IoT-oriented high-efficient anti-malware hardware focusing on time series metadata extractable from inside a processor core

Kazuki Koike¹ · Ryotaro Kobayashi¹ · Masahiko Katoh²

Published online: 25 February 2022
© The Author(s) 2022

Abstract

We aim to improve the efficiency of our previously proposed anti-malware hardware; it is a hardware-implemented malware detection mechanism that uses information inside the processor. We previously evaluated a prototype, but, due to its prototypical nature, there remain limitations, such as only detecting certain behaviors, high power consumption, and a tendency to bloat the training model. In this paper, we propose a circuit and a learning method to achieve high efficiency, low power consumption, and light weight for the model. In considering these three issues, we focus on time-series metadata obtained by transforming the processor information. To improve efficiency, we implement predictive detection to predict the behavior of metadata in the malware detection component. This lets the model detect malware within less than 19% of the number of execution cycles of the conventional method. To reduce power consumption, we implement a sampling circuit that interrupts the input to the detection circuit at regular intervals, reducing the system's uptime by 99% while maintaining judgment accuracy. Finally, for a light weight, we focus on the training process of the metadata generator based on a machine-learning model. By applying sampling learning and feature dimensionality reduction in the training process, a metadata generator approximately 16% smaller than the previous version is created.

Keywords Malware detection · Internet of Things · Machine learning · Processor information · QEMU · Behavior-based method

1 Introduction

The recent spread of Internet of Things (IoT) technology has enabled various devices to be operated via networks. The majority of IoT device users are likely using multiple devices simultaneously and, by the end of 2020, the number of IoT devices connected to the Internet is estimated to be 25 billion [1,2]. However, many IoT devices remain vulnerable to security risks [3] and there has been an increasing number of attacks targeting them.

Mirai, a malware that takes over IoT devices and executes distributed denial-of-service (DDoS) attacks on other servers through command-and-control servers, emerged in 2016 [4]. One prominent target of Mirai was the Domain Name System server “Dyn” [4]. With the release of Mirai's source code, many malware variants have been created, including some that destroy device data [5]. In this paper, we describe Anti-malware software covers a range of software-based security measures. For example, the signature method detects malware-specific code and hash values by pre-listing them, calculating the hash values from target files, and comparing them to the list. Behavioral methods focus on program behavior and dynamically determine malicious behavior by obtaining statistics on the type and order of application program interface (API) calls made by a program and comparing them to pre-listed malware statistics [6–8]. Heuristic methods apply machine learning and data mining to detect malware by a discriminator that dynamically collects behaviors and uses them as features [9]. However, these methods are difficult to implement in resource-constrained IoT devices because they

✉ Kazuki Koike
em19005@g.kogakuin.jp
Ryotaro Kobayashi
ryo.kobayashi@cc.kogakuin.ac.jp
Masahiko Katoh
masa@sun.ac.jp

¹ Kogakuin University, 1-24-2, Nishi-shinjuku, Shinjuku-ku, Tokyo 163-8677, Japan

² University of Nagasaki, 1-1-1, Manabino, Nagaya-cho, Nishisonogi-gun, Nagasaki 851-2195, Japan

are software-based and run in the background, constantly using memory and power.

Therefore, we proposed a malware detection mechanism for IoT devices, implemented on the large-scale integration (LSI) as dedicated hardware (hereafter called anti-malware hardware [10]). Unlike software-implemented mechanisms, the hardware can operate independently of the core to reduce the core resource consumption associated with malware detection to zero. Anti-Malware Hardware detects malware by using information directly obtained from the core on the LSI as electrical signals (hereafter called processor information) as features.

However, our previously proposed detection mechanism has four problems: (1) it cannot correctly detect a program that behaves differently than expected, (2) it can only detect malware after executing at least a certain number of cycles, (3) it has high power consumption because it runs every cycle, and (4) the hardware implementation size is likely to be huge.

In this paper, we implement and evaluate proposal methods to address these issues. We consider meta-information (metadata) that can be extracted from processor information. These metadata are data that label processor information as benign or malignant using a machine-learning (ML) model and transform the information into a form that can be interpreted over time as the behavior of the application. Significant differences between normal programs and malware emerge in metadata. We take advantage of this by applying sampling circuits and predictive detection to capture behaviors that are not identified by conventional methods. Predictive detection has the effect of speculating on metadata features and limiting the number of execution cycles required for malware detection. The sampling circuit reduces power consumption by efficiently thinning, or sampling, the processor information required for judgment. This sampling can also be leveraged in the learning process to reduce the size of the ML model implementation.

The rest of this paper is composed as follows. Section 2 reviews related works. Section 3 shows differences from previously published papers. Section 4 presents the concepts and problems of the proposed mechanism. Section 5 describes the proposed methods. Section 6 provides the evaluation method and its results, which are discussed in Sect. 8. Section 7 evaluates the hardware performance. Section 9 concludes the paper.

2 Related works

Recent CPUs are equipped with a mechanism called hardware performance counter (HPC): Intel Performance Counter Monitor (PCM), Intel Processor Trace (Intel PT), ARM Streamline, etc. The HPC monitors performance events inside processors. For example, Intel PCM counts instruc-

tions retired, cache hits and misses, and others. The information can be used to help performance analysis or tuning of application software. Many researchers have focused on the processor level information and proposed cyber security measures utilizing HPC [11]. Among them, the following researches use machine learning and HPC to detect attack behavior.

Torres et al. proposed a software method to detect data-oriented attacks using HPC [12]. A data-oriented attack is an attack that causes unintended behavior by rewriting the data in the memory space. Their proposed method generated a machine learning model using the data obtained from the HPC under normal conditions and under abnormal conditions when the data-oriented attack is being executed. They evaluated two-class classification using the SVM model and showed 92% detection accuracy.

Bahador et al. proposed a malware detection method that combines HPC, singular value decomposition, and machine learning [13]. This detection method, called HPCMalHunter, shares similarities with our method in that it uses processor information such as cache hit rate and uses machine learning to detect malware, but differs from our hardware-based mechanism in that it is implemented solely in software. They evaluated HPCMalHunter with some malware and benign programs. The evaluation results showed that the correct answer rate was 90.69% and the false positive rate was 0.79%, resulting in a high detection rate and a low false positive rate. However, it means that there still exist at least some undetected malware and falsely detected benign programs.

Junaid et al. proposed a software method to avoid side-channel attacks by using information obtained from HPC [14]. There are some situations where a side-channel attack can be established. They focused on the situation in which a side-channel attack occurs when two different applications share the same functional unit in a processor. In the proposed method, a neural network detects and predicts the execution phase of each application based on the hardware events which are obtained from the HPC by the OS scheduler. Based on the detected and predicted phases, the scheduler allocates applications so that they do not share the same functional unit at the same time. The software method proposed by Manaar et al. also uses hardware information to detect side-channel attacks [15]. This work is similar to that of Junaid et al. in that it focuses on side-channel attacks and uses performance counters, but differs in that it focuses on the correlation between the process of the attack program and the process of the Clefia encryption program.

Vinayaka et al. proposed a software-based framework called BRAIN, which adds HPC information as features to conventional DDoS detection models that use network and application statistics [16]. The conventional model that uses network and application statistics as input to the SVM showed a correct answer rate of 99.32%. In contrast, the

BRAIN framework succeeded in improving the correct answer rate to 99.8% while keeping the false positive rate at 0%.

Liu et al. proposed a security measure that does not use HPC, but Intel Processor Trace (IPT) to enhance the integrity of the application control flow [17]. IPT is a software-based function that collects processor execution traces and provides millions of lines of instruction history to the OS with low overhead.

The above studies used processor information and machine learning algorithms as well as our works. However, their cyber security measures are realized in software, in contrast to our AMH that is a hardware mechanism on the LSI. Therefore, these software-based methods are difficult to introduce in resource-constrained IoT devices. In addition, they used dedicated tools or instructions to extract processor information and directly used the output from the identifier generated by machine learning for the detection, while our AMH employs wired-logics to directly capture processor information and treats it as time-series data as mentioned in Sect. 4.4. As a result, our method accelerates malware detection speed while maintaining high accuracy.

By offloading certain processes to dedicated hardware, and thereby freeing central processing unit (CPU) resources for other applications, studies have reported reduced load on a CPU and increased processing speed. Ding et al. implemented a hardware-based transmission control protocol (TCP) offload engine that addresses the problem of traditional protocol stacks implemented in software [18].

ARM, a company heavily involved with processor core intellectual property (IP) for IoT devices, proposed edge device artificial intelligence (AI). One of its products, Ethos-U55, is a neural network processing unit (NPU) that specializes in machine-learning-based reasoning algorithms [19]. Combined with ARM's processor core, Cortex-M55, the processing performance of the inference algorithm was up to 480 times compared to Cortex-M33. As it requires less power than a general-purpose processor to perform similar processing, it has immense benefits for microcomputers and IoT devices.

Additionally, manufacturers have implemented security measures that are achieved through a combination of hardware and software. For example, Intel implements a feature called Intel Threat Detection Technology (Intel TDT) into some of its CPUs [20,21]. Intel TDT sends a hardware-based signal that indicates the run-time behavior at the CPU level. The signal can be used to help software threat detection agents that identify polymorphic malware, fileless scripts, cryptomining, ransomware, and other targeted attacks.

In November 2020, Microsoft announced Pluton, which they developed in collaboration with AMD, Intel, and Qualcomm Technologies, Inc [22]. Pluton is implemented as part of the LSI and can protect authentication informa-

tion, encryption keys, and personal data. The advantages of implementing security hardware on the LSI include communication between the core and the security hardware not needing to go through a bus and prevention of physical attacks on the security hardware, even if an attacker were to steal the computer.

However, these examples and the previous studies of security hardware are focused on passive security features, in contrast to our proposed method with active security features for malware detection.

Although software-based security measures are widely used in general-purpose computers, they are relatively difficult to implement in IoT devices with limited hardware resources. Nevertheless, the process of embedding certain functions, including security functions, in hardware has already begun to be implemented.

3 Relationship with previous studies

Takase et al. previously proposed the hardware-based malware-detection mechanism for IoT devices using processor information as a feature [10]. In the study, they investigated the effectiveness of the mechanism and found that processor information obtained from the core can be effectively applied for malware detection using machine learning on ARM architecture. However, the mechanism had some problems related to operation efficiency and speed. The above mechanism and problems are explained in Sect. 4.

In November 2019, we presented a paper at the CANDAR Workshop that is an interim report of this paper [23]. The contributions of the interim report include the improvement of the operation efficiency and speed of Takase's mechanism [10]. The former is achieved by sampling the processor information, which is a feature value. The latter is achieved by implementing a predictive detection algorithm that predicts the behavior of malware running on the CPU. These proposed methods are explained in Sect. 5. However, as the CANDAR paper is the interim report, it has the problem of insufficiencies in the description, analysis, implementation, and evaluation of the proposal.

This paper is the complete version of the interim report presented at CANDAR. All sentences and all figures in this paper are different from those in the previous work. The differences are divided into update and addition elements. For example, we updated the description of the hardware-implemented malware detection mechanism that was also presented in the interim report to explain the mechanism in more detail for improved understanding. In addition to update elements, we newly introduced addition ones. For example, we added the time-series metadata theory, which was not noted in the previous studies, and the analysis of the malware detection behavior based on the new theory. The major

differences from the CANDAR paper in this paper are shown below.

– Theory

Addition: The theory of time-series metadata is newly introduced to analyze the malware detection behavior of the conventional mechanism in each cycle (Sect. 4.4).

– Analysis

Addition: Qualitative discussion on power reduction (Sect. 5.2).

– Explanation

Update: A description of the malware detection circuitry and its operation, with very detailed figures and text (Sect. 5).

– Evaluation

Update: By adding preliminary evaluations to each evaluation process, we conclude performance improvements based on 100% Accuracy and 0% False Positive (Sect. 6).

Addition: We have redesigned the proposed mechanism in Verilog language, and performed the logic synthesis and place-and-route targeting FPGA (Sect. 7).

Addition: Based on the placement and routing results, we performed a quantitative evaluation of the hardware resource consumption (Sect. 7.2).

Addition: Based on the placement and routing results, we performed a quantitative evaluation of the power consumption (Sect. 7.3).

– Verification

Addition: Significant additions to the discussion based on the newly introduced theory of time-series metadata and new evaluation results (Sect. 8).

While the other sections, which are not noted in the above, are minor differences, they are helpful for understanding our study.

4 Hardware mechanism and problems

We hereafter refer to the hardware-implemented malware detection mechanism as anti-malware hardware (AMH). In this section, we outline the functions of AMH, how to implement it, and key issues.

Fig. 1 Comparison of methods for implementing anti-malware functions

4.1 Introduction of anti-malware hardware

The functions comprising the AMH are implemented in hardware on the LSI. Figure 1 compares the traditional software-based implementation and our proposed hardware-based implementation.

In Fig. 1a, the anti-malware functionality is implemented in software, wherein the user application and the anti-malware software are executed in parallel on the core on the LSI. In many cases, anti-malware software runs in the background, competing with other applications for core resources.

By contrast, in Fig. 1b, the anti-malware function is implemented on the LSI as AMH and operates independently of the core. The hardware information inside the core executing the user application is sent unidirectionally as features to the AMH. As the AMH only copies the internal state of the core running the application, it does not consume any core resources, thus preserving these resources for user applications.

4.2 Hardware mechanism and behavior

This section details the structure of the proposed mechanism. Figure 2 is a magnified and more detailed version of Fig. 1b.

The yellow elements represent logical areas, such as user applications, software, and the operating system (OS). The blue and orange elements, respectively, represent areas of the core and AMH, which are implemented on the LSI as independent circuits that can operate in parallel.

We describe the core in a general-purpose computer only as it relates to this study, such as the program counter, register file, and branch prediction mechanism. First, the core fetches an instruction to be executed by referring to the memory address in the program counter. At this time, if the instruction exists in the cache, it is fetched from the cache. When the fetched instruction is decoded, register fetching is performed, and the data necessary for instruction execution is loaded from memory into the register file. As well as instruction fetching, if there is data in the cache, the data is also

Fig. 2 A schematic of the proposed malware-detection mechanism

fetched from the cache. Thus, the instruction is executed for the first time when the value of the register file is complete. The execution of a branch instruction may cause the next instruction to branch, but the branch prediction mechanism intervenes to rewrite the speculative program counter.

The AMH connected to the core consists of three elements:

- Bypass and Feature-adding circuits,
- Metadata generator, and

- Malware detector.

A bypass circuit is wiring implemented on the LSI that outputs the status of various registers, caches, memory cells, etc., inside the core every cycle. The bypass circuit directly connects each element on the core, including the program counter, to the machine learning circuit. The program counter is a register that holds instruction addresses and is updated with the address of the next instruction each time the instruction in question is executed. The feature addition circuit consists of a hit counter composed of transistors, wiring between the cache and the hit counter, and wiring between the hit counter and the metadata generator. The hit counter converts cache hit/miss information into a cache hit rate. Thus, processor information can also be described as streaming data because it outputs the state of the core as it changes from one cycle to the next. However, it is different from the information obtained by static analysis of binaries with software. The processor information obtained through the bypass circuit or feature-adding circuit is sent to the metadata generator as a feature that identifies the software being run.

The metadata generator generates metadata necessary for judging software by using a built-in ML model. The metadata used to determine malware is a time series of attack and normal labels assigned to the processor information. It is implemented by a transistor on the LSI and, on each cycle, it labels the processor information as normal or attack. The algorithm is based on Random Forest, which has the advantage of not requiring normalization. It can be trained efficiently on processor information that contains both numerical and categorical values. Thus, the essence of the metadata generator is an ML model, but the classification result per cycle output from the model does not immediately affect the software decision result. Like other elements, this ML model is implemented on LSI in the end, but the whole process of implementation of the ML model is as follows: the model is generated by machine learning software, and converted to circuit information by circuit design tools, and then the circuit information is implemented as hardware circuits on LSI.

The metadata output from the metadata generator can be treated as time-series data representing the characteristics of the software. The malware detection circuit makes a decision based on the percentage of attack labels contained in the metadata. First, the decision circuit receives the metadata from the metadata generator and counts the number of attack labels contained in the metadata. If the percentage of attack labels exceeds the threshold value at the end of the software execution, then the decision circuit judges the software to be malware. This approach of conversion and interpretation into metadata allows more accurate determination than directly considering the processor information.

Table 1 Detail of the processor information

Data	Information	Detail
Raw	Instruction address	Address of the next instruction to be executed, stored in the program counter
	Instruction type	The value of four types of instructions: Load, Store, Jump, and No Operation (NOP)
	Operation code	Instruction identification number as defined in the Instruction Set Architecture
	Load/Store address or target PC	Data input/output destination address in the Load/Store instruction or branching address
	Condition flag	Flags set by comparison operations (e.g. Negative, Zero, Carry, Overflow)
	Register number	Register number as specified in the operand
	Register value	The value stored in the register specified above
Count	Hit rate of L1 instruction cache	The probability that the desired instruction existed in the L1 instruction cache
	Hit rate of L1 data cache	The probability that the desired data existed in the L1 data cache
	Hit rate of L2 cache	The probability that the desired instruction or data existed in the L2 cache
	Hit or miss by BTB	Prediction hit or miss by the Branch Target Buffer (BTB) mechanism
	Branch direction	The direction in which the instruction actually branched
	Predicted branch direction by Gshare	The branching direction output by the Gshare method's branching prediction mechanism
	Distance of Load	The number of instructions that have elapsed since the Load instruction was last executed
	Distance of Store	The number of instructions that have elapsed since the Store instruction was last executed
	Distance of Jump	The number of instructions that have elapsed since the Jump instruction was last executed
	Distance of NOP	The number of instructions that have elapsed since the NOP instruction was last executed
	Distance of Other Operation	The number of instructions that have elapsed since the last instruction other than NOP, Load, Store, or Jump was executed

4.3 Summary of the features

In this study, processor information is defined as the streaming data output every cycle from the core running the software. Table 1 lists the processor information used in this study.

The processor information is roughly divided into raw data in the first half and count data in the second half. Raw data is transmitted via direct wiring between the core and the metadata generator on the LSI. Among this raw data, the operation code and register number have a unique value and order for each software. In contrast, the instruction address and load/store address change each time they are executed, owing to address space layout randomization (ASLR).

Count data is sent via a feature-adding circuit, such as the hit counter. The hit ratio of the L1 and L2 caches is a fea-

ture computed by the hit counter. They are incorporated into the processor information to capture the spatial and temporal locality of the program. The items from hit or miss by BTB to predicted branch direction by Gshare are features related to various branching processes. Items from the distance of NOP to distance of other operation are called instruction distances, and refer to the number of instructions that have elapsed since the last instruction of the same type was executed. Although omitted in Fig. 2, various data related to branching and instruction distance are outputted by a dedicated circuit as well as the hit counter. Thus, processor information bypassed directly from the core or added by feature-adding circuits is inputted into the metadata generator as software features.

Figure 3 shows how processor information is outputted from the core each cycle.

Cycle	Rate_hit_L1-I	Rate_hit_L1-D	Addr_inst	Dist_nop	Rate_hit_L2	Other_feats
0	0.9432	0.8076	0x00509F18	-1	0.364499	
1	0.9432	0.8076	0x00509F1C	-1	0.364499	
2	0.9432	0.8076	0x00509F20	-1	0.364589	
3	0.9433	0.8076	0x00509F24	-1	0.364589	
4	0.9433	0.8076	0x00509F28	0	0.364589	
5	0.9433	0.8076	0x00509F2C	1	0.364589	
6	0.9433	0.8076	0x00509F30	2	0.364589	...
7	0.9433	0.8078	0x00509F34	3	0.364589	
8	0.9433	0.8078	0x00509F38	4	0.364589	
9	0.9433	0.8078	0xB6935A2C	5	0.364589	
10	0.9434	0.8078	0xB6935A30	0	0.364589	
11	0.9434	0.8078	0xB6935A34	1	0.364589	
12	0.9434	0.8078	0xB6935A38	2	0.364589	
⋮						

Fig. 3 An example of the processor information (streaming data)

This processor information is sorted based on the rate of contribution of the features described below. The values in the Cycle column are indices that start at 0 and increase by 1 for each cycle. The values in the Hit_L1-I, Hit_L1-D, and Hit_L2 columns increase or decrease by a number in the range of $0 \leq n \leq 1$ for each cache hit/miss output. The value of the instruction address column points to the address in memory where the instruction is stored, usually increasing by 4. However, depending on the instructions to be executed, the memory space may change during the execution, and the value of the instruction address may change significantly. The Distance_nop column records the number of cycles that have elapsed since the last NOP instruction was executed, and until the first NOP instruction is executed, this is recorded as -1.

4.4 Summary of metadata

The metadata is time-series data that is labeled for each cycle by inputting processor information into the metadata generator. Figure 4 shows the metadata of the software used in this study.

The horizontal line is the normalized number of cycles that have been executed on the core (hereafter referred to as execution cycles). This normalization makes it easier to compare metadata between software with different execution cycles. The vertical line is the number of instructions labeled as an attack by the metadata generator. The metadata is a set of binary (attack or normal) data, and we plot the number of attack labels per 250 cycles for illustrative purposes.

As a measure of metadata, we refer to the percentage of attack labels in a given interval as the score. Half of the specimens in Fig. 4a have continuous attack labels throughout and their metadata is linear at the top. In this case, the metadata score is 1 or close to 1. Some specimens, such as Trojan_1 and Kaiten_2, were found to have partially decreasing attack labels and nonlinear graphs. They had lower scores than the specimen with a continuous attack label. We investigated the phenomenon of normal labels in malware metadata and

(a) Malware

(b) Normal program

Fig. 4 Metadata of the software used in the experiment. The vertical axis is the number of attack labels included in every 250 cycles. The horizontal axis is the normalized execution cycles

found that it is caused by the execution of a shared library. Since library calls cannot be regarded as offensive behavior, it is necessary for the malware detection circuit to be able to handle irregularities in metadata.

In the software metadata shown in Fig. 4b (hereafter referred to as the normal program), the metadata is linear at the bottom. In this case, the metadata score is 0 or close to 0. However, we have identified nonlinear metadata containing attack labels such as the $\times 264$ command, which may be because of the same reason as the above issues with Trojan_1 and Kaiten_2.

4.5 Problem of the proposed mechanism

In this chapter, we describe some problems with AMH. The first is the metadata determination method. In conventional methods, applications are determined by comparing the overall metadata score with a threshold value. However, as shown in Fig. 4, there is a risk of missing or misdetecting software that contains many labels with opposite attributes in the conventional method. The score of the fourth interval decreases

to approximately 100, and the $\times 264$ command shown in (b) exhibits an increase in the number of attack labels in the middle of the graph. As this trend becomes more pronounced, simply looking at the overall metadata score can lead to missed instances or false positives of malware.

The second is detection speed. Traditional methods compare the threshold with the overall metadata score at the point when the software in question has finished. If malware is executed, this cannot be determined until a certain cycle core executes the process, and even if it is detected, the damage has already spread. Thus, we need a decision process that can detect malware with diverse metadata within fewer cycles.

The third is the availability of AMH. As mentioned in Sect. 4.2, the AMH operates in synchronization with the core and the timing of its operation. When judging software with high core utilization, the input and output per unit of time for bypass and processing circuits increases and, with it, the utilization of metadata generators becomes very high. In addition, there is a proportional relationship between availability and power consumption in hardware components, and naturally the same is true for AMH. We cannot ignore the issue of power consumption if the proposed mechanism is installed in an IoT device. Therefore, we study a method to reduce the utilization rate of the metadata generator by suppressing unnecessary operations while maintaining judgment accuracy.

The fourth is the size of the ML model. In conventional methods, when generating an ML model with a metadata generator, the processor information obtained during the execution of the application to be learned is learned for all cycles. The problem is that the model becomes huge when an application with high core utilization is trained. If the model is implemented as hardware, then the smaller the size of the model, the more desirable it is, as long as labeling accuracy is maintained. Thus, we investigate a method to reduce the total amount of training data to make the model lighter.

5 Proposal method

To address the problems described in Sect. 4.5, we propose the following approaches:

- Improving the efficiency of the detection process with predictive detection,
- Reduce hardware uptime through sampling, and
- Ensuring lightweight ML models through sampling and dimension reduction.

5.1 Efficiency with predictive detection

Predictive detection is a method of speculative malware detection that predicts the propensity of attack labels to

Fig. 5 Predictive detection algorithm

appear in metadata. It is implemented as a modification of the malware detector algorithm. Speculative detection reduces the malware execution cycle and accelerates detection speed. In addition, the metadata generated from the malware has some common features, as described below, which allows efficient detection of malware with a relatively high number of normal labels.

Figure 5 shows the algorithm for predictive detection.

We define W as the cycle width (called the window) to be processed at one time, S_w as the percentage of attack labels in W cycles, and S_t as the percentage of attack labels needed for detection (i.e., the threshold).

1. Label the processor information output from the bypass and processing circuits from the top W cycles and generate the metadata.
2. The malware detector calculates S_w from the metadata generated in step (1).
3. If $S_w \geq S_t$, then detect the running application as malware and exit the process.
4. Otherwise, the next output processor information is labeled in one cycle.
5. Recalculate S_w with the metadata of the latest W cycles.
6. Repeat steps (3) to (5).

Malware has a characteristic concentration of attack labels on some or all of the metadata. The predictive detection algorithm calculates the score each time while moving the window one cycle at a time, allowing for malware detection to focus on these types of concentrations.

Fig. 6 Sampling behavior

5.2 Reducing utilization rates with sampling circuit

We propose a method to reduce the operating rate of the metadata generator by implementing a sampling circuit and thinning the processor information that is outputted every cycle. The sampling circuit plays the role of a gate that repeatedly opens and closes at arbitrary intervals, and is placed just before the hit counter and the metadata generator, through which the output from the core must pass. Figure 6 shows the operation of the sampling circuit.

The operation of the sampling circuit follows the number of cycles x to keep the gate open and the number of cycles (or intervals) y to keep it closed. For example, if $x = 2$ and $y = 3$, then the sampling circuit passes processor information for two cycles and is closed for three cycles. If the utilization rate of the metadata generator before the implementation of the sampling circuit is 1, then the utilization rate R_u after the implementation is expressed by Eq. 1:

$$R_u = \frac{x}{x + y} \quad (1)$$

If $x = 2$, $y = 3$, then the operating rate of the metadata generator is 0.4 (= 40%). The reduction ratio R_r is expressed by:

$$R_r = 1 - R_u. \quad (2)$$

We now describe the power consumption of the metadata generator. For the circuit implemented on the LSI, when the switching probability is α , the capacitance of the circuit is C , the supply voltage of the circuit is V_{DD} , and the operating frequency is f , the energy consumption of the circuit is expressed by the following Eq. [24]:

$$E = \int_0^T (\alpha C V_{DD}^2 f) dt. \quad (3)$$

Assuming that C , V_{DD} , and f are time-independent constants, Eq. 3 can be rewritten as:

$$E = \alpha C V_{DD}^2 f T. \quad (4)$$

Here, power $P_{switching}$ is defined using frequency f , we obtain:

$$P_{switching} = \frac{E}{T} = \frac{\alpha C V_{DD}^2 f T}{T} = \alpha C V_{DD}^2 f. \quad (5)$$

Equation 5 shows that the power consumption of the circuit on the LSI is proportional to the switching probability. The utilization rate R_u can be regarded as the switching probability α in Eq. 5. For $R_u = 0.4$, the power consumption of the metadata generator can theoretically be reduced to 40%.

5.3 Lightweight metadata generator

We propose two methods to achieve lightweight metadata generators:

- Sampling learning, and
- Dimension reduction of features.

Sampling learning involves applying the sampling concept to the process of generating ML models, which accounts for most of the metadata generator. In conventional methods, the processor information that has been outputted since the start of software execution is directly inputted to the ML model for training. Sampling learning reduces the amount of processor information inputted to the model through sampling. Thus, the size of the model on the LSI is reduced compared to conventional methods. To distinguish this method from sampling in order to reduce utilization, we call it sampling learning.

Dimension reduction of features aims to reduce the weight of the model by reducing unnecessary features based on the contribution rate of the features obtained during model creation. In the conventional method, we use the seventeen types of 21-dimensional features shown in Table 1. Some dimensions or features have little or no influence, and the implementation size tends to grow concerning the performance of the generated model. Therefore, we reduce the model weight by calculating the contribution rate of the features during model generation and disabling unnecessary features. The procedure of calculating the contribution ratio and generating the model is assumed to be done on the emulator, and at the same time, a mechanism for disabling features is required on the wiring, as shown in Fig. 2. This hardware mechanism can be easily implemented by closing only the gates of the relevant features in the sampling circuit implemented in Sect. 5.2.

Table 2 List of malware specimens

Name	VirusTotal results md5	# of detections	Symantec analysis result	# of cycles	
				Training	Test
Kaiten_1	133b29184ce7a338e964d39996017c70	18/57	Linux.Kaiten	10,103	5234
Kaiten_2	9439f8da5c24ef20bdc60eff0e82ee76	36/55	Linux.Backdoor.Kaiten	18,360	26,425
Mirai_1	c8562078c83d02334956b82bc5b1dd4c	29/58	Linux.Mirai	7,418	10,592
Mirai_2	9df0fd7abb2c3f66c4c2a16ecf18b632	31/60	Linux.Mirai	11,333	17,290
Trojan_1	b5af5069310f48ae7b4200f9d81d5d57	34/59	Trojan.Gen.2	32,241	32,016
Trojan_2	d0aa40ca0878f26555d958a68232e986	32/60	Trojan.Gen.MBT	24,884	20,496

6 Evaluation

This section describes the common evaluation environment and the evaluation details for the proposed method.

6.1 Anti-malware hardware emulator

Because this study proposes and evaluates hardware that implements anti-malware functions, we used a hardware emulator for evaluation. In the LSI and processor study, when evaluating a new architecture or system, it is important to use a highly accurate and flexible simulator or emulator instead of actually creating the LSI [25,26]. SimpleScalar [27], a type of processor simulator, is used to verify various processor systems and architectures and is widely applied in processor research, such as comparing cache models and evaluating the development of branch prediction mechanisms [28,29]. Taking advantage of its open source nature, some researchers have made their own improvements to SimpleScalar and used it for evaluation [30].

6.2 Evaluation environment

To evaluate the proposed method, we created an emulator, which can be divided into the core part and the AMH part, as shown in Fig. 2. For the core part, we reproduced the branch prediction mechanism, program counter, register file, and various caches in C language. The core emulator is based on the open source QEMU (ver. 2.4.1) [31]. For the AMH part, we reproduced the hit counter, metadata generation circuit, and malware detector in Python. The emulator for the metadata generation circuit is Random Forest from the scikit-learn library. In addition to the AMH components shown in Fig. 2, we also created an emulator in Python to reproduce the proposed sampling circuit and predictive detection.

We booted the Raspbian OS on the core emulator and executed the malware and normal programs shown in Tables 2 and 3, respectively.

The VirusTotal columns in Table 2 show the information obtained by analyzing the samples with VirusTotal [32]. The # of detections column indicates the number of engines that

Table 3 List of normal programs

Name	Option	# of cycles	
		Training	Test
apache2	-k start	95,000	95,000
df	-h	26,432	26,083
gcc	N/A	95,000	95,000
ls	-al	21,241	22,562
tar	-xf	29,935	33,018
× 264	-vf resize:720,480	95,000	91,684

detected the uploaded file as malware. The Symantec analysis result column is the name registered in the Symantec detection engine, and the specimens were named on the basis of this result. The # of cycles columns are the training and testing execution cycles of the malware. All specimens in the table were captured by our Cowrie [33] honeypot. We selected specimens captured between July 31, 2018 and August 23, 2018 and ran them in an ARM/Linux environment. All the specimens were unpacked ELF files.

Table 3 shows the list of normal programs. We selected six commands expected to be used in IoT devices. The option column shows the options specified at the time of command execution, except for those that are mandatory. We emulated up to 100,000 cycles of processor information and disabled the first 5000 cycles, including the loading process to memory, in the evaluation experiment.

6.3 Evaluation of predictive detection

To evaluate the efficiency of predictive detection, we performed a preliminary evaluation where we examined a range of parameters that would allow successful predictive detection (i.e., accuracy of 100%). We define accuracy in AMH as the percentage of correct decisions made by the evaluation software. The initial candidates for W and S_t defined in Sect. 5.1 are

Fig. 7 Change in accuracy in predictive detection

$$W = \{1, 10, 100, 1000\}$$

$$S_t = \{0.1, 0.2, 0.3, \dots, 0.9, 1.0\}.$$

The larger the W , the lower the sensitivity to malignant labels, and the smaller the W , the higher the sensitivity. Combining these parameters, we emulated 40 different predictive detections for 12 different evaluation software.

Figure 7 shows W , S_t , and the change in accuracy.

The horizontal axis is S_t , the vertical axis is the accuracy, and the graph is summarized for each value of W . For example, if $W = 100$ and $S_t = 0.6$, the malware is judged to be malware when a section with an attack label is found for more than 60 cycles out of 100 cycles. The arrows indicate the point where the accuracy reaches a ceiling such that even if S_t increases, the accuracy remains flat. Of the 40 combinations, the following 8 combinations achieve 100% accuracy:

- $W = 100, S_t = 1.0$
- $W = 1000, 0.4 \leq S_t \leq 1.0$

In the main evaluation, we focused on ($W = 100, S_t = 1.0$), ($W = 1000, S_t = 0.4$), and ($W = 1000, S_t = 1.0$).

In the main evaluation, we compared the number of cycles of the executed malware. It is known that normal programs do not respond to predictive detection within the range of parameters described above; therefore, we omitted the main evaluation of normal programs. Figure 8 shows the number of execution cycles of malware specimens.

For the most sensitive parameters ($W = 100, S_t = 1.0$), all malware specimens were successfully detected with a sufficiently small number of cycles. Even for the least sensitive parameter ($W = 1000, S_t = 1.0$), the number of cycles ranged from 4% to 19% of the reference cycles (see the # of cycles (Test) column in Table 2). These results indicate that the use of predictive detection to speed up the decision-making process can reduce the number of cycles required for

Fig. 8 Result of predictive detection for malware. The reference cycle depends on each malware samples

detection to 19% or less of the number required by conventional methods.

6.4 Evaluation of sampling circuit

We used a sampling circuit to evaluate the accuracy and the reduction rate of the utilization rate reduction method. First, we show the method and results of the accuracy evaluation. We set the following candidate parameters for evaluation:

$$x = \{1, 10, 100\}$$

$$y = \{100, 500, 1000, 5000\}$$

$$\text{Threshold} = \{0.1, 0.3, 0.5, 0.7, 0.9, 1.0\}$$

We emulated the sampling circuit according to x and y and generated metadata for 12 different software. The number of metadata increases with the number of parameters, so the total number of metadata was 144. Each of the 144 metadata has a unique score, and the percentage of software that can be correctly determined, i.e., the accuracy, depends on the value of Threshold.

Figure 9 shows x , y , Threshold, and the change in accuracy.

The bars, grouped by Threshold, represent accuracy. The Threshold from 0.1 to 0.7 shown in (a) achieves 100% accuracy regardless of the values of x and y . This means that all 12 types of evaluation software can be classified correctly. However, in (b), one software fails to be classified when $x = 1, y = 5000$. Furthermore, regardless of the values of x and y in (c), more than one software fails to be classified. Since the judgments start to fail after Threshold = 0.9, $x = 1, y = 5000$, we limited the range of parameters for evaluating the reduction rate to $x \geq 10$ and $y \leq 1000$.

Next, Table 4 shows the metadata scores of malware specimens.

The values in the table are the average, minimum, and maximum scores of the six malware samples. For example,

Fig. 9 Parameters and Accuracy in Sampling Circuits

Table 4 Metadata scores of malware via sampling circuit

y	x = 10			x = 100		
	Avg	Min	Max	Avg	Min	Max
00	0.985	0.963	1.000	0.986	0.959	1.000
500	0.983	0.955	1.000	0.987	0.958	1.000
1000	0.986	0.944	1.000	0.989	0.954	1.000

The reference scores are as follows: Average = 0.986, Min = 0.958, Max = 1.000. Bold indicates a score that has **decreased** by more than 0.01 from the reference score

Table 5 Metadata scores of normal program via sampling circuit

y	x = 10			x = 100		
	Avg	Min	Max	Avg	Min	Max
100	0.014	0.000	0.080	0.014	0.000	0.082
500	0.014	0.000	0.081	0.014	0.000	0.083
1000	0.012	0.000	0.070	0.015	0.000	0.091

The reference scores are as follows: Average = 0.014, Min = 0.000, Max = 0.080. Bold indicates a score that has **increased** by more than 0.01 from the reference score

for $x = 10$, $y = 500$, the average, minimum, and maximum scores are 0.983, 0.955, and 1.000, respectively. The higher the score, the larger the percentage of attack labels in the metadata. The minimum score at $x = 10$, $y = 1000$ is relatively reduced from the reference but is still sufficient to correctly identify the malware.

Finally, Table 5 shows the metadata scores for the normal program.

For a normal program, a lower score indicates a smaller percentage of attack labels in the metadata. The average or maximum score increases from the reference for four of the six parameters, but the increase is small enough to maintain the overall score.

In the range of parameters to be evaluated, $x = 10$, $y = 1000$ is the most sparse sampling and the combination that maximizes the reduction rate. According to Eq. 2, we find that the implementation of the sampling circuit can reduce

the operating rate of the metadata generator by approximately 99% in the evaluation environment.

6.5 Evaluation of lightweight method

In this chapter, we present the evaluation of a method to reduce the model size using sampling learning and feature dimensionality reduction. First, we observed the changes in the parameters and scores of sampling learning and determined the parameters that do not interfere with the judgment. Next, we calculated the contribution rate of the model creation, observed the number of features and the metadata scores output by the model, and selected suitable features for the weight reduction evaluation. Finally, we compared and evaluated models created using the conventional method and the lightweight method.

6.5.1 Preliminary evaluation of sampling learning

We evaluated the accuracy and size reduction rate of machine learning models using sampling learning. We sampled the training data with the same candidate values as Sect. 6.4 for x and y . Metadata was generated using 12 models trained on the sampled data, and the accuracy of AMH was evaluated using the same procedure as in Sect. 6.4.

Figure 10 shows x , y , Threshold, and the change in accuracy.

The bars, grouped by Threshold, indicate accuracy. Compared to the sampling circuit, the overall accuracy tends to decrease, and only Threshold = 0.1 in (a) achieves 100% accuracy regardless of the values of x and y . In the graphs after (b), the number of parameters that fail to make a decision gradually increases. In (f), which shows Threshold = 1.0, more than one software fails to be determined, regardless of x and y . Since the judgments start to fail after Threshold = 0.3, $x = 1$, $y = 5000$, we limited the range of parameters to evaluate the reduction rate to $x \geq 10$ and $y \leq 1000$.

Fig. 10 Parameters and accuracy in sampling learning

Table 6 Metadata scores of malware using sampling training

y	x = 10			x = 100		
	Avg	Min	Max	Avg	Min	Max
100	0.963	0.901	1.000	0.984	0.933	1.000
500	0.938	0.776	1.000	0.993	0.983	1.000
1000	0.970	0.892	1.000	0.956	0.811	1.000

The reference scores are as follows: Average = 0.986, Min = 0.958, Max = 1.000. Bold indicates a score that has **decreased** by more than 0.01 from the reference score

The scores for each parameter are also described in this section. Table 6 shows the metadata scores of malware specimens.

The values in the table are the average, minimum, and maximum scores of the six malware samples. For example, for $x = 100$, $y = 500$, the mean, minimum, and maximum scores are 0.969, 0.873, and 1.000, respectively. Bold type indicates scores that decreased by more than 0.01 from the reference. Compared to Table 4, which summarizes the scores of sampling circuits, the scores tend to decrease in several combinations.

Table 7 shows the metadata scores for the normal program.

While the score of the malware tended to decrease, the score of the normal program did not increase in a way that affected the accuracy. As shown by the mean and maximum, the sampling learning resulted in lower scores overall for both malware and normal programs. In the range of param-

Table 7 Metadata scores of normal program using sampling training

y	x = 10			x = 100		
	Avg	Min	Max	Avg	Min	Max
100	0.002	0.000	0.013	0.010	0.000	0.057
500	0.003	0.000	0.011	0.005	0.000	0.023
1000	0.004	0.000	0.016	0.003	0.000	0.011

The reference scores are as follows: Average = 0.014, Min = 0.000, Max = 0.080. Bold indicates a score that has **increased** by more than 0.01 from the reference score

eters to be evaluated, $x = 10$, $y = 1000$ is the most sparse sampling and is the combination that maximizes the reduction rate. In this case, the training data is reduced by about 99%. However, for the final model size, it is necessary to check the file output from the actual training process. We compare and evaluate the actual size of the output models with dimensionality reduction in Sect. 6.5.3.

6.5.2 Preliminary evaluation of dimensional reduction

The emulator of the metadata generator is based on Random Forest, which is equipped with a function to output the contribution rates of features. In conventional methods, contribution rates are output at the stage of creating the metadata generator, and unnecessary features are examined. To evaluate the dimensionality reduction, we trained the model with features according to the following criteria:

Table 8 Contribution rates

Rank	Feature	Contribution rate (%)
1	Hit rate of L1 instruction cache	25.76
2	Hit rate of L1 data cache	21.48
3	Instruction address	21.46
4	Distance of NOP	14.89
5	L2 hit rate	11.05
6	2nd register value	1.62
7	Load/Store address or target PC	1.42
8	1st register value	0.71
9	Distance of Load	0.39
10	Distance of Jump	0.29
11	Distance of Store	0.23
12	Operation code	0.14
13	2nd register number	0.14
14	1st register number	0.10
15	3rd register value	0.06
16	Hit or Miss by BTB	0.04
17	Distance of Other Operation	0.04
18	3rd register number	0.03
19	Predicted branch direction	0.03
20	Instruction type	0.03
21	Branch direction	0.01
22	Conditional flag	0.00

- Features with a contribution rate of more than 1%
- Features with a contribution rate of more than 10%

The metadata of malware and normal programs were output by these two models, and the scores were compared with the reference model. Based on the comparison results, we selected a dimensionality reduction model suitable for evaluation, and evaluated the weight reduction method with three models: the reference model, the sampling learning model, and the model with dimensionality reduction.

Table 8 shows the processor information output at the time of model creation in the order of contribution rate.

The ruled lines in the table indicate the boundaries between the contribution rates of 10% and 1%. We created *clf_Ref*, in which all the features were trained, *clf_1*, in which features with a contribution rate of 1% or higher were trained, and *clf_10*, in which only features with a contribution rate of 10% or higher were trained.

6.5.3 Model size evaluation

We compared and evaluated the sizes of the models created by the conventional method and the models with sampling learning and dimensionality reduction. The evaluation condition for sampling learning was $x = 10$, $y = 1000$. For dimensionality reduction, we selected models that learned

Table 9 Model size compare

Method	Normalized hardware size	KB
Reference	100	346
Sampling training	16	56
Dimensional reduction	19	67

only features with a contribution rate of 1% or higher. Table 9 shows the results of the model size comparison.

When the size of the reference model was 100, the model size could be reduced to 16 by applying sampling learning. By applying dimensionality reduction, the model size was reduced to 19.

7 Hardware performance

The contribution of the proposed method is the reduction of power and hardware resource consumption. In Sect. 6.4, we evaluated a sampling circuit that improves the utilization rate and showed its effectiveness. In addition, in Sect. 6.5.3, we implemented two approaches that reduce the weight of ML models. In this chapter, in order to evaluate these hardware contributions quantitatively, we designed the sampling circuit and ML model in Verilog language, and performed

Table 10 Measurement environment of hardware

Measurement tool	Vivado 2018.2.2 HL WebPACK Edition
Target device	XC7Z020-2CLG484I
RAM	512 MB

Table 11 Hardware resource consumption

Category	Target	CLB
ML model	Reference	74,498
ML model	Sampling training	13,171
ML model	Dimensional reduction	6505
Circuit	Sampling circuit	26

It should be noted that the results assume the case of the target FPGA device, the actual power consumption is decided by which device is implemented on: manufacturing process, architecture, and others

logic synthesis and place-and-route on a targeting Field Programmable Gate Array (FPGA).

7.1 Hardware evaluation environment

Table 10 shows the environment for measuring power and hardware resources consumption.

We used Vivado, an integrated development tool from Xilinx, for logic synthesis and implementation. We also specified Xilinx's XC7Z020-2CLG484I as the target FPGA in Vivado.

7.2 Evaluation of hardware resource consumption

Table 11 shows the measurement results of the hardware resource consumption.

A configurable logic block (CLB) is the multiple basic logic resources in an FPGA, and the number of look-up tables (LUTs) contained in the CLB is an indicator of hardware resource consumption. In the ML model, the number of LUTs for the Reference is 74498, while the number of LUTs for the model with dimensionality reduction is 6505, resulting in a significant reduction. From another point of view, the number of LUTs is reduced to 8.7% compared to the Reference, and the weight reduction effect is higher compared to the emulation results shown in Table 9. On the other hand, the number of LUTs for the sampling circuit is much lower than that of the ML model, indicating that it can be implemented with a small scale circuit.

7.3 Evaluation of power consumption

At first, we evaluated the power consumption of ML model and the sampling circuit when each circuit performs one operation in the case of the target FPGA device. The results show

that reference ML model, the one with sampling training, the one with dimensional reduction, and the sampling circuit are 18.95 W, 14.16 W, 14.55 W, and 0.19 W, respectively.

When the sampling training and the dimensional reduction are introduced, the power consumption is reduced by 23.2% and 25.3%, respectively. However, Table 12 does not indicate the utilization rate reduction effect of our sampling circuit, since the ML model for Reference with sampling circuit only operates when it is activated by the sampling circuit which always operates.

Therefore, at second, we evaluate the total hardware power consumption including the one for the ML model and the one for the sampling circuit. The variables α and $C V_{DD}^2 f$ shown in Eq. 5 correspond to the utilization rate R_u (Ref. Eq. 1) and power consumption of ML model P_{ML} (ref. Table 12), respectively. Let P_{SC} be the power consumed by the sampling circuit. When each variable is $W = 18.95$, $x = 10$, $y = 1000$, $P_{SC} = 0.19$, respectively, the total power consumption P_{total} is calculated as follows:

$$P_{total} = \frac{x}{x+y} P_{ML} + P_{SC} = \frac{10}{10+1000} 18.95 + 0.19 \approx 0.38. \quad (6)$$

Table 13 summarizes the power consumption reduction effect of the sampling method.

The values in the table are, from left to right, the enable or disable of the sampling circuit, the parameters when the sampling circuit is enabled, and the total power consumption of the sampling circuit and the ML model. The results show that the total power consumption can be reduced significantly by the sampling circuit. Equation 6 shows that the power consumption is proportional to R_u when the sampling circuit is enabled. Therefore, we can optimize the power reduction rate by the parameters x and y with maintaining 100 % accuracy.

8 Discussion

In this chapter, we will discuss the evaluation results shown in Sect. 6.

8.1 Predictive detection timing

In the evaluation of predictive detection using malware specimens, a difference of more than 10% was observed in the detection timing, depending on the parameters. Thus, using Trojan_1 as an example, we analyzed the features that appear in the metadata and the timing of detection. Figure 11 shows the metadata and two patterns of detection timing for Trojan_1.

The figure shows Trojan_1 extracted from Fig. 4a with the first half enlarged. The red vertical dashed line on the left is

Table 12 Each hardware power consumption per one operation

Category	Target	Power consumption (W)
ML model	Reference	18.95
ML model	Sampling training	14.16
ML model	Dimensional reduction	14.55
Circuit	Sampling circuit	0.19

It should be noted that the results assume the case of the target FPGA device, the actual power consumption is decided by which device is implemented on: manufacturing process, architecture, and others

Table 13 Total hardware power consumption for the ML model with the sampling circuit

Sampling circuit	Parameter	Power consumption (W)
Disable	N/A	18.95
Enable	$x = 100, y = 100$	9.67
Enable	$x = 10, y = 1000$	0.37

Fig. 11 Detection timing of Trojan_1. High sensitive parameter (left side): $W = 100$, $S_t = 0.4$. Low sensitive parameter (right side): $W = 1000$, $S_t = 1.0$

the detection timing under the condition of high sensitivity to attack labels with parameters $W = 100$ and $S_t = 0.4$. For the highly sensitive parameters, the attack labels are detected at the timing immediately after execution, when the percentage of attack labels increases instantaneously.

The second dashed line shows the detection timing for a parameter with low sensitivity to attack labels ($W = 1000$, $S_t = 1.0$). In this case, the system detects the attack label when it appears for 1000 consecutive cycles, which corresponds to the concentration of the attack label being in the interval 0.10–0.15 on the horizontal axis. Noting the hypothesis that there are always intervals in malware metadata with a concentration of attack labels, there is a tradeoff between the sensitivity of predictive detection and the detection speed. However, we believe that the number of specimens handled in this study is insufficient to cover all the features of malware, and so it is necessary to continue collecting metadata.

Figure 7 from the preliminary evaluation of predictive detection shows that many parameters have an accuracy of 91.7% (11/12). The exception is the $\times 264$ command of the

Fig. 12 Detection timing of $\times 264$ command ($W = 100$, $S_t = 0.9$)

normal program. Figure 12 shows an example of $\times 264$ being incorrectly identified as malware by predictive detection.

The dashed line indicates the timing of the detection. In the case of $W = 100$, $S_t = 0.9$, the $\times 264$ command is determined to be malware by predictive detection at the 54811th cycle. Even if the metadata as a whole contains less than half of the attack labels, when the number of attack labels suddenly increases, the parameters that have not been sufficiently studied will induce false positives in normal programs. Figure 12 is a typical example.

The dashed line indicates the timing of the detection. In the case of $W = 100$, $S_t = 0.9$, the $\times 264$ command is determined to be malware by predictive detection at the 54811th cycle. Even if the metadata as a whole contains less than half of the attack labels, when the number of attack labels suddenly increases, the parameters that have not been sufficiently studied induce false positives in normal programs.

(a) Malware

(b) Normal program

Fig. 13 Metadata with sampling circuit ($x = 100$, $y = 1000$)

8.2 Sampling circuit

In the evaluation results shown in Sect. 6.4, the operation rate of the metadata generator decreased while maintaining accuracy by appropriately setting the parameters of the sampling circuit. Therefore, we observed the change in metadata when the sampling circuit was implemented. Figure 13 shows the metadata output via the sampling circuit with $x = 100$, $y = 1000$.

Compared to the metadata without the sampling circuit shown in Fig. 4, the implementation of the sampling circuit has smoothed the metadata instead of reducing the number of labels. In particular, the Trojan_1 and $\times 264$ commands originally had irregular label appearances, but these irregularities are absorbed. We initially implemented the sampling circuit to reduce the utilization rate. However, with the confirmed effect of smoothing the metadata, the sampling circuit can also be expected to prevent false positives of normal programs in predictive detection, such as that shown in Fig. 12.

(a) Malware

(b) Normal program

Fig. 14 Metadata with sampling learning ($x = 10$, $y = 500$)

8.3 Lightweight

In Sect. 6.5.1, we performed sampling training and evaluated the scores output from the model. From Table 6, when $x = 10$, $y = 500$, the minimum score is reduced by more than 0.2 from the reference. As an example of a malware training failure, Fig. 14 shows the metadata output from a model trained to sample at $x = 10$, $y = 500$.

Among the generated metadata, the metadata of the malware is even more irregular, particularly for Kaiten_1. In the conventional method, Kaiten_1 is relatively smooth, but in the example of Fig. 14, it becomes irregular from around 0.2 on the horizontal axis to 0.8. By contrast, the $\times 264$ command, which was the only normal program with irregular metadata, was smoothed by sampling learning.

As mentioned above, the attack labels decrease for both malware and normal programs. As indicated by the parameter $x = 10$, $y = 500$, the interval is large relative to the number of cycles per extraction. Therefore, it is highly likely that the training data (processor information) necessary to output the attack labeling during the sampling process is included in the interval y . In addition, considering the bias in the amount of training data between malware and normal programs (see

Tables 2 and 3, respectively), there is a possibility that this induces labeling that is more normal. As the same tendency is observed in other conditions, we determine that in our evaluation environment, using a model with low labeling accuracy makes it difficult for attack labels to appear overall.

9 Conclusion

In this study, we proposed circuits and learning methods to achieve high efficiency, low power consumption, and light weight for malware detection hardware that uses processor information as a feature value. To investigate these three methods, we focused on the time-series metadata that can be obtained from the processor information. For efficiency, we implemented predictive detection in the malware detection part of the proposed mechanism, and we performed behavioral detection to predict metadata characteristics. Using predictive detection, we were able to detect malware in 19% of the execution cycles a conventional method requires. We implemented a sampling circuit to achieve low power consumption and reduce the processor information output from the core. We were able to reduce the operating rate of the metadata generator by approximately 99%. To achieve weight reduction, we focused on the learning process of the metadata generator based on machine learning models. By separately verifying the sampling learning and feature dimensionality reduction in the learning process, we succeeded in creating a metadata generator that was 16% of the conventional method size, while maintaining labeling performance.

Comparing with the previous works using HPC, the HPC-MalHunter [13] achieves the detection rate of 90.69 % and an false positive rate of 0.79 % with making decision after running 400,000 instructions from the beginning; on the other hand, our AMH accomplishes the 100 % detection rate and the 0% false positive rate with the high detection speed requiring only less than instructions because of utilizing time-series metadata of ML model.

The predictive detection proposed in this study calculates the percentage of attack labels every time the processor information is output for one cycle. Therefore, there is a concern that the overhead will be large even if it is implemented as hardware. As a countermeasure, we envisage a method that combines predictive detection and sampling circuits. However, since predictive detection assumes that the processor information is not sampled, future work should reexamine the predictive detection algorithm based on the sampling circuit.

In addition, there is an issue of the ML model and signature updates. In this paper, the proposed ML model, that is implemented as hardware, does not have an update mechanism. Therefore, it is difficult to handle new malware strains

that behave similar to normal programs. Future work should introduce a configurable mechanism to update the ML model repeatedly.

Acknowledgements A part of this research was supported by JSPS KAKENHI Grant Number 20K11818, 19K11968 and 19H04108.

Funding This study was funded by Japan Society for the Promotion of Science (Grant Number 20K11818, 19K11968 and 19H04108).

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of things (IoT): a vision, architectural elements, and future directions. *Future Gen. Comput. Syst.* **29**(7), 1645–1660 (2013)
2. Taylor, S.: The next generation of the internet. CISCO Point of view (2013)
3. (2018) OWASP Foundation | OWASP IoT Top 10. <https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf>, (Accessed on 08/21/2021)
4. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.: DDoS in the IoT: Mirai and Other Botnets. *Computer* **50**(7), 80–84 (2017). <https://doi.org/10.1109/MC.2017.201>
5. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.: Ddos in the iot: Mirai and other botnets. *Computer* **50**(7), 80–84 (2017). <https://doi.org/10.1109/MC.2017.201>
6. Elhadi, A., Maarof, M., Hamza Osman, A.: Malware detection based on hybrid signature behaviour application programming interface call graph. *Am. J. Appl. Sci.* **9**, 283–288 (2012)
7. Li, J., Sun, L., Yan, Q., Li, Z., Srisa-an, W., Ye, H.: Significant permission identification for machine-learning-based android malware detection. *IEEE Trans. Ind. Inform.* **14**(7), 3216–3225 (2018). <https://doi.org/10.1109/TII.2017.2789219>
8. Sewak, M., Sahay, S.K., Rathore, H.: Comparison of deep learning and the classical machine learning algorithm for the malware detection. In: 2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp 293–296. <https://doi.org/10.1109/SNPD.2018.8441123> (2018)

9. Bazrafshan, Z., Hashemi, H., Fard, S.M.H., Hamzeh, A.: A survey on heuristic malware detection techniques. In: The 5th Conference on Information and Knowledge Technology, pp 113–120. <https://doi.org/10.1109/IKT.2013.6620049> (2013)
10. Takase, H., Kobayashi, R., Kato, M., Ohmura, R.: A prototype implementation and evaluation of the malware detection mechanism for IoT devices using the processor information. *Int. J. Inf. Secur.* **19**(1), 71–81 (2020). <https://doi.org/10.1007/s10207-019-00437-y>
11. Foreman, J.C.: A survey of cyber security countermeasures using hardware performance counters. [arXiv:1807.10868](https://arxiv.org/abs/1807.10868) (2018)
12. Torres, G., Liu, C.: Can data-only exploits be detected at runtime using hardware events? A case study of the heartbleed vulnerability. In: HASP 2016: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, Association for Computing Machinery, New York, NY, USA, HASP 2016. <https://doi.org/10.1145/2948618.2948620> (2016)
13. Bahador, M.B., Abadi, M., Tajoddin, A.: Hpcmalhunter: behavioral malware detection using hardware performance counters and singular value decomposition. In: 2014 4th International Conference on Computer and Knowledge Engineering (ICCKE), pp 703–708. <https://doi.org/10.1109/ICCKE.2014.6993402> (2014)
14. Nomani, J., Szefer, J.: Predicting program phases and defending against side-channel attacks using hardware performance counters. In: Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, Association for Computing Machinery, New York, NY, USA, HASP '15. <https://doi.org/10.1145/2768566.2768575> (2015)
15. Alam, M., Bhattacharya, S., Mukhopadhyay, D., Bhattacharya, S.: Performance counters to rescue: a machine learning based safeguard against micro-architectural side-channel-attacks. *IACR Cryptol. ePrint Arch.* **2017**, 564 (2017)
16. Jyothi, V., Wang, X., Addepalli, S.K., Karri, R.: Brain: Behavior based adaptive intrusion detection in networks: using hardware performance counters to detect ddos attacks. In: 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), pp 587–588. <https://doi.org/10.1109/VLSID.2016.115> (2016)
17. Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., Guan, H.: Transparent and efficient cfi enforcement with intel processor trace. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 529–540. <https://doi.org/10.1109/HPCA.2017.18> (2017)
18. Ding, Li, Kang, Ping, Yin, Wenbo, Feng, Zhi-Hua: Design and implementation of hardware-based low latency TCP offload engine for 10 Gbps Ethernet. In: 2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), pp 701–703. <https://doi.org/10.1109/ICSICT.2016.7999016> (2016)
19. (2020) Ethos-u55 machine learning processor—arm. <https://www.arm.com/ja/products/silicon-ip-cpu/ethos/ethos-u55>. Accessed 21 Aug 2021
20. (2021) Intel hardware shield overview. <https://www.intel.co.jp/content/dam/www/central-libraries/us/en/documents/white-paper-intel-hardware-shield.pdf>. Accessed 21 Aug 2021
21. (2021) Intel® threat detection technology. <https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/threat-detection-technology-solution-brief.pdf>. Accessed 21 Aug 2021
22. Stiles, D.: The hardware security behind azure sphere. *IEEE Micro* **39**(2), 20–28 (2019). <https://doi.org/10.1109/MM.2019.2898633>
23. Koike, K., Kobayashi, R., Katoh, M.: Reduction of classifier size and acceleration of classification algorithm in malware detection mechanism using processor information. In: 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW), pp 339–345. <https://doi.org/10.1109/CANDARW.2019.00066> (2019)
24. Weste, N., Eshraghian, K.: CMOS VLSI Design: A Circuits and Systems Perspective. Addison Wesley Longman Publishing Co (1985)
25. Alves, M.A.Z., Villavieja, C., Diener, M., Moreira, F.B., Navaux, P.O.A.: Sinuca: A validated micro-architecture simulator. In: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, pp 605–610. <https://doi.org/10.1109/HPCC-CSS-ICCESS.2015.166> (2015)
26. Akram, A., Sawalha, L.: A survey of computer architecture simulation techniques and tools. *IEEE Access* **7**, 78,120–78,145 (2019). <https://doi.org/10.1109/ACCESS.2019.2917698>
27. Austin, T., Larson, E., Ernst, D.: Simplescalar: an infrastructure for computer system modeling. *Computer* **35**(2), 59–67 (2002). <https://doi.org/10.1109/2.982917>
28. Kim, H.K., Kim, H.S., Eun, C.M., Cho, H.H., Jeong, O.H.: A high-performance branch predictor design considering memory capacity limitations. In: 2017 International Conference on Circuits, System and Simulation (ICSSS), pp 49–53. <https://doi.org/10.1109/CIRSYSSIM.2017.8023180> (2017)
29. Arora, H., Bhatia, U., Pandita, S.: Comparative study of different cache models on simplescalar architecture. In: Fourth International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom2012), pp 302–304. <https://doi.org/10.1049/cp.2012.2555> (2012)
30. Du, Z., Xia, B., Qiao, F., Yang, H.: System-level evaluation of video processing system using simplescalar-based multi-core processor simulator. In: 2011 Tenth International Symposium on Autonomous Decentralized Systems, pp 256–259. <https://doi.org/10.1109/ISADS.2011.34> (2011)
31. QEMU.: QEMU. <https://www.qemu.org/> (2019). Accessed 21 Aug 2021
32. VirusTotal.: VirusTotal. <https://www.virustotal.com/gui/home/upload> (2020). Accessed 21 Aug 2021
33. GitHub (2018) GitHub—micheloosterhof/cowrie-dev. <https://github.com/micheloosterhof/cowrie-dev>. Accessed 21 Aug 2021

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.