Received August 9, 2018, accepted September 16, 2018, date of publication September 24, 2018, date of current version October 17, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2872044

Efficient and Robust Detection of Code-Reuse Attacks Through Probabilistic Packet Inspection in Industrial IoT Devices

JUN-WON HO
Department of Information Security, Seoul Women's University, Seoul 01797, South Korea
(e-mail: jwho@swu.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2016R1C1B1014126).

ABSTRACT Industrial IoT devices are vulnerable to code-reuse attacks in which benign codes of these devices are reused for malicious activities. In the sense that adversary can compromise industrial IoT devices by means of code-reuse attacks and impair entire industrial IoT ecosystems through the compromised industrial IoT devices, it is very imperative to detect code-reuse attacks in industrial IoT devices. Although different types of code-reuse attack detection schemes have been devised in the literature, they are mainly system level or inefficient/vulnerable network level defense techniques. For the efficient and robust network level defense, we propose a scheme that detects code-reuse attacks efficiently and resiliently by incorporating the sequential probability ratio test (SPRT) with the probabilistic inspection on the packets incoming into industrial IoT devices. Through experimental and analytical study, we demonstrate that our proposed detection scheme resiliently and efficiently defends against code-reuse attacks in industrial IoT devices. In particular, our simulation results show that the SPRT with probabilistic packet inspection achieves at least 93.2% and 99.0% average detection rate for small and large set of code-reuse packets, respectively, while demanding below five samples for detection on an average. They also exhibit that it achieves at most 0.4% average false positives with below four samples on an average.

INDEX TERMS Code-reuse attacks, probabilistic packet inspection, sequential probability ratio test (SPRT).

I. INTRODUCTION

Industrial IoT devices could be deployed for various industrial IoT systems. For instance, they can be used for smart manufacturing, smart grid, smart metering, and smart farming. Moreover, they could be applied to build up smart city and industrial system security. For successful employment of industrial IoT systems, it is inevitable to efficiently and effectively establish industrial IoT systems with using various functionalities of industrial IoT devices. However, attacker can paralyze this industrial IoT system establishment by exploiting the vulnerabilities in industrial IoT devices. In the sense that these industrial IoT devices share the substantial number of vulnerabilities with conventional computing systems, it will not be difficult for attacker to compromise industrial IoT devices by exploiting these vulnerabilities and mount myriad attacks with the compromised devices, making havoc of industrial IoT systems.

In particular, industrial IoT devices are vulnerable to code-reuse attacks [19] that are very deleterious from the perspective that the malign reuse of the normal instructions in devices will incur a variety of damages to those devices. A variety of schemes [1]–[4], [6]–[12], [15], [16], [20], [23], [25] have been contrived to guard against code-reuse attacks. However, the works developed in [1]–[4], [6]–[10], [15], [16], [20], [23], and [25] mainly provide the defenses at system level against code-reuse attacks while not detecting code-reuse attack packets incoming into industrial IoT devices. Thus, this system level defense may not gain the advantage that code-reuse attacks could be detected at the front defense line of packet inspection. Although our prior schemes devised in [11] and [12] are rooted on packet inspection, they are not efficient and not robust, respectively. More specifically, our prior scheme in [11] demands considerable overhead of examining every incoming and outgoing packets for code-reuse attack detection. In [12], there are security drawbacks that attacker can bypass the detection by craftily placing packets for code-reuse attacks. As a result, a new packet inspection scheme is needed for efficient and resilient detection against code-reuse attacks in industrial IoT devices.

To comply with this necessity, we develop a scheme to detect code-reuse attacks by integrating the Sequential Probability Ratio Test (SPRT) [24] with probabilistic packet inspection under the intuition that code-reuse packets contain the addresses of instructions or library functions in industrial IoT devices. More specifically, whenever receiving packets in bytes, each industrial IoT device inspects incoming bytes in accordance with probabilistic selection. It then performs the SPRT with the inspected bytes. The SPRT is deemed to be a decision process in which a decision is statistically brought to completion with null hypothesis and alternate hypothesis. In our proposed scheme, H_1 (resp. H_0) is the alternate (resp. null) hypothesis that the inspected bytes do (resp. not) belong to address space of instructions or library functions. Putting it in different way, H_1 is associated with the hypothesis that packets incoming into industrial devices contain code-reuse packets. H_0 corresponds to the hypothesis that incoming packets do not contain code-reuse packets. The SPRT will go toward the adoption of H_1 (resp. H_0) if it takes the inspected bytes with H_1 (resp. H_0) type, and it eventually put an end to either H_0 or H_1 decision. When H_1 is the final decision of the SPRT, the SPRT detects code-reuse attacks and thereafter the bytes contributing to the H_1 decision are quarantined for further investigation of maliciousness. If they are considered to be used for malicious activities, they will be discarded from industrial IoT devices.

From the perspective of combining the SPRT with probabilistic packet inspection for code-reuse attack detection, our proposed scheme is regarded as network level defense technique and thus it is novel and distinct from the existing work on system level defense. Hence, our proposed scheme takes a key advantage of the network level defense: Code-reuse packets can be detected and eliminated from industrial IoT devices before they come into the systems of these devices and compromise these devices. Moreover, packet inspection is probabilistically performed and thus the overhead incurred by it is not substantial.

We demonstrate that our proposed scheme works efficiently and resiliently by way of simulation and analysis. In particular, the simulation results show that an average detection rate is at least 93.2% and 99.0% while an average number of samples is at most 4.234 and 4.443 for small and large set of code-reuse packets, respectively. Moreover, our proposed scheme fulfills at most 0.4% average false positives with at most 3.984 samples on an average, respectively. From these simulation results, we see that code-reuse attacks are robustly and efficiently detected by the SPRT with probabilistic packet inspection. Moreover, we compare our proposed scheme to our prior work [12]. The comparison results show that the detection rates of our proposed scheme are higher than 3.47 times as much the highest detection rate of [12] as while needing less number of samples for detection than [12].

We first explain the details of the relevant work, background, and our proposed scheme in the following sections. We then put on display of the simulation results of our proposed scheme and make a conclusion on the paper.

II. RELATED WORK

In this section, we give an introduction to diverse relevant works to code-reuse attacks in the literature and give an explanation of why our proposed scheme is still imperative for code-reuse attack detection in spite of substantial number of related works.

Gras *et al.* [8] demonstrate that address space layout randomization (ASLR) technique, which is regarded as defense technique against code-reuse attacks, is vulnerable on AMD, Intel, ARM architectures based on cache. In [2], layout randomization technique, which is resilient to information leakage, is proposed for mobile embedded devices. Crane *et al.* [4] propose defense technique based on layout randomization of code-pointer tables against the reuse attacks of dynamically bound functions. Lu *et al.* [15] develop a method to cease address space leakage that could be occurred when to use ASLR. Seo *et al.* [20] devise a ALSR method tuned to Intel Software Guard Extension (SGX). Lu *et al.* [16] propose a technique of re-randomizing address space layout at runtime to hamper clone-probing attacks against ALSR. Crane *et al.* [3] develop a code randomization technique called Readactor that is resilient to memory leakage. In [10], novel instruction location randomization technique is proposed to hamper return-oriented programming attacks. To guard against code-reuse attacks, the proposed scheme in [9] performs the verification on the target address of every instruction being recognized as possible attack points by binary code analysis in IoT devices.

Snow *et al.* [21] present just-in-time code-reuse attacks that impair the effectiveness of fine-grained ASLR. In [6], code randomization technique called Isomeron is proposed as defense technique against JIT-ROP attacks. Tang *et al.* [23] develop Heisenbyte system to guard against code-reuse attacks. In Heisenbyte system, codes are distorted after code reading, leading to code-reuse attack prevention. Werner *et al.* [25] design No-Execute-After-Read (NEAR) technique to combat against just-in time code-reuse attacks. Snow *et al.* [22] introduce code inference attacks that paralyze the code-reuse defense technique in which codes are destructed after code reading.

Davi *et al.* [5] demonstrate that coarse-grained control-flow integrity defense approaches are not effective against ROP attacks. In [13], demonstrate that gadget-chain length is not accurate metric to be used for code-reuse attack defense. In [17], JIT-ROP attacks and defenses are thoroughly explored. The proposed scheme in [1] defends against code-reuse attacks by checking the integrity of control flows in program codes. In [7], a practical tool called ROPdefender is proposed against return-oriented programming attacks. Jang *et al.* [14] show that kernel ASLR with intel TSX can be paralyzed though a timing attack.

From the perspective that the aforementioned works on code-reuse attack defense are basically thought to be system level protection, they do not consider faster detection method of packet inspection in IoT devices in which attacker can mix code-reuse attack packets with normal packets.

On the other hand, our previously devised schemes [11], [12] are based on packet inspection technique. In [11], code-reuse attack detection technique has been devised to adapt Kullback-Leibler divergence to the incoming and outgoing packets in IoT devices. However, this work incurs substantial overhead of scrutinizing every incoming and outgoing packet in IoT devices. A code-reuse attack detection scheme with using the SPRT [12] has been proposed to reduce the substantial overhead incurred by [11]. However, the SPRT-based detection scheme in [12] has restriction in terms of detection capability. Specifically, it divides the incoming packets into 4-byte or 8-byte blocks and applies the SPRT to inspect whether a 4-byte or 8-byte block matches with the address space of instructions in code regions of IoT devices, and thus the packets for code-reuse attacks can stretch over two adjacent blocks to evade the block-based detection of [12]. In general, attacker can craftily locate the packets for code-reuse attacks on basis of bytes in order to avoid the block-based detection of [12].

We pacify this limitation of [12] by efficiently and resiliently combining the SPRT with probabilistic packet inspection in unit of a byte rather than 4-byte or 8-byte block. In particular, our proposed scheme is more efficient than [12] in terms of how the SPRT is incorporated into the code-reuse attack detection.

III. BACKGROUND

In this section, we begin describing attacker model for our proposed scheme and explain the illustrative example of code-reuse attacks.

A. ATTACKER MODEL

The basic concept of code-reuse attacks is that attacker can carry out malicious activities in the target systems by craftily reusing the executable codes or library functions of target systems. More specifically, attacker makes the chains with the addresses of instruction gadgets or with the arguments and addresses of library functions needed for malicious behavior implementation in target systems, where a gadget is a sequence of instructions. He then injects these chains into target systems by harnessing the vulnerabilities of them, leading to malicious activity execution in target systems. We denote these chains for code-reuse attacks as *code-reuse packets*. Buffer overflow vulnerability is one of the vulnerabilities largely used for code-reuse attacks. We call code-reuse attacks using instruction gadgets return-oriented programming (ROP) attacks. We also call code-reuse attacks using library functions in libc return-to-libc (RTL) attacks. Although that library functions in libc are basically used in RTL attacks, we do not restrict that attacker should only use libc for RTL attacks. Rather, we assume that attacker can create vulnerable programs containing a variety of useful library functions and install these programs into target systems. He then exploit these library functions of vulnerable programs to launch RTL attacks.

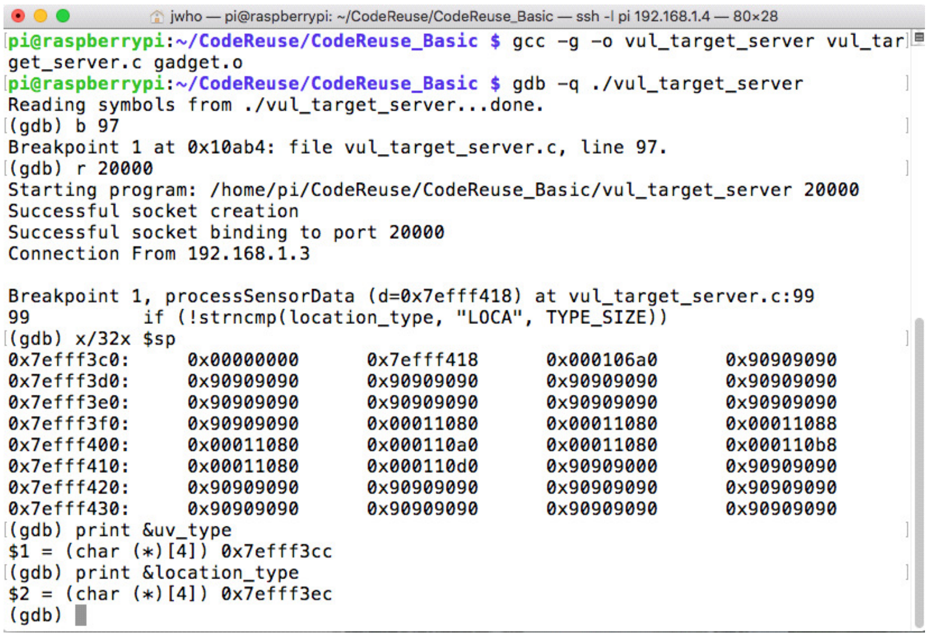
Attacker can compromise the benign devices through code-reuse attacks. More specifically, he can inject code-reuse packets combined with normal packets into the benign devices, leading to the compromise of benign devices. The main rational behind mingling code-reuse packets with the normal packets is to masquerade code-reuse packets as parts of the normal packets, leading to diminish the chance that code-reuse attacks are detected.

B. ILLUSTRATIVE EXAMPLE OF CODE-REUSE ATTACKS

For illustration purpose, let us consider RTL attacks with using instruction gadgets. This scenario is thought of as combination of RTL and ROP attacks. More specifically, we implement attack client program and target server program in Raspberry Pi 3 Model B [18]. All programs are written in C language. To let target server have instruction gadgets and library functions to be reused by attack client, two object codes are linked into target server program such that the first object code contains instruction gadgets of *pop lr* and *pop pc* and the second one contains library functions of *sys_df*, *sys_ps*, *sys_uname*, *sys_sh*, which is defined as a *system* function with argument */bin/df /bin/ps*, */bin/uname -a*, */bin/sh*, respectively. Attack client (resp. target server) program runs on Pi machine with IP address of 192.168.1.3 (resp. 192.168.1.4). Target server is a simple server in which it receives 9 types of sensor data sent by clients and stores each type of data into separate file. 9 types of sensor data are assumed to be location, image, air quality, sound, infrared, temperature, humidity, water quality, and ultraviolet.

Furthermore, target server uses 9 strcpy functions to process sensor data from clients and all these strcpy functions have buffer overflow vulnerabilities. Attack client sends target server code-reuse packet with total size of 396 bytes. Code-reuse packet is composed of 9 sections such that each section with 44 byte size is disguised with each sensor data type with 4 byte size and each sensor data with 40 byte size. Attack client exploits buffer overflow vulnerability of strcpy function to copy each section into 4 byte array in stack of target server and thus 40 bytes overflow per section occur in stack of target server. Code-reuse packet is first initialized to 0x90, which is NOP (no operation) code. The last 4 byte of each section in code-reuse packet is then craftily reset to one of the addresses of *{pop lr, pop pc}*, *sys_df, sys_ps, sys_uname, sys_sh* for code-reuse attacks, where *lr* and *pc* indicate *link register* and *program counter*, respectively. The last 4 bytes of 9 sections in code-reuse packet will be eventually placed in the target server's stack.

Figure 1 displays the snapshot of stack in target server after code-reuse packet is located inside target server's stack though buffer overflow vulnerability. From the snapshot in Figure 1, we construct Table 1 to figure out the stack addresses for the last 4 bytes of 9 sections in code-reuse packet. As shown in Table 1, the stack region from 0x7efff3cc to 0x7efff3ec is for 9 types of sensor data and thus the


jwho — pi@raspberrypi: ~/CodeReuse/CodeReuse_Basic — ssh -l pi 192.168.1.4 — 80×28

```
pi@raspberrypi:~/CodeReuse/CodeReuse_Basic $ gcc -g -o vul_target_server vul_tar
get_server.c gadget.o
pi@raspberrypi:~/CodeReuse/CodeReuse_Basic $ gdb -q ./vul_target_server
Reading symbols from ./vul_target_server...done.
(gdb) b 97
Breakpoint 1 at 0x10ab4: file vul_target_server.c, line 97.
(gdb) r 20000
Starting program: /home/pi/CodeReuse/CodeReuse_Basic/vul_target_server 20000
Successful socket creation
Successful socket binding to port 20000
Connection From 192.168.1.3

Breakpoint 1, processSensorData (d=0x7efff418) at vul_target_server.c:99
99          if (!strncmp(location_type, "LOCA", TYPE_SIZE))
(gdb) x/32x $sp
0x7efff3c0:     0x00000000      0x7efff418      0x000106a0      0x90909090
0x7efff3d0:     0x90909090      0x90909090      0x90909090      0x90909090
0x7efff3e0:     0x90909090      0x90909090      0x90909090      0x90909090
0x7efff3f0:     0x90909090      0x00011080      0x00011080      0x00011088
0x7efff400:     0x00011080      0x000110a0      0x00011080      0x000110b8
0x7efff410:     0x00011080      0x000110d0      0x90909000      0x90909090
0x7efff420:     0x90909090      0x90909090      0x90909090      0x90909090
0x7efff430:     0x90909090      0x90909090      0x90909090      0x90909090
(gdb) print &uv_type
$1 = (char (*)[4]) 0x7efff3cc
(gdb) print &location_type
$2 = (char (*)[4]) 0x7efff3ec
(gdb) 
```

FIGURE 1. **Stack snapshot of target server after code-reuse packets are placed in the stack of target server through buffer overflow vulnerability.**

TABLE 1. **Stack addresses for the last 4 bytes of 9 sections in code-reuse packets.**

Addresses	Last 4 bytes of 9 sections in code-reuse packet
0x7efff3f4	address of {*pop lr, pop pc*} instruction gadget set
0x7efff3f8	address of {*pop lr, pop pc*} instruction gadget set
0x7efff3fc	address of *sys_df* library function
0x7efff400	address of {*pop lr, pop pc*} instruction gadget set
0x7efff404	address of *sys_ps* library function
0x7efff408	address of {*pop lr, pop pc*} instruction gadget set
0x7efff40c	address of *sys_uname* library function
0x7efff410	address of {*pop lr, pop pc*} instruction gadget set
0x7efff414	address of *sys_sh* library function

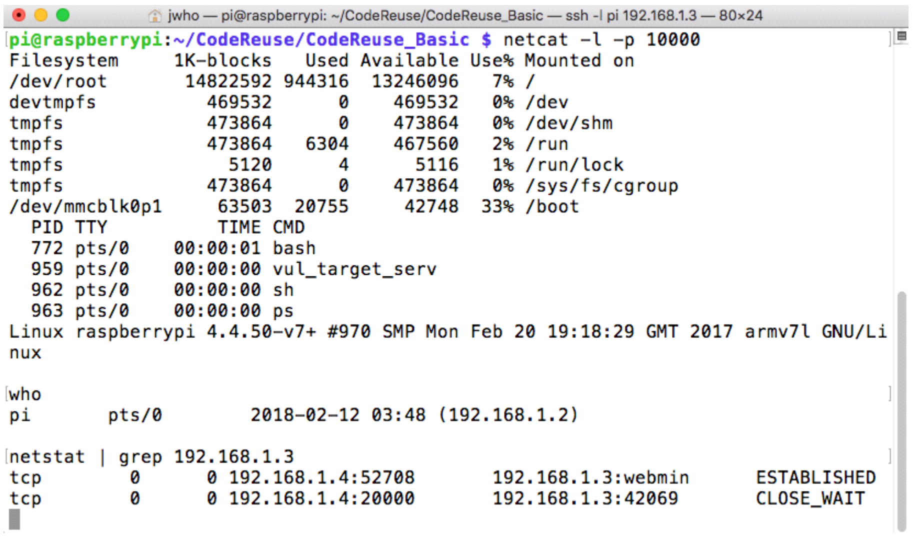
overflow region starts at 0x7efff3f0. Indeed, the information for *lr* and *pc* reside at 0x7efff3f0 and 0x7efff3f4, respectively. Hence, the last 4 bytes of the first section in code-reuse packet are placed in the stack region at 0x7efff3f4, the last 4 bytes of the following 8 sections consecutively reside in region from 0x7efff3f8 to 0x7efff414. Note that 9 strcpy functions with buffer overflow vulnerability are called in *processSensorData* function of target server. When *processSensorData* function terminates, address of *{pop lr, pop pc}* at 0x7efff3f4 is popped from stack in target server and *pc* is set to the popped address of 0x00011080. Once the process control hits 0x00011080, *pop lr* and *pop pc* instructions are executed in turn and hence *pc* is set to *sys_df*'s address of 0x00011088. Also, *lr* is set to address of *{pop lr, pop pc}*.

Once the process control hits 0x00011088, *sys_df* function is executed. When *sys_df* function terminates, *pc* is set to *lr* and thus it becomes 0x00011080. Once the process control hits 0x00011080, *pop lr* and *pop pc* instructions are executed in turn and hence *pc* is set to *sys_ps*'s address of 0x000110a0. Also, *lr* is set to address of *{pop lr, pop pc}*. Once the process control hits 0x000110a0, *sys_ps* function is run. The above chaining process from the execution of *sys_df* to the execution of *sys_ps* is similarly applied to the executions of *sys_uname, sys_sh*. We discern that *{pop lr, pop pc}* instruction gadget set plays a role of transferring the process control from one library function to another library function.

Figure 2 shows the outputs of code-reuse attacks in the terminal of attack client. In target server, *sys_df, sys_ps, sys_uname* library functions are executed in turn. Target server uses *dup2* system call to redirect the execution results of these functions to netcat program running on attack client. After these three library functions' executions, *sys_sh* library function is finally run and thus attack client is able to gain the control of target server. To demonstrate that target server is under the control of attack client, we run *who* and *netstat | grep 192.168.1.3* commands and confirm that the outputs of these commands are displayed in the terminal of attack client.

IV. CODE-REUSE ATTACK DETECTION BY INTEGRATING THE SPRT INTO PROBABILISTIC PACKET INSPECTION

Although our proposed scheme can be applied to industrial IoT devices with any size of address space of instruction


jwho — pi@raspberrypi: ~/CodeReuse/CodeReuse_Basic — ssh -l pi 192.168.1.3 — 80×24

```
pi@raspberrypi:~/CodeReuse/CodeReuse_Basic $ netcat -l -p 10000
Filesystem     1K-blocks   Used Available Use% Mounted on
/dev/root       14822592 944316  13246096   7% /
devtmpfs          469532      0    469532   0% /dev
tmpfs             473864      0    473864   0% /dev/shm
tmpfs             473864   6304    467560   2% /run
tmpfs               5120      4      5116   1% /run/lock
tmpfs             473864      0    473864   0% /sys/fs/cgroup
/dev/mmcblk0p1     63503  20755     42748  33% /boot
  PID TTY          TIME CMD
  772 pts/0    00:00:01 bash
  959 pts/0    00:00:00 vul_target_serv
  962 pts/0    00:00:00 sh
  963 pts/0    00:00:00 ps
Linux raspberrypi 4.4.50-v7+ #970 SMP Mon Feb 20 19:18:29 GMT 2017 armv7l GNU/Li
nux

who
pi       pts/0        2018-02-12 03:48 (192.168.1.2)

netstat | grep 192.168.1.3
tcp        0      0 192.168.1.4:52708       192.168.1.3:webmin      ESTABLISHED
tcp        0      0 192.168.1.4:20000       192.168.1.3:42069       CLOSE_WAIT
```

FIGURE 2. **After code-reuse attacks are successfully mounted against target server, the execution results of *sys_df, sys_ps, sys_uname* library functions in target server are displayed in the terminal of attack client, and then attack client gains control of target server upon the execution of *sys_sh* library function.**

gadgets and library functions, we focus on the devices with 4-byte address space of instruction gadgets and library functions for simplicity.

A. SCHEME DESCRIPTION

In this section, we describe how the SPRT is integrated into probabilistic packet inspection to detect code-reuse attacks.

We first define the bth *address-type byte* as a byte falling within the address space of bth bytes out of 4-byte addresses of instruction gadgets and library functions in ROP attacks and RTL attacks as mentioned in Section III-A, respectively $(1 \leq b \leq 4)$. Moreover, we define *address-type block* as a 4-byte set consisting of the first, second, third, fourth address-type bytes. Address-type block is used as a manifestation that code-reuse packets are coming into industrial IoT devices.

For probabilistic packet inspection, we break a series of incoming packets into a series of bytes. Each time industrial IoT device receives a series of incoming bytes, it performs probabilistic packet inspection to decide whether these incoming bytes include address-type blocks or not. The specific procedure for probabilistic packet inspection is as follows: It selects a byte for inspection with probability p. If the byte is chosen and it is the first address-type byte, industrial IoT device checks whether the subsequent three incoming bytes are the second, third, fourth address-type bytes, respectively. If so, an address-type block is formed from these four address-type bytes. Otherwise, an address-type block is not formed. Industrial IoT device repeats probabilistic packet inspection procedure from the byte following the byte that succeeds or fails to form address-type block.

Now, we explain how the SPRT is integrated with probabilistic packet inspection procedure. In the SPRT, H_1 (resp. H_0) is defined as the alternate (resp. null) hypothesis that a sequence of incoming bytes do (resp. not) include address-type block. We also define a Bernoulli random variable W_k $(k \geq 1)$, which is considered to be a sample in the SPRT, as follows:

We set $W_k = 1$ if the following two conditions are all satisfied. The first condition is that kth byte selected in probabilistic packet inspection procedure is the first address-type byte and subsequently address-type block is formed. The second condition is that W_k is the first sample with H_1 type value of 1 or there is a previous sample $W_m = 1$ $(1 \leq m \leq k-1, k \geq 2)$ such that $k - m \leq \tau$ holds. Note that τ is a number of bytes used for regulating the number of H_1 type samples. By controlling the creation of H_1 type samples through τ, we can reduce the chance that benign packets are misclassified as code-reuse packets.

We set $W_k = 0$ if the kth byte selected in probabilistic packet inspection procedure is the first address-type byte and subsequently address-type block is not formed.

$z = \Pr(W_k = 1) = 1 - \Pr(W_k = 0)$ hold under the definition of success probability z of the Bernoulli distribution. Given two thresholds of z_0 and z_1 $(z_0 < z_1)$, the SPRT gets to accept H_0 (resp. H_1) if $z \leq z_0$ (resp. $z \geq z_1$) holds. We have the log-probability ratio I_g on $g(g \geq 1)$ samples such that

$$I_g = \ln \frac{\Pr(W_1, \ldots, W_g | H_1)}{\Pr(W_1, \ldots, W_g | H_0)}$$

In the SPRT, we have a user-configured false positive rate α' and a user-configured false negative rate β'. If $W_g \leq \ln \frac{\beta'}{1-\alpha'}$, the SPRT accepts H_0 and it starts the test process again with new samples. If $W_g \geq \ln \frac{1-\beta'}{\alpha'}$, the SPRT ends in the acceptance of H_1. Upon the completion of the SPRT with H_1 decision, probabilistic packet inspection procedure is also terminated and we quarantine the bytes leading to the H_1 decision while inspecting the maliciousness of these bytes. If they are judged to be used for malicious activities, they are disused from industrial IoT devices. Otherwise, it keeps on the testing process.

In probabilistic packet inspection procedure, an incoming byte is chosen with probability p. This means that the selection of an incoming byte is independent to other byte selection and the selection probability p is identically applied to each incoming byte. As a result, we can make use of independent and identically distributed (i.i.d.) assumption for incoming byte selection process and thus W_k is assumed to be i.i.d.

Given this i.i.d. assumption, the SPRT can be converted as follows, where N_g is defined as the number of times that $W_k = 1$ in the g samples: If $N_g \leq \frac{\ln \frac{\beta'}{1-\alpha'} + g \ln \frac{1-z_0}{1-z_1}}{\ln \frac{z_1}{z_0} - \ln \frac{1-z_1}{1-z_0}}$, the SPRT accepts H_0 and starts test process again. If $N_g \geq \frac{\ln \frac{1-\beta'}{\alpha'} + g \ln \frac{1-z_0}{1-z_1}}{\ln \frac{z_1}{z_0} - \ln \frac{1-z_1}{1-z_0}}$, the SPRT accepts H_1 and finishes up the test process. Otherwise, the SPRT keeps on the test process.

B. SECURITY ANALYSIS

In this section, we describe the error rates of the SPRT and discuss about the detection capability and false positives of our SPRT-based detection scheme against RTL and ROP attacks.

According to [24], the SPRT generally consummates low false positive rate α and low false negative rate β such that α and β are bounded above by $\frac{\alpha'}{1-\beta'}$ and $\frac{\beta'}{1-\alpha'}$, respectively, and thus we have $\alpha = \beta = 0.01$ when $\alpha' = \beta' = 0.01$.

Next, we explore the detection capability of our proposed scheme on code-reuse attacks. For simplicity, we assume that the probabilistic packet inspection procedure starts at the first byte in code-reuse packet. In ROP attacks, code-reuse packet is a sequence of addresses of instruction gadgets and thus it consists of only address-type blocks. Accordingly, probabilistic packet inspection procedure selects only address-type blocks from code-reuse packet. As long as τ is configured in such a way that a H_1 type sample is created per selected address-type block, the SPRT takes the consecutive H_1-type samples from the address-type blocks chosen through the probabilistic packet inspection procedure. Since the SPRT accepts H_1 if $N_g \geq \frac{\ln \frac{1-\beta'}{\alpha'} + g \ln \frac{1-z_0}{1-z_1}}{\ln \frac{z_1}{z_0} - \ln \frac{1-z_1}{1-z_0}}$, ROP attacks are detected if $g \geq \frac{ln \frac{1-\beta'}{\alpha'}}{\ln \frac{z_1}{z_0}}$ holds under g consecutive H_1-type samples inside code-reuse packet. This implies that ROP attacks are detected if the number of the address-type blocks consecutively sampled by the SPRT within code-reuse packet reaches $\lceil \frac{ln \frac{1-\beta'}{\alpha'}}{\ln \frac{z_1}{z_0}} \rceil$. When $\alpha' = \beta' = 0.01$ and $z_0 = 0.1, z_1 = 0.9$, only three address-type blocks inside code-reuse packet are enough for ROP attack detection.

In RTL case, code-reuse packet is a sequence of addresses and arguments of library functions. We assume that the arguments of library functions are basically filled with non-address-type bytes. This assumption is reasonable in the sense that most library functions rarely use the arguments containing address-type bytes. Under this assumption, probabilistic packet inspection procedure chooses only address-type blocks from code-reuse packet because the arguments of library functions consist of non-address-type bytes and thus they are not selected through probabilistic packet inspection procedure. Accordingly, as in ROP case, as long as τ is configured in such a way that a H_1 type sample is created per selected address-type block, all samples taken by the SPRT are successive H_1-type samples from address-type-blocks determined by the probabilistic packet inspection procedure. As a consequence, we can apply the analysis for ROP case to RTL case.

Finally, we look into the false positive issue of our proposed scheme. Since our proposed scheme runs on industrial IoT devices, most of data exchanged between industrial IoT devices will be likely sensor data. As a result, they will not likely contain address-type blocks, leading to low false positive rate when to deploy our proposed scheme.

V. PERFORMANCE EVALUATION

In this section, we first depict the evaluation environments, and then represent evaluation results of our proposed scheme and the comparison results to our previous work [12].

A. EVALUATION ENVIRONMENTS

We perform the evaluation on our proposed scheme through a plain simulation program. We also compare our proposed scheme to our previous work [12]. From the perspective that our proposed scheme can be thought of as the byte-basis SPRT while our previous work [12] can be regarded as the block-basis SPRT, we call our proposed scheme *Byte-SPRT* and call our previous work [12] *Block-SPRT*. For our evaluations of the Byte-SPRT and the Block-SPRT, we consider three cases such as *Benign*, *ROP*, and *RTL* cases. In all three cases of the Byte-SPRT and the Block-SPRT, we set the total number of bytes incoming into industrial IoT device to 10000 and set a block size to 4 bytes.

We first describe the simulation parameters for our proposed Byte-SPRT. In the Byte-SPRT, we configure a number of bytes $\tau = 300$ for governing the number of H_1 type samples. We also set byte selection probability p from 0.1 to 0.5 in increases of 0.1. In order to emulate the possible case in network traffic, we perform *address-type byte setup* process in all three cases as follows: We choose N_c uniformly at random from the range of [0, 5), and then select N_c non-address-type incoming bytes uniformly at random and set them to the one selected from the first, second, third, fourth address-type bytes uniformly at random. Additionally, we repeatedly apply address-type byte setup process to the cases of two-consecutive, three-consecutive, four-consecutive incoming bytes.

In Benign case of the Byte-SPRT, we denote the number of address-type blocks by N_b. We consider small and large N_b sets. Small N_b set is composed of $N_b = 5, 4, 3, 2, 1$ when $p = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively. Large N_b set consists of $N_b = 10, 8, 6, 4, 2$ when $p = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively. We set N_b in accordance with p in such a way that the larger N_b is configured when we have the lower p. The main rationale behind these settings is that the more number of address-type blocks are likely selected as the lower p is configured. The reason why we break N_b sets into small and large ones is to perform the evaluation under the relatively small and large values of N_b. The reason why the value of N_b is at most 10 is that the cases of H_1 in benign blocks are not common and thus we configure the maximum value of N_b to as small number as possible. We perform *individual address-type block setup* process in which we select N_b four-consecutive non-address-type incoming bytes uniformly at random and convert these selected bytes into N_b address-type blocks.

In ROP case of the Byte-SPRT, we denote the number of address-type blocks by N_o. We consider small and large N_o sets. Small N_o set is composed of $N_o = 50, 40, 30, 20, 10$ when $p = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively. Large N_o set consists of $N_o = 100, 80, 60, 40, 20$ when $p = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively. The main rationale behind these settings of N_o is similar to the one described in Benign case. We then run *continuous address-type block setup* process in which we choose N_o continuous four-consecutive non-address-type incoming bytes uniformly at random and turn these selected bytes into N_o continuous address-type-blocks.

In RTL case of the Byte-SPRT, we denote the number of address-type blocks by N_t. We consider small and large N_t sets. Small N_t set is composed of $N_t = 50, 40, 30, 20, 10$ when $p = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively. Large N_t set consists of $N_t = 100, 80, 60, 40, 20$ when $p = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively. The main rationale behind these settings of N_t is similar to the one explained in Benign case. We then execute *discrete address-type block setup* process in which we select N_t continuous four-consecutive non-address-type incoming bytes uniformly at random and change these chosen bytes into N_t address-type block groups such that a group consists of an address-type block followed by $4 \times N_l$ argument bytes of library functions, where N_l is the number of arguments of library functions and it is picked uniformly at random from the range of [0, 4).

The reason why the values of N_o and N_t are at most 100 is that the cases of H_1 in ROP and RTL blocks are common and hence we set the maximum values of N_o and N_t to as reasonably enough number as possible. We set the arguments of library functions to have no address-type bytes according to the assumption in Section IV-B. Indeed, attacker can put more than one address-type block into a group in such a way that address-type blocks are consecutively placed in a group. However, each address-type block group contains only one address-type block in RTL case. From the perspective that the less number of address-type blocks in a group leads to the lower detection rate, our RTL case with one address-type block is regarded as the worst case in terms of RTL attack detection. In other words, we perform evaluation of our detection scheme against the best scenario of RTL attacks.

As far as the SPRT configuration parameters of the Byte-SPRT are concerned, we set $\alpha' = \beta' = 0.01$ and $z_1 = 0.9$. We also configure $z_0 = 0.05$ in cases that $(p, N_b) = (0.1, 5), (0.5, 1), (p, N_o) = (p, N_t) = (0.1, 50), (0.5, 10)$. We set $z_0 = 0.1$ in all other cases except the cases of $z_0 = 0.05$. The main rationale behind these settings is to boost the SPRT detection capability by having very low value of z_0 when address-type blocks are sampled with very low probability.

Next, we explain the simulation parameters for the Block-SPRT [12]. We let the first block starts from the simulation. In Benign case of the Block-SPRT, we set $N_b = 1, 2, 3, 4, 5$. In RTL and ROP cases of the Block-SPRT, we set $N_t = N_o = 10, 20, 30, 40, 50$. Additionally, the Block-SPRT employs the same address-type byte setup, individual address-type block setup, continuous address-type block setup, discrete address-type block setup processes as in the Byte-SPRT. In all these settings of the Block-SPRT, we fix ϵ to 1.0. In the sense that a sample in the Block-SPRT is obtained from a block with probability ϵ, $\epsilon = 1.0$ indicates that the Block-SPRT utilizes every block for code-reuse attack detection. The main rational behind this ϵ configuration is to compare the Byte-SPRT to the best case of the Block-SPRT in terms of code-reuse attack detection capability. In the Block-SPRT, we configure $\alpha' = \beta' = 0.01$ and $z_1 = 0.9$. We also set $z_0 = 0.05$ in cases that $N_b = 1, 5$ and $N_o = N_t = 10, 50$. We set $z_0 = 0.1$ in all other cases except the cases of $z_0 = 0.05$.

Note that simulation configurations used in [12] are different to the ones specified in this section. In [12], the fraction of the number of address-type blocks over the total number of entire packets in code-reuse attack case is approximately 0.5 on an average, whereas it is at most 0.04 in our simulation settings. $\epsilon = 0.1, 0.15, 0.2$ in [12], whereas $\epsilon = 1$ in our simulation configurations. Moreover, code-reuse packets are randomly distributed in unit of a block in [12], whereas code-reuse packets for RTL and ROP cases are randomly distributed in unit of bytes in our simulation configurations. Overall, our simulation configurations for code-reuse packets are harsher than the ones in [12] in terms of code-reuse attack detection. As a consequence, the evaluation results of the Block-SPRT through our simulation will be worse than the ones in [12].

In both the Byte-SPRT and the Block-SPRT, the simulation has been repeated 1000 times and an average results of 1000 iterations are presented in the following section.

B. EVALUATION RESULTS

We employ the following metrics to evaluate code-reuse attack detection scheme based on the SPRT and packet inspection.

- Number of Samples for the SPRT decision: an average number of samples for the SPRT to accept H_0 Hypothesis in Benign case, H_1 hypothesis in RTL and ROP cases.
- False positive rate (%): an average error rate for benign packets to be misjudged as code-reuse packets.

- Detection rate (%): an average rate for detecting code-reuse packets.
- No decision rate (%): an average rate for the SPRT not to make a decision.
- Fraction of selected address-type bytes: an average fraction of the number of address-type bytes selected for the SPRT over the total number of bytes incoming into industrial IoT device.

False positive rate is calculated as the percentage of the number of times that benign case is misidentified as code-reuse attack case plus the one that it is not decided by the SPRT out of 1000 executions. Detection rate is computed as the percentage of the number of times that code-reuse attack case is detected by the SPRT out of 1000 executions. Note that we classify the case that the SPRT does not make a decision as being undetected.

Furthermore, note that the number of address-type bytes selected for the SPRT is measured from the start of simulation to the end of simulation when the SPRT adopts H_0 decision or does not reach a decision. On the other hand, the numbers of address-type bytes selected for the SPRT in RTL an ROP cases represent are measured from the start of simulation to the termination of the SPRT in case of adoption of H_1 decision.

In Benign case of the Byte-SPRT, the false positive rate is measured as 0.4% when $N_b = 5$ and $p = 0.1$. It is measured as 0% in all other configurations. Furthermore, no decision rate is observed from 67.5% to 98.4% in case of small N_b set and from 70% to 99.2% in case of large N_b set. This indicates that the Byte-SPRT does not reach a decision in large number of executions in Benign case while incurring very low false positive error. This is mainly because the combination of small number of the first address-type bytes and byte selection probability p with at most 0.5 makes it difficult for enough number of samples for decision to be collected, having the Byte-SPRT come to no decision. On the other hand, in Benign case of the Block-SPRT, the false positive is measured as 0% in all configurations and no decision rate is measured as 0%, which means that the Block-SPRT makes a correct decision for each execution in Benign case. This is mainly due to the fact that $\epsilon = 1$ makes each block correspond to each sample and thus the Block-SPRT gets sufficient number of samples for decision.

As show in Figure 3, number of samples for the Byte-SPRT and the Block-SPRT in Benign case is measured as at most 3.428 and at most 3.003 in all settings of small N_b set, respectively. From these observations, number of samples in the Byte-SPRT with small N_b set is slightly more than the one in the Block-SPRT. However, note that our proposed Byte-SPRT with small N_b set still requires a small number of samples for H_0 decision in Benign case, which is below 3.5 on an average. Figure 11 displays the comparison results on number of samples between small and large N_b sets in the Byte-SPRT. We first perceive that the Byte-SPRT needs at most 3.984 samples on an average in all settings of p, small and large N_b sets. Except the case of $p = 0.1$, number of samples in small N_b set tends to be lower than the one in large N_b set. From this observation, we see that the less number of address-type blocks tends to result in the smaller number of samples for decision in Benign case of the Byte-SPRT. We also discern that number of samples in large N_b set is more dynamically affected by p than the one in small N_b set. We infer from this observation that a rise in number of address-type blocks contributes to dynamic effect by p in Benign case of the Byte-SPRT.

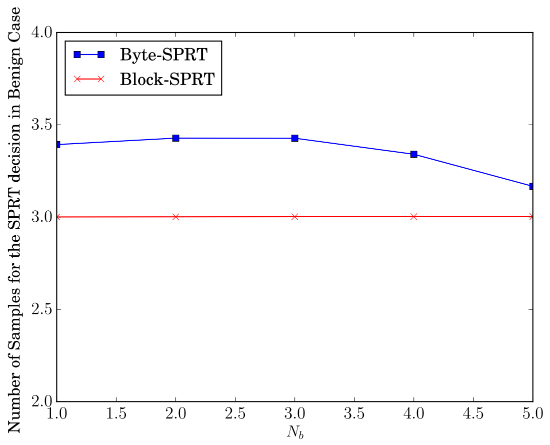


FIGURE 3. Comparing the number of samples for the Byte-SPRT decision to the one for the Block-SPRT decision while varying the number of address-type blocks (N_b) in Benign case.

In RTL case of the Byte-SPRT, no decision rate is observed from 1.2% to 6.8% in case of small N_t set and at most 1% in case of large N_t set. This signifies that the Byte-SPRT does not reach a decision in a very small number of executions in small N_t set while it reaches a decision in almost all executions in large N_t set. In RTL case of the Block-SPRT, no decision rate is measured as 0%, which indicates that the Block-SPRT makes a decision for each execution in RTL case.

As shown in Figure 4, number of samples for the Byte-SPRT and the Block-SPRT to detect RTL attacks is measured as at most 4.201 and at most 8.737 in all settings

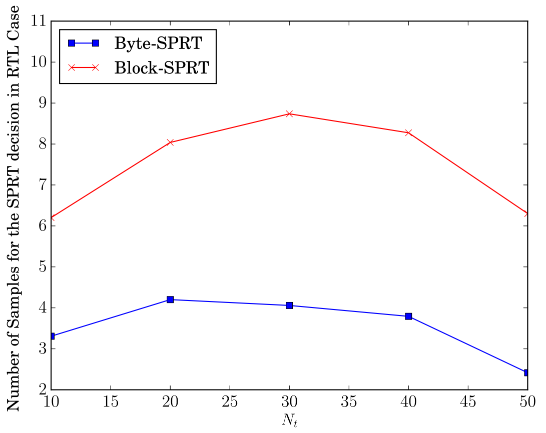


FIGURE 4. Comparing the number of samples for the Byte-SPRT decision to the one for the Block-SPRT decision while changing the number of address-type blocks (N_t) in RTL case.

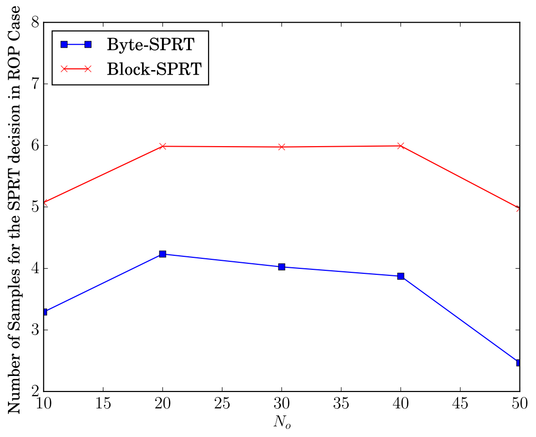


FIGURE 5. **Comparing the number of samples for the Byte-SPRT decision to the one for the Block-SPRT decision while changing the number of address-type blocks (N_o) in ROP case.**

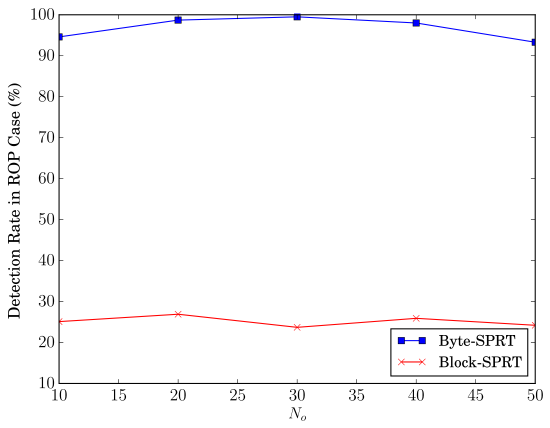


FIGURE 7. **Comparing detection rate (%) of the Byte-SPRT to one of the Block-SPRT while changing the number of address-type blocks (N_o) in ROP case.**

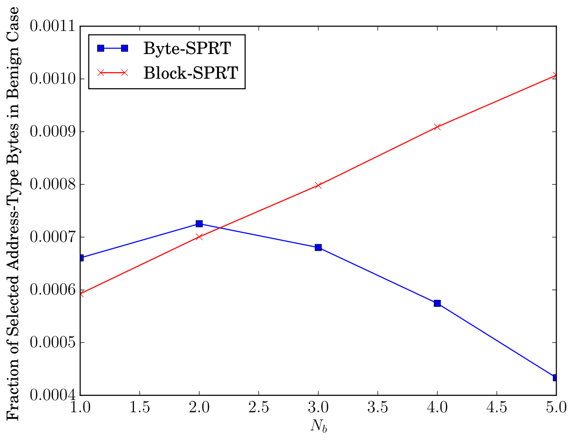


FIGURE 8. **Comparing the fraction of selected address-type bytes of the Byte-SPRT to the one of the Block-SPRT while varying the number of address-type blocks (N_b) in Benign case.**

of small N_t set, respectively. These measurements indicate that the Byte-SPRT with small N_t set fulfills faster RTL attack detection than the Block-SPRT while requiring less than half of the Block-SPRT in terms of the maximum value of number of samples. We also see that the Block-SPRT is more considerably affected by N_t than the Byte-SPRT with small N_t set. Figure 12 shows the comparison results on an average number of samples between small and large N_t sets in the Byte-SPRT. We first discern that the Byte-SPRT needs at most 4.341 samples on an average in all settings of p, small and large N_t sets. We also figure out that number of samples in small N_t set is smaller than the one in large N_t set in cases of $p = 0.1, 0.5$. From this observation, we see that lower value of $z_0 = 0.05$ in cases of $(N_t, p) = (50, 0.1), (10, 0.5)$ leads to the smaller number of samples than the higher value of $z_0 = 0.1$ in other cases of (N_t, p).

As shown in Figure 6, RTL attack detection rate of the Byte-SPRT with small N_t set ranges from 93.2% to 98.4% while detection rate in the Block-SPRT ranges from 10.4% to 23.5%. This signifies that the lowest detection rate of our proposed Byte-SPRT with small N_t set is higher than 3.96 times as much the highest detection rate of the Block-SPRT as. Figure 14 displays the comparison results between the detection rates of small and large N_t sets in the RTL case of the Byte-SPRT. We observe that the Byte-SPRT with large N_t set detects RTL attacks at least 99.0% rate. We also perceive that detection rates in large N_t set are higher than the ones in small N_t set. We infer from this that the more number of address-type blocks leads to the higher RTL attack detection.

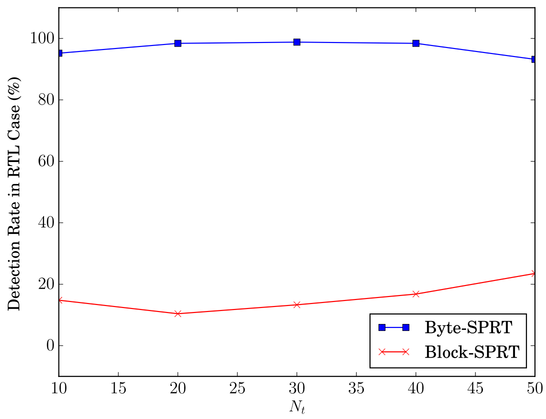


FIGURE 6. **Comparing detection rate (%) of the Byte-SPRT to the one of the Block-SPRT while changing the number of address-type blocks (N_t) in RTL case.**

In ROP case of the Byte-SPRT, no decision rate is observed from 0.4% to 6.7% in case of small N_o set and at most 0.6% in case of large N_o set. This indicates that the Byte-SPRT does not reach a decision in a few number of executions in small N_o set while it reaches a decision in almost all executions in large N_o set. In ROP case of the Block-SPRT, no decision rate is measured as 0%, which means that the Block-SPRT comes to a decision for each execution in ROP case.

As shown in Figure 5, number of samples for the Byte-SPRT and the Block-SPRT to detect ROP attacks is

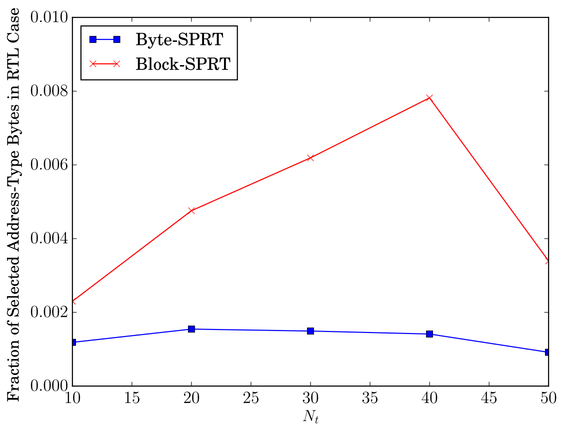


FIGURE 9. Comparing the fraction of selected address-type bytes of the Byte-SPRT to the one of the Block-SPRT while varying the number of address-type blocks (N_t) in RTL case.

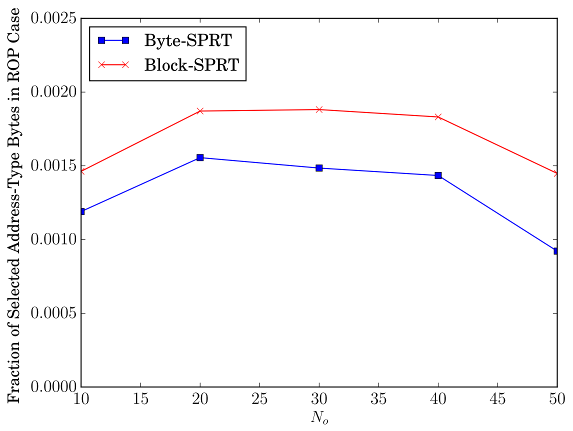


FIGURE 10. Comparing the fraction of selected address-type bytes of the Byte-SPRT to the one of the Block-SPRT while varying the number of address-type blocks (N_o) in ROP case.

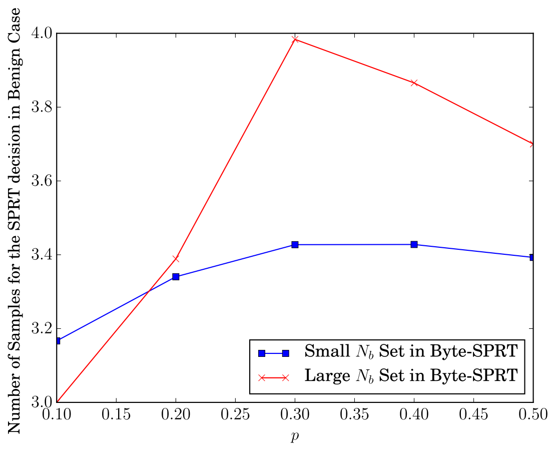


FIGURE 11. In Benign case, comparing the number of samples for the Byte-SPRT decision in small N_b set to the one in large N_b set while varying the byte selection probability p in probabilistic packet inspection procedure.

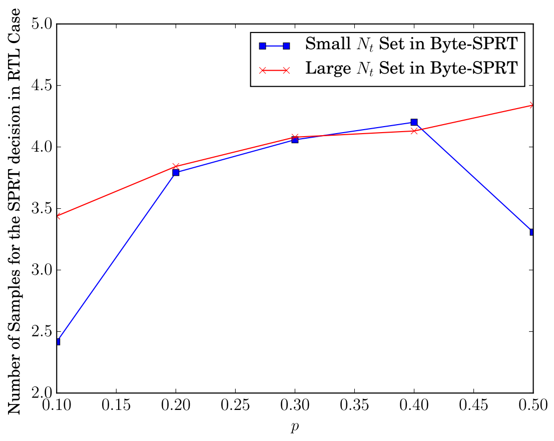


FIGURE 12. In RTL case, comparing the number of samples for the Byte-SPRT decision in small N_t set to the one in large N_t set while changing the byte selection probability p in probabilistic packet inspection procedure.

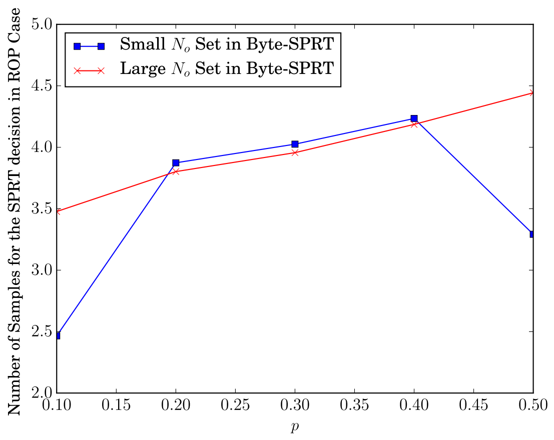


FIGURE 13. In ROP case, comparing the number of samples for the Byte-SPRT decision in small N_o set to the one in large N_o set while varying the byte selection probability p in probabilistic packet inspection procedure.

measured as at most 4.234 and as at most 5.992 in all settings of small N_o set, respectively. From these measurements, we notice that the Byte-SPRT with small N_o set accomplishes faster ROP attack detection than the Block-SPRT. Figure 13 shows the comparison results on number of samples between small and large N_o sets in the Byte-SPRT. We first recognize that the Byte-SPRT demands at most 4.443 samples on an average in all settings of p, small and large N_o sets. We also observe that number of samples in small N_o set is fewer than the one in large N_o set in cases of $p = 0.1, 0.5$. Similar to RTL case, we perceive from this observation that the lower value of $z_0 = 0.05$ in cases of $(N_o, p) = (50, 0.1), (10, 0.5)$ results in the fewer number of samples than the higher value of $z_0 = 0.1$ in other cases of (N_o, p).

As shown in Figure 7, ROP attack detection rate of the Byte-SPRT with small N_o set ranges from 93.3% to 99.5% while detection rate in the Block-SPRT ranges from 23.7% to 26.9%. This indicates that the lowest detection rate of our proposed Byte-SPRT with small N_o set is higher than 3.47 times as much the highest detection rate of the Block-SPRT as. Figure 15 displays the comparison results between the detection rates of small and large N_o sets in the ROP case of the Byte-SPRT. We see that the Byte-SPRT with large N_o

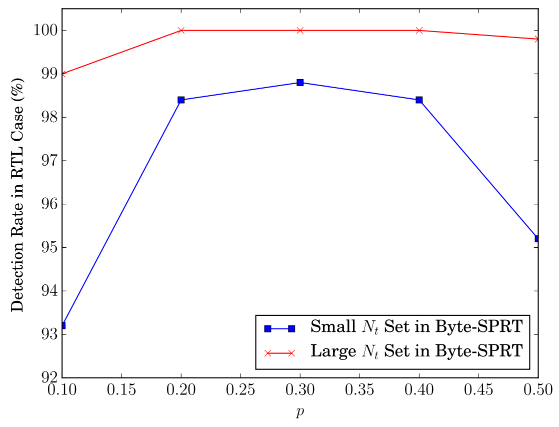


FIGURE 14. In RTL case, comparing detection rate (%) of the Byte-SPRT in small N_t set to the one in large N_t set while varying the byte selection probability p in probabilistic packet inspection procedure.

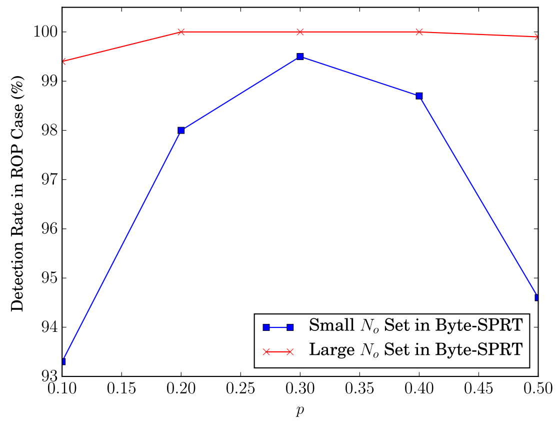


FIGURE 15. In ROP case, comparing detection rate (%) of the Byte-SPRT in small N_o set to the one in large N_o set while varying the byte selection probability p in probabilistic packet inspection procedure.

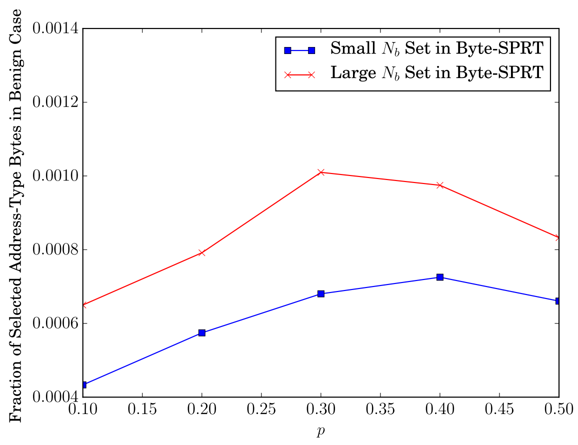


FIGURE 16. In Benign case, comparing fraction of selected address-type bytes of the Byte-SPRT in small N_b set to the one in large N_b set while varying the byte selection probability p in probabilistic packet inspection procedure.

set detects ROP attacks at least 99.4% rate. We also perceive that detection rates in large N_o set are higher than the ones in small N_o set. From this, we discern that the more number of address-type blocks leads to the higher ROP attack detection.

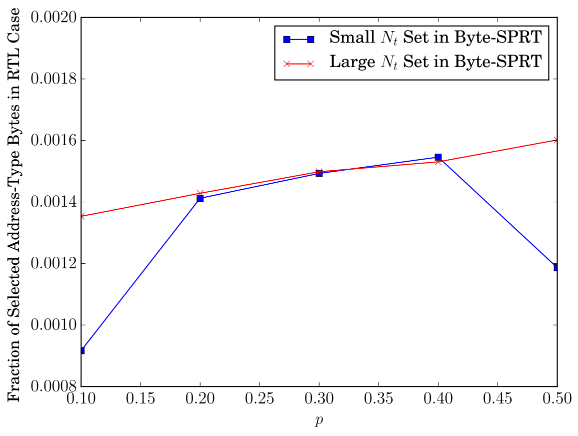


FIGURE 17. In RTL case, comparing fraction of selected address-type bytes of the Byte-SPRT in small N_t set to the one in large N_t set while varying the byte selection probability p in probabilistic packet inspection procedure.

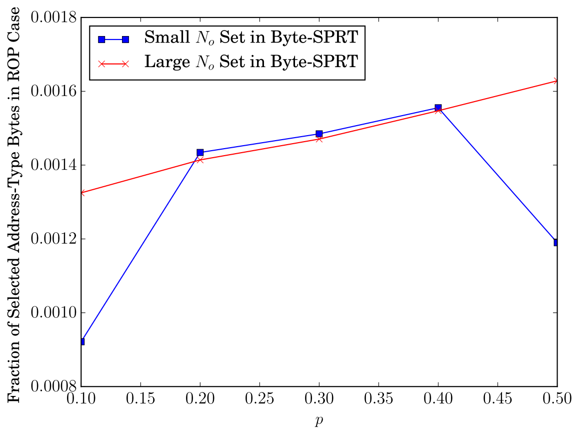


FIGURE 18. In ROP case, comparing fraction of selected address-type bytes of the Byte-SPRT in small N_o set to the one in large N_o set while varying the byte selection probability p in probabilistic packet inspection procedure.

Figures 8, 9, 10 display the comparison results of fraction of selected address-type bytes between the Block-SPRT and the Byte-SPRT with small N_b, N_t, N_o sets, respectively. Fraction of selected address-type bytes is refrained from below 0.001, 0.0078, and 0.0019 in Benign, RTL, and ROP cases of the Block-SPRT, respectively. Except the cases that $N_b = 1, 2$ in Figure 8, the fractions of selected address-type bytes in the Block-SPRT tend to be higher than the ones in the Byte-SPRT with small N_b, N_t, N_o sets. As shown in Figures 16, 17, 18, fraction of selected address-type bytes is restricted to below 0.001, 0.0016, and 0.0016 in Benign, RTL, and ROP cases of the Byte-SPRT, respectively. These imply that our proposed Byte-SPRT works with a small fraction of address-type bytes selected for the SPRT.

VI. CONCLUSION

In this paper, we devise a code-reuse attack detection scheme based on the probabilistic packet inspection and the SPRT in industrial IoT devices and demonstrate that our developed scheme attains resilient detection competence with little overhead through simulation and analysis. In particular, our simulation results exhibit that the SPRT with probabilistic packet inspection accomplishes at least 93.2% detection rate on an average for small set of code-reuse packets and at least 99.0% detection rate on an average for large set of code-reuse packets while requiring below five samples for detection on an average. Furthermore, it attains at most 0.4% average false positive rate with below four samples on an average.

REFERENCES

[1] T. Bletsch, X. Jiang, and V. Fresh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. ACSAC*, 2011, pp. 353–362.
[2] K. Braden *et al.*, "Leakage-resilient layout randomization for mobile devices," in *Proc. NDSS*, 2016, pp. 1–15.
[3] S. Crane *et al.*, "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. IEEE S&P*, May 2015, pp. 763–780.
[4] S. J. Crane *et al.*, "It's a TRaP: Table randomization and protection against function-reuse attacks," in *Proc. ACM CCS*, 2015, pp. 243–255.
[5] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. USENIX Secur.*, 2014, pp. 1–17.
[6] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proc. NDSS*, 2015, pp. 1–15.
[7] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. ASIACCS*, 2011, pp. 40–51.
[8] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Proc. NDSS*, 2017, pp. 1–15.
[9] J. Habibi, A. Panicker, A. Gupta, and E. Bertino, "DisARM: Mitigating buffer overflow attacks on embedded devices," in *Proc. Int. Conf. Netw. Syst. Secur.*, 2015, pp. 112–129.
[10] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proc. IEEE S&P*, May 2012, pp. 571–585.
[11] J.-W. Ho, "Code-reuse attack detection using Kullback-Leibler divergence in IoT," *Int. J. Adv. Smart Converg.*, vol. 5, no. 4, pp. 54–56, 2016.
[12] J. Ho, S. Kang, and S. Kim. (2017). *Poster: Code-Reuse Attack Detection Using Sequential Hypothesis Testing in IoT*. [Online]. Available: https://www.ieee-security.org/TC/EuroSP2017/posters/poster14.pdf
[13] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proc. USENIX Secur.*, 2014, pp. 417–432.
[14] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with Intel TSX," in *Proc. ACM CCS*, 2016, pp. 380–392.
[15] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proc. ACM CCS*, 2015, pp. 280–291.
[16] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make ASLR win the clone wars: Runtime re-randomization," in *Proc. NDSS*, 2016, pp. 1–15.
[17] G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? Revisiting assumptions of JIT-ROP defenses," in *Proc. USENIX Secur.*, 2016, pp. 1–19.
[18] *Raspberry Pi 3*. Accessed: Feb. 1, 2018. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[19] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.
[20] J. Seo *et al.*, "SGX-Shield: Enabling address space layout randomization for SGX programs," in *Proc. NDSS*, 2017, pp. 1–15.
[21] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE S&P*, May 2013, pp. 574–588.
[22] K. Z. Snow, R. Rogowski, F. Monrose, J. Werner, H. Koo, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *Proc. IEEE S&P*, May 2016, pp. 954–968.
[23] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proc. ACM CCS*, 2015, pp. 256–267.
[24] A. Wald, *Sequential Analysis*. New York, NY, USA: Dover, 2004.
[25] J. Werner *et al.*, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proc. CCS*, 2016, pp. 35–46.

JUN-WON HO received the B.S. degree from the Department of Computer Science, Yonsei University, Seoul, South Korea, the M.S. degree from the Department of Electrical and Computer Engineering, University of Wisconsin at Madison, and the Ph.D. degree from the Department of Computer Science, University of Texas at Arlington, in 2010. His Ph.D. dissertation was on the detection and prevention of wide-spread node compromise in wireless sensor networks. He is currently an Associate Professor with the Department of Information Security, Seoul Women's University. His current research interests include system and android security.