the locked EOS-RAM.

We conducted an experiment to check the feasibility of this attack in our testing environment. We prepared a ransomware SC and a victim account that grants the `eosio.code` permission to this SC. To fasten the completion of EOS-RAM exhaustion, we used the same methodology described in §4.1, whereby the attack SC recursively invokes `send_deferred()` multiple times. As Figure 4.3 shows, it took only around 22 seconds to deplete 2 GB of EOS-RAM, which is the EOS-RAM capacity of the user with the most EOS-RAM when excluding the EOS.IO foundation [11].

Chapter 5. Mitigation

In this section, we propose mitigation methods for the presented attacks. We first enumerate precautionary suggestions that require no changes to the current EOS.IO system. We then discuss several design changes to the EOS.IO system to remove the root causes of the attacks presented herein.

5.1 Preventative measures against attacks

The SCP CPU-drain , SCP RAM-drain , and RAMsomware attacks can be easily prevented if SCPs and users carefully inspect their SC source code as well as the SCPs . For the SCP CPU-drain attack, an SCP should verify if the code exists in their SC that checks the SC’s callers in a manner that they cannot abuse. For example, one can add a permission check at the beginning of the SC , so that only authorized users can use the SC . On the other hand, to prevent the SCP RAM-drain attack, SCPs should carefully audit their SCs to determine whether there exists code that mistakenly uses their EOS-RAM . In the case of the RAMsomware attack, a user should give his/her permission code only to the fully trusted SCPs .

These simple preventions seem to be textbook examples, and none of them requires any change to the underlying EOS.IO architecture. To render the system more secure, however, we do believe that the system must do its best effort to address threats from every entity in the system. For instance, in case of the SCP CPU-drain attack, the EOS.IO system can improve its security by adding extra security components to it. This approach is not uncommon as one of the largest systems, a Linux kernel, adopted a similar concept to alleviate the threats derived from mistakenly omitted permission checks [29]. In the following subsections, we discuss potential changes to the EOS.IO system to fundamentally eliminate the root causes of the attacks presented above.

5.2 Shifting the payment responsibility

As described in §4.2.1, in the SCP CPU-drain attack, a malicious user intentionally triggers a target SC to deplete the EOS-CPU of its SCP. Of course, an SCP can prevent this by carefully inspecting the code of their SC. However, this attack stems from the current design that encourages SCPs to pay for the execution of their SCs originating from user transactions.

In the current EOS.IO system, to impose the transaction cost on a specific user, an SCP has to specify the user as authorized in the SC , and this process requires the `eosio.code` permission from the user. In general, users are not normally willing to grant this permission code since anyone with the code is able to control the user account. To avoid this, an SCP is supposed to create an SC with his/her own permission code that imposes the transaction cost of executing the SC onto the SCP himself/herself so that any user can be pleasant for the execution of the SC without providing the permission or paying the cost. Therefore, our design change suggestion removes the constraints of SCPs, thus enabling them to avoid using their own resources, which also eliminates the target of the attacker.

One way to address the SCP CPU-drain attack could be to redesign the EOS.IO system so that a user who initiates a transaction should pay the transaction cost without having to provide the `eosio.code` permission. It is quite intuitive to impose a transaction cost on the user who actually

wants to run the SC. This fundamentally eliminates the liability of checking the permission of the SC, as well as the benefit of the attack—thus, the adversary would no longer be motivated to commit it.

On the other hand, this mitigation increases the possibility of opposite cases in which a malicious SC could deplete the resources of a user because it does not need the permission code of the user. Even when the user chooses to run a malicious SC, the system should be responsible for alleviating the severity of the threat. However, as the current EOS.IO system supports limiting the maximum resource use for running an SC, this can be mitigated as well [39].

5.3 Fine-grained permission control

The root cause of the RAMsomware attack is a user giving his/her permission code (`eosio.code`) to an SCP in the belief that the SCP will not abuse this permission. This, in fact, can be considered the fault of the user as s/he gives permission with full awareness of the significance of their action. Nevertheless, the EOS.IO design can be changed to protect users from these kinds of threats.

Currently, the EOS.IO permission system pursues an all-or-nothing approach. That is, a user can make only one kind of action regarding his/her permission—either give or revoke it. This fundamentally facilitates threats on assets in EOS.IO, including the RAMsomware attack. The EOS.IO developers are also aware of this issue and have proposed pinning the permission into a specific version of the target SC. However, this still remains an unresolved issue [3]. Nevertheless, we have proven the feasibility of this threat, as shown in §4.2.3, and here we suggest possible mitigation that can eradicate such threats.

We propose a fine-grained permission control, including the permission pinning described above. It enables a user to set a specific period or a specific version that the permission depends on. As a result, in case of the RAMsomware attack, even though a malicious SCP replaces a benign SC with a fraudulent one, the given permission would automatically lose its validity. Moreover, the expiration date of the permission can reduce the feasibility of the threats occurring because a user might forget to revoke permission. Therefore, the fine-grained permission control can help to mitigate several threats to the EOS.IO assets.

Chapter 6. Scalability evaluation

The Resource drain attacks are supposed to have patterns of vulnerabilities. Thus, In this section, we introduce our scalability evaluation tool that we called E-SCV(Evaluation Smart Contract vulnerability) for evaluating how many smart contracts have exposed to our attacks.

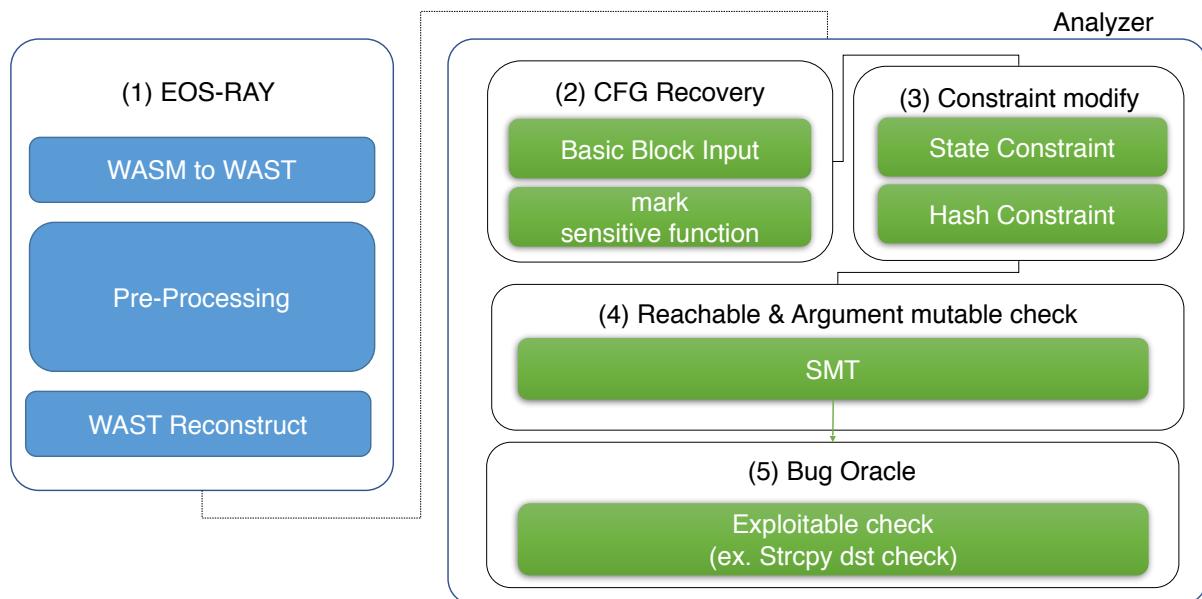


Figure 6.1: E-SCV Architecture

6.1 Crawling EOS Smart Contract

We have collected webassembly smart contracts for use as seed files. We also do analysis BP Node's packets to collect smart Contracts from producers. The easy way is that we used "get_raw_code_and.abi" URL which returns smart contracts and Abi files from producer ID. As a result, we can get not only 2960 smart contracts but also Abi files that are existed as binary types, not JSON formats. there is various information for using smart contracts in an ABI file as type of dictionary-style. the Action type includes external function lists and structs have types of arguments which are consist of Names, Fields. Name field means literally a name of method. and Fields are abstraction for a type of arguments. the Action type includes external function lists and structs that have types of arguments which are consist of Names, Fields. The name field means literally a name of the method. and Fields is an abstraction to a type of argument.

6.2 Regulating binary format ABI to JSON format ABI

An ABI file which is coming from BP Node is not readable format. We extracted data which is needed to call smart contract such as method Name, argument types from binary format ABI as reverse

engineering. and We transform this data into JSON file format. We analysis the binary format and the pattern is below.

An ABI binary file has a specific file format. you can analysis using hex-editor. The first byte which is in the ABI file has the length of the version string. We can get version string from ABI file reading length byte's size. This method which read specific size for the string is shown repeatedly. and let's called this pattern *Carving[Target]*. Right after version information string, there are "structs"’s a number value. We can get all the "struct" names using the *Carving [StructsName]* method by Repeating the number of times previously obtained. Each "struct" Name value has its own Field. Because of this, After *Carving[StructsName]* 1 byte has values of numbers which mean Field count. we can get all "field" names using the *Carving[StructsField]* method by repeating the number of times previously obtained. After "struct" name comes "action name". An "action" name is similar to a "struct". we can get an "action" name with *Carving[ActionName]*, but there is a difference that has a 16-byte header. so after 16-bytes, there is the "Action" name string. the next format is "table". A "table" part also has a 16-bytes header, and we can get the "Table" name string using *Carving[TableName]*. Each table has a number of key names and key types. to express this. tableName format next byte is count value for types, names. With same method before, we can get all "type" value and "name" value using the *Carving[KeyName]* and *Carving[KeyType]* method by repeating the number of times previously obtain number. Next byte is name for "key type" it also could get using *Carving[KeyTypeValue]*.

6.3 EOSRAY

We need to change WASM-file to WAST-file which is a more convenient type for analysis. Fortunately, The EOS foundation gave the tools called "wasm-dis" to developers for transforming from a wasm file to a wast file. thus, We also used the "wasm-dis" binary file for converting wasm to wast. There are 8-types in web assembly; export, import, type, elem, data, global and function. Especially, elem type defines external callable function aka action method. We redefined all webassembly instructions, which are 172 instructions, to C-level pseudo-codes so that we can track a control flow from a function of elem to end of that function. The C-level pseudo-codes include various expressions. The various expressions are used to the symbolic execution’s input value, and we talk about this Analyzer sections.

6.4 Analyzer

The main standard point that decides a bug is that data reachability. This means that if the sensitive function exists in the function without input data, it is not a bug. thus, E-SCV makes a control flow graph (CFG) base on basic block units to the tree data structure shape. Then, E-SCV marks the function’s arguments and tracks the arguments every single step. If the argument reach branch, E-SCV judge reachability using taint analysis. This process is terminated when arguments reach the leaf node or sensitive function.

6.5 CFG recovery

In computer science, a control-flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control-flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets;

Table 6.1: Estimated insecure smart contracts

	send_dereffered[]	db_store_i64[3-5]	db_update_i64[1][2]	memcpy[1]	strcpy[1]
Total API calls	790	8,059	11,620	205,033	63
Reachable APIs	117	11	7	393	1

Total: 2960 smart-contracts

”[num]” means that argv[num] reachability

”[â]” means that All argument except argv[â]

jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. The CFG is essential to many compiler optimizations and static analysis tools. CFG helps to find the whole code coverages. teTher [21], which is ethereum audit tool, also uses this method. We adopt the CFG recovery methodology to web assembly smart contracts, and it is base on the basic block tree. E-SCV builds the basic block tree with expressions that are used each basic block.

6.6 Reachability Checker

After reconstructing, E-SCV has to mark the function’s arguments and track the arguments every single step and make sure reachability to sensitive functions and argument manipulatable via users. Therefore, E-SCV calculates every instruction and data along to constraint-path as Depth First Search (DFS) method. If the code flow can be reached to sensitive functions, E-SCV could solve conditional equations using Satisfiability Modulo Theories (SMT) Solver, while back tracing every basic block with saved variables and expressions but this part is not implemented yet. For now, the Only thing have is input reachability to sensitive functions using taint analysis.

6.7 Define Sensitive Functions

The send_dereffer() API use EOS-CPU which has to staking EOS coin. So, To reduce unnecessary waste of resources, If the developer wants to use send_dereffer() APIs, s/he is supposed to set authorization for using its. db_update_i64() and db_store_i64() are also have to set strict ram-payer for preventing overcharge EOS-RAM which has to purchase by not only attacker but also benign users. This is just the developers mistake. so we evaluated how many smart contracts that have mistake is being in real EOS-ecosystem using E-SCV.

6.8 Limitation and Evaluation

These methodologies have two limitations. First is that when the smart contract has constrained access with the condition of status. and the second case is that when each context is connected with databases directly. The ”constrained access” means that If the APIs that are check parameters exist before sensitive data, there is no bug from here. Even If the external parameter reaches the sensitive functions. To figure out this problem. we should have check a precondition APIs but this solution remains in future works. The ”Each context is connected with databases directly” means that If a function loads

the data from the smart contracts database and inserts the data into the sensitive functions then E-SCV could not get detect reachability to sensitive functions. because the values don't directly come from the external parameters.

First, We do search that the argument is simply passed to the sensitive functions which the externally called functions. And We simply collect how many patterns are being in a total of 2960 smart contracts. The number of the result is written in TotalAPIcalls in [Table 6.1](#).

However, Not All arguments of function which is pass to sensitive functions are not vulnerable. Each function becomes vulnerable depending on whether the hacker's parameter is passed to the specified argument. In the case of db_update_i64, the first argument ram-payer must set as "self" and the second argument which is data must be coming from the external parameters. However, it is hard to directly check the value is "self". thus, We check that the ram-payer is not coming from the outside instead of the value is set by "self". In the same way, db_store_i64 detects functions from which the third argument (payer) does not come from the outside and the fifth argument (buffer) comes from the outside. In the case of "memcpy" is that It detects that 1-argument which means buffer is coming from the external parameters. In case of send_dereffer() which mention of EOS-CPU and EOS-NET drain attack is detected based on the absence of require_auth before the function. Because send_dereffer() API used to use its EOS-CPU and EOS-NET normally. We determined the reachability with these criteria and the result is reachableAPI in table [Table 6.1](#).

Chapter 7. Related work

7.1 Security analysis of blockchain systems

Many previous studies have investigated the security of blockchain systems [32, 43, 7, 10, 38, 27, 30]. Most of them focused on analyzing the two blockchain systems; Bitcoin and Ethereum. There is a large body of work on Bitcoin attacks. Researchers have analyzed double spending [19], network-related attacks [14, 2], illicit transactions [26], transaction malleability [9, 1], and selfish mining [36, 22, 13, 12, 44]. Eyal *et al.* [13] introduced a selfish mining problem in which rational Bitcoin miners collude to control the entire Bitcoin system, including taking over the mining profit. Heilman *et al.* [14] demonstrated the eclipse attack on the Bitcoin network where an adversary can fabricate Bitcoin mining power by spoofing the routing tables of other miners. They utilized this to conduct double spending attacks as well. Kwon *et al.* [22] proposed the fork-after-withholding attack, in which selfish miners secretly keep new blocks as long as possible to maximize their profit.

Previous research has also investigated the security of Ethereum smart contracts [28, 21, 18, 41, 42, 40]. Luu *et al.* [28] introduced an automatic bug finding tool for Ethereum SCs. They aimed to discover bugs that allow colluding miners to intentionally reorder the execution of SCs or to change the timestamp of SCs. They also presented the notorious re-entrancy bug, which allows an adversary to continuously execute a target SC for various purposes. Krupp *et al.* [21] developed a tool that automatically generates exploits for identified SC vulnerabilities, and Kalra *et al.* [18] presented another tool that automatically remediates a target SC by inserting security checks.

Recently, researchers have started analyzing the security of other blockchain systems, such as Ripple [33], Monero [15, 31], and Stellar [20]. In addition, there have been studies on evaluating various consensus algorithms [45], and analyzing the decentrality in permissionless blockchains [23].

In line with this research, we analyzed the EOS.IO system and identified new security concerns that are rooted in the unique characteristics of EOS.IO and also described their attack scenarios. We are planning to develop an analysis platform for EOS.IO SCs, following the previous papers [28, 21].

7.2 Security analysis of EOS.IO

Even though there exist many studies on Bitcoin and Ethereum, there have been only a few studies conducted on EOS.IO. We only found several bugs [35, 17, 25, 4, 34] reported from the industry. These bugs can be categorized as software bugs in the core EOS.IO system [35], logical bugs in EOS.IO SCs [17, 25, 4], and a design bug [35].

Chen *et al.* [35] found an arbitrary code execution bug in EOS.IO. This bug is essentially caused by an integer overflow bug when executing an SC that leads to out-of-bound buffer access. There are bugs from mis-implementation of SCs as well [17, 25, 4]. One example is that an adversary can exploit an SC that does not properly check the caller of the SC, allowing multiple EOS.IO SCs to be executed subordinately [25]. The third category is a bug that stems from the EOS.IO core design, allowing a large number of transactions to delay the block creation time [34]. This appears to be similar to our block delay attack; however, they essentially differ in that the block delay attack pauses BPs by exploiting their transaction scheduling algorithm. Hence, it can effectively break the system down.

In this paper, we documented four new threats and attacks. Even though some of them seem trivial, their underlying causes are related to the EOS.IO system design. We also addressed specific design concerns with the current EOS.IO system and noted how fixing them would address potential threats to EOS.IO .

Conclusion

Building a secure system demands an understanding of potential threats and their underlying root causes. We conducted the first study that analyzes the security of the EOS.IO system, which has recently become one of the most popular blockchain systems. We presented three novel attack methods along with one concrete attack scenario, abusing the controversial EOS.IO design policy. All the attacks exploit new system components, including EOS-CPU, EOS-RAM, and BP scheduling, which are unique to EOS.IO. We also conducted the experiments for each attack, thereby demonstrating their severities. Finally, we proposed the possible mitigations of the presented threats.

In this paper, we manually checked several smart contracts whether they are vulnerable to the presented attacks. However, we believe there still exist other vulnerable SCs that have yet to be discovered. We leave finding vulnerable EOS.IO SCs on a larger scale as a promising future work.

Bibliography

- [1] Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: On the malleability of bitcoin transactions. In: International Conference on Financial Cryptography and Data Security. pp. 1–18. Springer (2015)
- [2] Apostolaki, M., Zohar, A., Vanbever, L.: Hijacking bitcoin: Routing attacks on cryptocurrencies. In: 2017 IEEE Symposium on Security and Privacy (SP 2017). pp. 375–392. IEEE (2017)
- [3] arhag: eosio.code permission argument. <https://github.com/EOSIO/eos/issues/3050> (2018)
- [4] Barkley, A.: 1 billion fake eos tokens used to steal \$58k from decentralised exchange. <https://cryptodaily.co.uk/2018/09/1-billion-fake-eos-tokens-used-to-steal-58k-from-decentralised-exchange> (2018)
- [5] Bastien, J., Gohman, D.: WebAssembly: Here Be Dragons. <https://llvm.org/devmtg/2015-10/slides/BastienGohman-WebAssembly-HereBeDragons.pdf> (2015)
- [6] Top 100 cryptocurrencies by market capitalization. <https://coinmarketcap.com/> (2019)
- [7] classic community, E.: Ethereum Classic Documentation. Self-published (2016)
- [8] EOS Dapps Rankings. <https://dappradar.com/rankings/protocol/eos> (2019)
- [9] Decker, C., Wattenhofer, R.: Bitcoin transaction malleability and mtgox. In: European Symposium on Research in Computer Security. pp. 313–326. Springer (2014)
- [10] EOS.IO Technical White Paper. <https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md> (2018)
- [11] EOS Park - EOS data service provider. <https://eospark.com/>
- [12] Eyal, I.: The Miner’s Dilemma. In: 2015 IEEE Symposium on Security and Privacy (SP 2015). IEEE (2015)
- [13] Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. Communications of the ACM 61(7), 95–102 (2018)
- [14] Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin’s peer-to-peer network. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 129–144 (2015)
- [15] Hinteregger, A., Haslhofer, B.: An empirical analysis of monero cross-chain traceability. arXiv preprint arXiv:1812.02808 (2018)
- [16] Howard, M., Lipner, S.: The security development lifecycle, vol. 8. Microsoft Press Redmond (2006)
- [17] Jing, K.: EOS smart contract development security best practices. https://github.com/slowmist/eos-smart-contract-security-best-practices/blob/master/README_EN.md (2018)
- [18] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS. pp. 18–21 (2018)
- [19] Karame, G.O., Androulaki, E., Capkun, S.: Double-spending fast payments in bitcoin. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 906–917. ACM (2012)
- [20] Kim, M., Kwon, Y., Kim, Y.: Is Stellar As Secure As You Think? IEEE Security and Privacy on the Blockchain (IEEE S&B 2019) (2019)
- [21] Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1317–1333 (2018)
- [22] Kwon, Y., Kim, D., Son, Y., Vasserman, E., Kim, Y.: Be Selfish and Avoid Dilemmas: Fork After Withholding (FAW) Attacks on Bitcoin. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM (2017)
- [23] Kwon, Y., Liu, J., Kim, M., Song, D., Kim, Y.: Impossibility of full decentralization in permissionless blockchains. arXiv preprint arXiv:1905.05158 (2019)
- [24] Larimer, D.: Delegated proof-of-stake (dpos). Bitshare whitepaper (2014)
- [25] Le, Q.: How hackers attack eos contracts and ways to prevent it. <https://medium.com/leclevietnam/hacking-in-eos-contracts-and-how-to-prevent-it-b8663c8bffa6> (2018)

- [26] Lee, S., Yoon, C., Kang, H., Kim, Y., Kim, Y., Han, D., Son, S., Shin, S.: Cybercriminal minds: An investigative study of cryptocurrency abuses in the dark web. In: 26th Annual Network and Distributed System Security Symposium, NDSS (2019)
- [27] Litecoin. <https://litecoin.org/> (2018)
- [28] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 254–269. ACM (2016)
- [29] McClure, S., Scambray, J., Kurtz, G., Kurtz: Hacking exposed: network security secrets and solutions. McGraw-Hill (2009)
- [30] Monero. <https://monero.org/> (2018)
- [31] Möser, M., Soska, K., Heilman, E., Lee, K., Heffan, H., Srivastava, S., Hogan, K., Hennessey, J., Miller, A., Narayanan, A., et al.: An empirical analysis of traceability in the monero blockchain. Proceedings on Privacy Enhancing Technologies 2018(3), 143–163 (2018)
- [32] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
- [33] Osono, U.: Vulnerabilities in bitcoin, ripple, and ethereum digital signatures discovered by researchers. <https://smartereum.com/46548/cryptocurrency-digital-signatures-vulnerabilities-in-bitcoin-ripple-and-ethereum-digital-signatures-discovered-by-researchers-cryptocurrency-news-today/> (2019)
- [34] PeckShield: Eos “transaction congestion attack”: Attackers could paralyze eos network with minimal cost. <https://medium.com/@peckshield/eos-transaction-congestion-attack-attackers-could-paralyze-eos-network-with-minimal-cost-9adfb4d16c82> (2019)
- [35] of Qihoo 360, Y.C.: Eos arbitrary code execution. <http://blogs.360.cn/post/eos-node-remote-code-execution-vulnerability.html> (2018)
- [36] Recabarren, R., Carbunar, B.: Hardening stratum, the bitcoin pool mining protocol. Proceedings on Privacy Enhancing Technologies 2017(3), 57–74 (2017)
- [37] research, B.: The bitcoin cash hardfork – three interrelated incidents (2019), <https://blog.bitmex.com/the-bitcoin-cash-hardfork-three-interrelated-incidents/>
- [38] Schwartz, D., Youngs, N., Britto, A., et al.: The ripple protocol consensus algorithm. Ripple Labs Inc White Paper 5 (2014)
- [39] Set account permission. <https://developers.eos.io/eosio-cleos/reference#cleos-set-account> (2019)
- [40] Suiche, M.: Porosity: A decompiler for blockchain-based smart contracts bytecode. DEF CON 25, 11 (2017)
- [41] Team, C.: Mythril: Security analysis tool for ethereum smart contracts. <https://github.com/ConsenSys/mythril> (2018)
- [42] Trailofbits: Manticore. <https://github.com/trailofbits/manticore> (2017)
- [43] Wood, G.: Ethereum: A Secure Decentralized Generalized Transaction Ledger. Ethereum Project Yellow Paper 151 (2014)
- [44] Zhang, R., Preneel, B.: Publish or perish: A backward-compatible defense against selfish mining in bitcoin. In: Cryptographers’ Track at the RSA Conference. pp. 277–292. Springer (2017)
- [45] Zhang, R., Preneel, B.: Lay down the common metrics: Evaluating proof-of-work consensus protocols’ security. In: 2019 IEEE Symposium on Security and Privacy (SP 2019) (2019)

Acknowledgments in Korean

가장 먼저 학술연수 파견이라는 좋은 기회로 저를 KAIST로 보내주신 삼성 리서치의 황용호 랩장님께 감사를 드리고 싶습니다. 부하 직원을 발전시키는 상사가 좋은 상사라는 말씀을 해주시며 이상섭이라는 한사원에게 2년이라는 긴 시간을 투자해 주신 것은 단순히 존경스러운 상사를 넘어 인생의 멘토로 평생 가슴에 품고 살것 같습니다. 더불어 저를 선택해 주시고, 학교를 들어오기 전부터 많은 조언을 주셨던 김용대 교수님께 특별히 감사의 말씀을 드리고 싶습니다.

둘째로 늘 창의적인 아이디어가 변뜩이는 해킹-크리에이터 (김)대준이, 논문 Writing이란 이런 것이다를 보여준 동관이, writing 더불어 석사 생활의 tipping point를 만들어주신 손수엘 교수님께 무엇보다 감사의 말씀을 전합니다. 이 분들이 없었다면 너무나 많은 것을 놓치고 졸업을 했을 것 같습니다.

셋째로 Sysec SA팀에게 감사의 인사를 드립니다. 분야무관, 모든 과제를 케어하는 정신적 지주 동관이부터, 늘 뛰어난 판단력과 이성적인 조언을 많이 해준 동환이, fact advisor 민근이, 모든 기술들에 대한 답을 가지고 있던 갓 해커 은수, 2년을 함께 지내며 허당인 내가 놓치는 부분을 잘 항상 챙겨 주었던 현기, 곧 회사 메신저에서 볼 요한이, 보안 모든 분야를 팔로우하는 general hacker 창훈이, 주 분야가 아닌 새로운 분야도 모든 빨리 배우고 익숙해지는 브레인 묘준이. 이들 모두 덕에 재미있고 의미 있는 2년을 보내고 졸업하게 되었습니다. 그 외에도 타팀이자, 나이 많은 형인데도 외면해 주지 않고 케어 해준 상욱이, 지호, 도현이, 철준이, 민철이, 호준이, 수빈이, 민정이, (권)유진이 그리고 적진 않았지만 그 외 Syssec 친구들 너무너무 고맙습니다. 그리고 정말 좋은 추억을 만들어준 엄청난 실력+인성까지 반듯한 갓 해커들이 집합: Kaishack 멤버들에게도 너무나 고마움을 표하고 싶습니다. 더불어 1학년 1학기 때 같은 조별 과제로 뭉쳤다가 베스트 프랜드가 된 엘리트들 (오)동현이, 박선녀, 김홍식이! 2년 동안 마음의 안식처 같은 카톡방을 꼽으라면 단연 이친구들의 카톡방이였고 덕분에 재미있는 석사 생활을 보낼 수 있었습니다.

마지막으로 정신적 고민의 순간에 늘 정신적 지주가 되어주신 어머니, 아버지, 그리고 힘들 때는 응원 해주고 어려울때는 안식처가 되어주며 늘 옆에서 지켜준 은이에게 무한한 감사의 말씀을 전하고 싶습니다.

석사생활 동안 제가 가진 것보다 많은 것을 해낼 수 있었던 것은 이렇게 주변의 좋은 친구들 덕이라고 생각하고 있습니다. 카이스트에서 만난 모든 분들에게 너무나 감사드립니다.

Curriculum Vitae in Korean

이 름: 이 상섭
생 년 월 일: 1987년 07월 20일
전 자 주 소: k1rh4.lee@gmail.com

학 력

2017. 9. – 2019. 2. 한국과학기술원 (석사)
2007. 2. – 2013. 8. 홍익대학교 (학사)
2004. 3. – 2007. 2. 돌마고등학교

경 력

2016. 3. – current BoB(Best of Best) Mentor
2014. 3. – current 삼성 리서치 (Security LAB)
2013. 3. – 2014. 2. 삼성전자 (소프트웨어 멤버십 운영자)
2011. 8. – 2012. 2. piosoft(linkhub) - startup
2009. 3. – 2011. 4. 공군 작전사령부 정보보호병

학회활동

1. 2019.08. USENIX WOOT'19 Conference Speaker
2. 2019.08. DEFCON CTF 27 Finalist(KaishHack)
3. 2016, 2017 SCTF 운영 및 문제 출제(Samsung Resurc & KaisHack)
4. 2016, 2017 벨루미나 CTF(invited)
5. 2014.9. CodeEngine PSD hacking Speaker
6. 2013.6. KISA HDCON Finalist
7. 2013.6. WhiteHat contest Finalist
8. 2012-2013. 삼성소프트웨어 멤버십(대전)
9. 2012.3. 홍익대학교 해킹방어대회 학과장상 수상
10. 2012.3. 삼성 소프트웨어 멤버십 기술면접 수상
11. 2010.3. 공군배 해킹방어대회 최우수(참모총장상) 수상

연구업적

1. Sangsup Lee, Daejun Kim, Dongkwan Kim, Sooel Son, and Yongdae Kim, KAIST *Who Spent My EOS? On the (In)Security of Resource Management of EOS.IO*, Master Thesis, Conference on USENIX WOOT'19 2019.