Edge-Based Intrusion Detection for IoT devices

ANAND MUDGERIKAR, Purdue University, West Lafayette, Indiana, USA
PUNEET SHARMA, Hewlett Packard Labs, Milpitas, CA, USA
ELISA BERTINO, Purdue University, West Lafayette, Indiana, USA

As the Internet of Things (IoT) is estimated to grow to 25 billion by 2021, there is a need for an effective and efficient Intrusion Detection System (IDS) for IoT devices. Traditional network-based IDSs are unable to efficiently detect IoT malware and new evolving forms of attacks like file-less attacks. In this article, we present a system level Device-Edge split IDS for IoT devices. Our IDS profiles IoT devices according to their "behavior" using system-level information like running process parameters and their system calls in an autonomous, efficient, and scalable manner and then detects anomalous behavior indicative of intrusions. The modular design of our IDS along with a unique device-edge split architecture allows for effective attack detection with minimal overhead on the IoT devices. We have extensively evaluated our system using a dataset of 3,973 traditional IoT malware samples and 8 types of sophisticated file-less attacks recently observed against IoT devices in our testbed. We report the evaluation results in terms of detection efficiency and computational.

CCS Concepts: • **Security and privacy** → **Intrusion detection systems**; **Malware and its mitigation**; **Security protocols**;

Additional Key Words and Phrases: Intrusion detection, IoT security, malware, edge, AI

ACM Reference format:
Anand Mudgerikar, Puneet Sharma, and Elisa Bertino. 2020. Edge-Based Intrusion Detection for IoT devices. *ACM Trans. Manage. Inf. Syst.* 11, 4, Article 18 (October 2020), 21 pages.
https://doi.org/10.1145/3382159

1 INTRODUCTION

With the growing use of IoT devices in health care, transportation, home appliances, and smart cities, security of these devices is a primary concern. The lack of security is evident in the huge number of IoT devices that have been compromised and exploited for launching large-scale DDoS attacks like Mirai in 2016, Hajime in early 2017, BrickerBot and IoT Reaper later in 2017, and Hakai in 2018 [12, 22]. The alarming fact about these attacks is that these malwares are basic and straightforward when compared to malware used to attack traditional computer systems. We should thus expect more sophisticated IoT malware in the future. It might even be the case that such advanced malware are already present on current devices going completely undetected. An example of such evolution of IoT malware is the Reaper malware [17] which is a successor to Mirai. It builds on

This work was initiated as part Anand Mudgerikar's summer internship at Hewlett Packard Labs.
Authors' addresses: A. Mudgerikar and E. Bertino, CS Department, Purdue University, 305 N. University Street West Lafayette, IN 47907; emails: {amudgeri, bertino}@purdue.edu; P. Sharma, Hewlett Packard Labs, 940 N. McCathy Blvd, Milpitas, CA 95035; email: puneet.sharma@hpe.com.

2158-656X/2020/10-ART18 $15.00
https://doi.org/10.1145/3382159

Mirai's code by adding various known exploits for different IoT devices/architectures, along with the simple password-guessing techniques of Mirai, to its infection methods. Recently, file-less attacks have been observed along with traditional malware-based attacks against IoT devices [15]. Such attacks are significantly harder to detect and comprise a high percentage of recently seen attacks in the wild.

To keep up with evolution in malware, we must also keep evolving our security techniques. Achieving comprehensive security for IoT devices and systems requires combining different layers of security techniques and systems [5]—among which intrusion detection is one of the most important. Thus, evolution of intrusion detection techniques for IoT devices is of utmost importance.

There has been significant research in designing intrusion detection systems (IDSes) for traditional computer systems, such as Snort [36] and Bro [33], and more recently for IoT devices, such as Svelte [35], Kalis [28] and Heimdall [18]. Most IDSes for IoT devices are network-traffic-based and detect attacks by analyzing the network traffic for either anomalies or attack signatures. The main problem of these IDSes is that, because of the large amounts of network traffic in today's IoT networks coupled with the heterogeneity of IoT devices in terms of protocols, manufacturers, applications, and so on, they tend to miss some attacks and have a significant number of false positives. To overcome those shortcomings, we propose `E-Spion`, which monitors and analyzes system data rather than network traffic of IoT devices.

Motivation. System-level IDSes like anti-viruses, commonly used for traditional computer systems, employ attack-signature-based detection. System-level-anomaly-based detection in such IDSes is not practical as a traditional computer system runs a number of different kinds of applications which might be very similar to malware in terms of computing operations and commands. So, it becomes very difficult to differentiate between benign applications and malware, resulting in high numbers of false positives and false negatives. This, however, is not the case with IoT devices.

In general, IoT devices have a primary function for which computation is required. For example, a DVR is meant to record and store videos but computation is required for this, like receiving an input video stream on a network socket and then storing the files in a local database. Similarly, a television is meant to display, a camera is meant to record videos, a car is meant to drive safely, and so on. We see that the computations done on the device are a means to an end or the main function. This is not the case with traditional computer systems, like desktops and laptops, where computation is the end goal. These devices can run multiple applications and allow us to perform computations, like browsing the Internet, performing calculations, playing games, and so on. So, we see intuitively that the IoT devices, like DVRs, cameras, cars, televisions, and the like, have a main function and these IoT devices, when not compromised, should be performing just this main function and nothing else. We also note that these main functions are periodic in nature and consistently repeat themselves with different arguments after device-specific time intervals. We use these intuitions to build IoT device profiles that are later used for anomaly detection.

Approach. At a high level, the goal of our IDS is to identify the main functions of the IoT device in terms of data collected from the running processes and system calls made by these processes, in order to create the baseline behavior profile of the IoT device. We then continuously monitor the device and use this baseline to detect anomalous behavior that may be indicative of intrusions or other malicious activities.

Contributions. To summarize, we make the following contributions in this article:

(1) A system-level IDS that uses anomaly detection to detect attacks on IoT devices in an efficient and scalable manner.

(2) An approach to build baseline behavior profiles of IoT devices according to system information (device logs) collected from running processes and system calls on these devices.
(3) A device-edge split architecture with the server components running on the network edge-server performing the bulk of the computational work and the components running on the IoT device performing minimal work.
(4) A three-layer anomaly detection engine with each more advanced layer providing more fine-grained accuracy but at the same time having higher overhead costs on the device.
(5) An extensive evaluation of our system using 3,973 malware samples assembled from recent attacks on IoT devices and an analysis of the malware samples in terms of our device logs.
(6) The first IDS for IoT devices effective against 8 types of file-less attacks.

The rest of this article is organized as follows: First we give some background on how IoT attacks are propagated, different types of IoT malware, file-less attacks, and the IoT security architecture in general. Then, we discuss in detail the design of our system and its components in Section 3. Next we discuss some of the key implementation challenges and analyze the design decisions made in Section 4. We perform the evaluation in terms of detection efficiency and overhead costs in Section 5. In Section 6, we discuss and compare related work in this area and finally conclude in Section 7.

2 BACKGROUND

2.1 IoT Attacks

Most of the IoT attacks comprise of three operation stages: injection, infection, and attack. The *injection stage* involves gaining "control" of the IoT device (getting root access in most cases) through credential hijacking, password brute-forcing/dictionary attacks or utilizing known device/system/firmware vulnerabilities [3, 42]. The injection stage follows in which the attacker "prepares" the device by performing operations on the device like setting up communication with the bot master (C&C server), downloading required malware/rootkits, stopping security services, and so on. Most of the popular IoT attacks involve downloading some form of malware on to the device in the "prepare" stage but the recently seen file-less attacks do not follow this trend. Some attacks might even skip the prepare step and proceed directly to the attack stage. Finally, the "attack" stage includes performing Denial of Service (DoS), coordinated DDoS attacks, ransom attacks, bitcoin mining, device-specific malicious/unsafe activities, and the like [13, 40].

2.2 File-Less Attacks

Recently another class of attacks has been observed which does not involve downloading any malware during the infection stage. This makes attacks harder to detect since malware fingerprinting is the primary detection technique used in traditional IDSs. The attacks involve setting up backdoors, port-forwarding, and the like using shell scripting or modifying system-level files. They can be classified into eight categories [15] described below which we use for our evaluation:

(1) *Type 1*: Changes the password of the device using the *passwd* command; it thus locks up the device and does not allow legitimate users to access it.
(2) *Type 2*: Removes certain config files or system programs using the *rm* or *dd* commands. The goal is to remove watchdog and other security service daemons so that the infection/malfunction is not detected.
(3) *Type 3*: Stops certain system processes or services using the *kill* or *service* commands. The goal is similar to Type 2 in that the infection is not detected.

(4) *Type 4*: Retrieves system information like architecture, operating system, and networking/process information using commands like *lscpu, uname, netstat, ps,* and so on. The goal is to gain more information about the device and network for further attacks.
(5) *Type 5*: Steals user information using the *cat* command to read stored hashed passwords, config files, and the like. The goal is to breach privacy, learn user information, and/or behavior patterns.
(6) *Type 6*: Launches network attacks through malformed HTTP requests to web servers using the *wget* or *curl* commands exploiting known vulnerabilities like HeartBleed, SQL injection, and the like. The goal is to propagate the attack in the local network and compromise other devices on the network.
(7) *Type 7*: Uses various shell commands for collecting device/user data like *who, help, lastlog, sleep,* and so on. The goal is to learn how the user uses the device and if other users are using the device.
(8) *Type 8*: Sets up port forwarding using either the *ssh* or *iptables* utilities in order to use the device as a port forwarding proxy so that the real IP address of the attacker is masked.

2.3 IoT Security Architecture

The IoT security solutions and services can be broadly classified into two categories: centralized-cloud-based and distributed-edge-based. The current trend in enterprise security is towards providing a security overlay network [37] using a centralized cloud architecture where the service providers have primary ownership of the data and IDS service. The advantage of such services is the flexibility in deployment and management, lower infrastructure costs, performance benefits and a centralized point of control. However, a completely centralized-cloud-based security architecture is not scalable [2, 7] for the distributed nature of an IoT environment because of the huge number of devices, high volume of data, low-latency requirements, ad-hoc environments and user privacy concerns. The edge-based security architecture proposed in this work follows the fog computing paradigm [6, 43] where the main workload of the IDS is performed at the edge device rather than at the cloud. It is important to note that the cloud computing resources can be utilized as an additional layer over the edge as discussed later in Section 4.

3 DESIGN

In this section, we first provide an overview of our system design. We then discuss in detail our anomaly detection engine and the typical life cycle of a device in the network. Finally, we detail our log authentication scheme (hash-chain-verifier) for secure device log transfer.

3.1 Design Overview

Our system, called `E-Spion`, proposes a novel device-edge split architecture with two components: a server side (Edge-Server) and a client side (Device) (see Figure 1). The server component is maintained on the edge system or gateway router of the network. The client component is installed on each IoT device connected to the edge system. The client component is responsible for recording all system logs on the device and periodically transferring them to the edge server. The edge-server and IoT devices communicate periodically via a secure encrypted and authenticated channel. All the computationally intensive operations, i.e., parsing logs to extract features, authentication of logs, training classifiers, running classifier models, module management, and so on. , are performed on the edge-server. We have employed such a local edge compute strategy to minimize the workload on the IoT devices.

Our device 3-layered behavior profile is built in three layers using three types of device logs (one for each layer) obtained from three types of information: running process names, running process

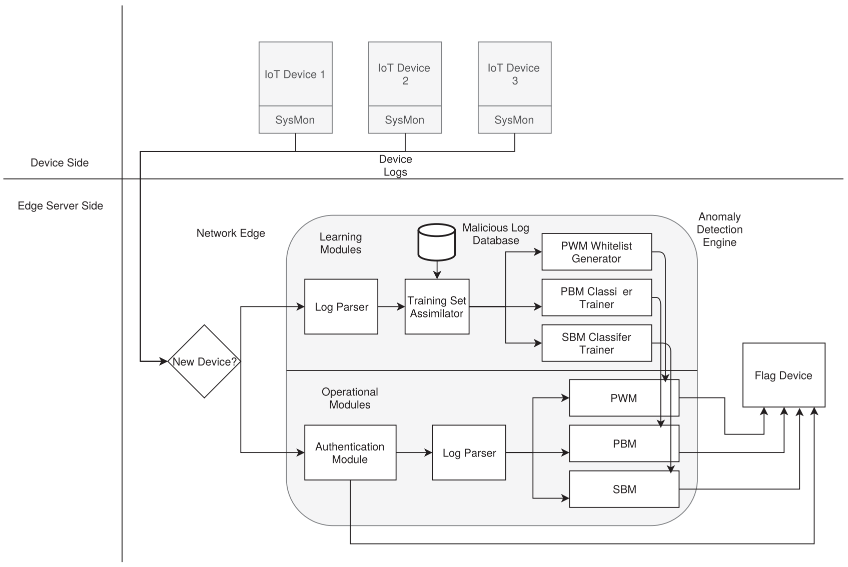

Fig. 1. Architecture of `E-Spion`.

parameters, and system calls made by these processes. Since each of these log types has different overheads for recording, storing, and analyzing, we maintain three separate modules which handle each type of device log, namely Process White listing Module (PWM), Process Behavior Module (PBM), and System-call Behavior Module (SBM). These modules can run concurrently with different configuration values (recording intervals, sleep times, etc.) according to the device/network requirements (resource consumption, associated risk, etc.). All three modules are managed by a module manager which helps them inter-operate efficiently and according to the user/device requirements. The modules interact with each other using the common module manager to improve overall detection efficiency, provide more fine grained intrusion alerts and reduce overhead on the devices.

3.2 Anomaly Detection Engine

As stated before, our anomaly detection engine is organized into three detection modules as follows:

PWM. This module uses a whitelisting-based approach and is the least expensive module. In the learning stage, our system monitors all the benign processes running on the device and builds a device specific white-list of all running processes. The system collects all the running process names and PIDs during the operation stage and compares them with the device white-list to detect anomalous new processes. The goal is to detect simple malware which spawn new malicious processes on the device, as efficiently as possible.

PNM. This module monitors the behavior of each running process on the device and detects any process behaving anomalously. The module monitors various parameters for each running process on the device during the learning stage. After collecting the logs during the learning stage, we train a machine learning classifier for each device and use it for detecting anomalous process behavior. We extract 8 metrics/features (see Table 1) from the parameters and use them in our PBM classifier model. This module is more expensive than the PWM but it provides more fine-grained detection and is able to detect more sophisticated malware that have the ability to masquerade as benign processes rather than spawning new malicious processes.

Table 1. Metrics for the *PBM* Module

Metric #	Metric Name	Description
1	SysCPU	CPU time consumption of the process in system mode (kernel mode)
2	UsrCPU	CPU time consumption of the process in user mode
3	CPU Usage	Overall CPU utilization
4	RGROW	The amount of resident memory that the process has grown during the last interval
5	VGROW	The amount of virtual memory that the process has grown during the last interval.
6	WRDSK	The number of write accesses issued physically on disk
7	RDDSK	The number of read accesses issued physically on disk
8	Instance Count	The number of instances of the process spawned in the interval

SBM. This module monitors the behavior of each running process on the device according to system calls issued by the process. This is the most expensive module but it provides the most effective and fine-grained detection strategy. The module monitors 34 different kinds of system calls issued by each running process as described in Table A.1. We list the metrics/features used for training our classifier in Table 2. For each type of system call recorded, we monitor four metrics/features: number of calls made (#.1), %of time taken (#.2), total time taken (#.3) and average time taken per each call (#.4). So, in total we have $34 * 4 = 136$ metrics in our SBM classifier model.

3.3 Life Cycle of a Device

For the purpose of our IDS system, a device in the network goes through the following phases: Initialization, Learning, Operation, and finally Anomaly Detection.

Initialization Phase: When a new device joins the network, a new empty device profile and public-private key pair are created on the edge-server. Then the private key is uploaded to the device and all SSH sessions between the edge and device are secured using this key pair. The corresponding device architecture (Mips, Arm, x86, Spark, and so on) specific client side modules are transferred to the device. The client side modules include Linux shell script recording modules for the PWM, PBM, and SBM modules, scripts to create and maintain hash chains for log authentication, and finally scripts to securely transfer logs from the device to the edge.

Learning Phase: In this phase, the edge-server receives PWM, PBM, and SBM logs from the device and builds a single 3-layered baseline profile for the device. The PWM logs are used to build the baseline process white list for the device. The data collected from the PBM and PWM logs in the learning stage combined with pre-recorded malicious data serves as the training dataset for these modules. Using these training sets, device specific binary classifiers are built for both PBM and SBM modules. We discuss in detail our design choice of binary classifiers over unary classifiers in Section 4.

Training Set Creation and On-the-Fly Classifier Training: In order to train the binary classifiers for PBM and SBM modules for each new device, we require both benign and malicious labeled logs in our training set. In our threat model [29], we assume that the device is uncompromised during the learning stage. In practice, one can also consider the scenario where device vendors can perform the learning phase in their isolated environments and provide device profiles to their users. So, all logs obtained during the learning stage can also be assumed to be benign. In order to obtain malicious labeled logs, we emulated different CPU architectures and firmwares using qemu [4] system level emulation and [10]. On these device-specific virtual machines, we ran a portion of the IoT malware samples and collected the device logs for various *interval* values from all malicious processes spawned for different CPU architectures. The generation of maliciously labeled logs

Table 2. Metrics for the *SBM* Module

Metric #	Metric Name #.1	Metric Name #.2	Metric Name #.3 (seconds)	Metric Name #.4 (usecs/call)
1	No. of connect calls	%Time of connect calls	Time taken by connect calls	Time/call of connect calls
2	No. of _newselect calls	%Time of _newselect calls	Time taken by _newselect calls	Time/call of _newselect calls
3	No. of close calls	%Time of close calls	Time taken by close calls	Time/call of close calls
4	No. of nanosleep calls	%Time of nanosleep calls	Time taken by nanosleep calls	Time/call of nanosleep calls
5	No. of fcntl calls	%Time of fcntl calls	Time taken by fcntl calls	Time/call of fcntl calls
6	No. of socket calls	%Time of socket calls	Time taken by socket calls	Time/call of socket calls
7	No. of rt_sigprocmask calls	%Time of rt_sigprocmask calls	Time taken by rt_sigprocmask calls	Time/call of rt_sigprocmask calls
8	No. of getsockopt calls	%Time of getsockopt calls	Time taken by getsockopt calls	Time/call of getsockopt calls
9	No. of read calls	%Time of read calls	Time taken by read calls	Time/call of read calls
10	No. of open calls	%Time of open calls	Time taken by open calls	Time/call of open calls
11	No. of execve calls	%Time of execve calls	Time taken by execve calls	Time/call of execve calls
12	No. of chdir calls	%Time of chdir calls	Time taken by chdir calls	Time/call of chdir calls
13	No. of access calls	%Time of access calls	Time taken by access calls	Time/call of access calls
14	No. of brk calls	%Time of brk calls	Time taken by brk calls	Time/call of brk calls
15	No. of ioctl calls	%Time of ioctl calls	Time taken by ioctl calls	Time/call of ioctl calls
16	No. of setsid calls	%Time of setsid calls	Time taken by setsid calls	Time/call of setsid calls
17	No. of munmap calls	%Time of munmap calls	Time taken by munmap calls	Time/call of munmap calls
18	No. of wait calls	%Time of wait calls	Time taken by wait calls	Time/call of wait calls
19	No. of clone calls	%Time of clone calls	Time taken by clone calls	Time/call of clone calls
20	No. of uname calls	%Time of uname calls	Time taken by uname calls	Time/call of uname calls
21	No. of mprotect calls	%Time of mprotect calls	Time taken by mprotect calls	Time/call of mprotect calls
22	No. of prctl calls	%Time of prctl calls	Time taken by prctl calls	Time/call of prctl calls
23	No. of rt_sigaction calls	%Time of rt_sigaction calls	Time taken by rt_sigaction calls	Time/call of rt_sigaction calls
24	No. of ugetrlimit calls	%Time of ugetrlimit calls	Time taken by ugetrlimit calls	Time/call of ugetrlimit calls
25	No. of mmap2 calls	%Time of mmap2 calls	Time taken by mmap2 calls	Time/call of mmap2 calls
26	No. of fstat calls	%Time of fstat calls	Time taken by fstat calls	Time/call of fstat calls
27	No. of getuid calls	%Time of getuid calls	Time taken by getuid calls	Time/call of getuid calls
28	No. of getgid calls	%Time of getgid calls	Time taken by getgid calls	Time/call of getgid calls
29	No. of geteuid calls	%Time of geteuid calls	Time taken by geteuid calls	Time/call of geteuid calls
30	No. of getegid calls	%Time of getegid calls	Time taken by getegid calls	Time/call of getegid calls
31	No. of madvise calls	%Time of madvise calls	Time taken by madvise calls	Time/call of madvise calls
32	No. of set_thread_area calls	%Time of set_thread_area calls	Time taken by set_thread_area calls	Time/call of set_thread_area calls
33	No. of get_tid_address calls	%Time of get_tid_address calls	Time taken by get_tid_address calls	Time/call of get_tid_address calls
34	No. of prlimit calls	%Time of prlimit calls	Time taken by prlimit calls	Time/call of prlimit calls

is done offline, that is, before the initialization phase, in order to create pre-recorded malicious logs for different CPU architectures, endianess and *intervals*. We store these in the malicious log database. During the learning stage, the training set assimilator creates device-specific training data by combining benign labeled logs from the learning phase and malicious labeled logs from the the malicious log database according to the device CPU architecture and *interval*. Our binary classifier is then trained during the learning stage to distinguish between malicious and benign logs. These classifier models for the PBM and the SBM modules along with the running process whitelist for the PWM module are combined together to form the complete behavioral baseline profile for the device. In our current implementation, we choose a random forest classifier for both

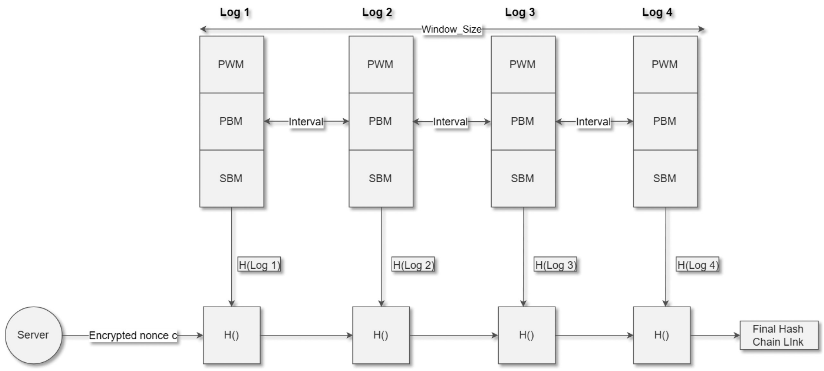

Fig. 2. Hash Chain Verifier.

the PBM and SBM modules as it provides the highest detection efficiency. We discuss more about the performance of various binary classifiers in Section 4.

Operational Phase: Once the baseline profiles for all modules are built at the edge, the operational phase starts. In this phase, the device can operate as desired and we assume the attacker has full access to the device. The device continuously records and transfers logs and corresponding hashes to the edge-server according to the configuration values of *interval* and *windowsize*.

The device constantly records logs for each module every *interval* seconds and transfers them to the edge every *windowsize* seconds. Overall, every transfer contains *windowsize/interval* number of PWM, PBM, and SBM logs. We discuss about the choice of values of *windowsize* and *interval* and how it impacts the performance of `E-Spion` in Section 4.

Anomaly Detection Stage: For each of the logs received, the hash-chain-verifier at the network edge first checks the integrity of the logs. If the logs fail the integrity check or no logs are received, then the device is considered compromised or malfunctioning and the IDS raises an alert. We can find out of the approximate time of infection and the likely source of compromise by analyzing the received logs. If the integrity check goes through, the logs are forwarded to the anomaly detection engine at the edge-server. Here, the logs are compared with the baseline profiles for each of the three modules and, in case an anomaly is detected, an alert is raised. The PWM module simply compares the current running process list to the PWM whitelist, while the PBM and SBM modules classify the current device logs using the binary classifiers trained in the learning stage.

3.4 Hash Chain Verifier

This component verifies the integrity of the logs received by the edge-server. It maintains hash chains of the logs as shown in Figure 2. For the window of time *windowsize* as defined before, the client generates a hash chain of the device logs during this window of time. We use the SHA256sum utility [1] to compute the SHA-256 one-way hashes of the logs. The device logs include the running process lists, process behavior logs, and system call behavior logs collected for the PWM, PBM, and SBM, respectively. A new hash chain link is added after every interval specified by *interval*. So, each window contains a hash chain made of *windowsize*/*interval* hash links. We use the Merkle-Damgard construction [14] to compute these hash links. So, at the start of every hash chain initiation (every *windowsize* seconds), the server sends an encrypted random nonce c to the device.

The hash list uses c as an initialization vector for the hash chain. This value is deleted as soon as the first hash is created. The hash value and the log are stored. After *interval* seconds, the second link of the hash chain is computed using the hash of the previous log and the hash of the current

Table 3. Performance Evaluation for Various PBM Module Classifiers

Classifier	Accuracy (%)	ROC Area	True Positive Rate	False Positive Rate	Precision	Recall	F1-Measure
Naive Bayes	26.24	0.82	0.26	0.15	0.84	0.26	0.23
Logistic Regression	85.72	0.84	0.85	0.66	0.83	0.85	0.82
Decision Tree	97.64	0.94	0.97	0.12	0.97	0.97	0.97
Random Forest	97.75	0.98	0.97	0.10	0.97	0.97	0.97
K-NN	97.66	0.97	0.97	0.10	0.97	0.97	0.97
ANN	95.26	0.89	0.95	0.24	0.95	0.95	0.94

device log. After which the hash of the previous log is deleted from the system. This is done every *interval* seconds for the duration of the window. After the window closes (*windowsize* seconds), the final hash value along with the corresponding files (device logs) in the *windowsize* are sent to the edge-server. The edge-server then computes the final hash from the received logs (and random nonce c) and compares it with the hash value received from the device. If the values do not match, the authentication fails. We do not go into details about the security of the hash chain verifier, and refer the reader to our previous paper for details on the threat model and security analysis [29].

4 IMPLEMENTATION DETAILS

In this section, we discuss key challenges in the implementation of the prototype of our system and how we have addressed these challenges, namely: deployability, choice of classifiers, timing/interval choices, modularity, and distributed nature of logs.

Deployability. Due to the heterogeneity and resource constrained nature of IoT devices, deployment of a system-level IDS like ours is a challenge, and as with all host-based IDSs, it is the limiting factor. We implement our system with the goal to overcome this challenge and make sure that deployment of the system is feasible for all IoT devices. We observed that 71.3% of all IoT devices run some version of Linux as their operating system and "Linux is becoming the standard OS for all gateway and resource constrained devices" according to the 2017 IoT developer survey [19]. So in order to maximize deployment, we build our client side (SysMon) modules using common system tools like atop, ps, sha256sum, openssh, and strace present on all standard Linux distributions. We tested our system on various CPU architectures and Linux operating systems along with several IoT device emulations using Firmadyne [10] in order to make the device modules scalable and easy to deploy. It must be noted that similar tools are available on other OSes and `E-Spion` variants for them can be implemented along similar lines.

Choice of Classifiers. The common approach for detecting anomalies involves using unary or one-class classifiers. However, IoT devices are prone to malfunctions and fluctuations in performance [23]. This results in one-class classifiers classifying benign malfunctions and fluctuations as malicious activities, thus causing high numbers of false positives. To avoid this problem, we use binary classifiers and train our classifiers using existing IoT malware samples.

For both the PBM and SBM module, we tested different binary classifiers like naive Bayes, decision tree, logistic regression, random forest, K nearest neighbor (K-NN) and multi-layer perceptron (feed forward artificial neural network (ANN)) classifiers. We found that the random forest classifier works best for detecting intrusions for our dataset as shown in Tables 3 and 4. Also, tree-based classifiers are the fastest and most efficient to build. So, we use the random forest classifiers for both our PBM and SBM modules.

The low accuracy (~26%) of the naive Bayes classifier in the PBM module can be attributed to its class independence property. From our observation, IoT malware usually employs a combination of

Table 4. Performance Evaluation for Various SBM Module Classifiers

Classifier	Accuracy (%)	ROC Area	True Positive Rate	False Positive Rate	Precision	Recall	F1-Measure
Naive Bayes	100.00	1	1	0.0	1	1	1
Logistic Regression	99.74	0.99	0.99	0.002	0.99	0.99	0.99
Decision Tree	99.94	0.99	0.99	0.001	0.99	0.99	0.99
Random Forest	100.00	1	1	0.0	1	1	1
K-NN	99.98	1	1	0.0	1	1	1
ANN	99.94	1	0.99	0.001	0.99	0.99	0.99

CPU, memory and disk resources; thus, the usages of these resources are correlated to each other. Therefore, assuming that these features are independent of each other leads to a lower accuracy for the PBM module. However, for the SBM module, the naive Bayes classifier works perfectly. The reason is that the system calls made by a process are not directly correlated to other system calls made by the device, so the class independence property is a fair assumption in the case of the SBM module. We find that all the binary classifiers give high detection rates for the SBM module. This can be attributed to the larger number of good features (that is, 136) used for training our SBM classifier.

The regression classifier works better than the naive Bayes classifier with an accuracy of ~85% for the PBM module. A logistic regression model searches for a single linear decision boundary in the feature space. Hence, the lower accuracy can be attributed to the fact that our data does not have a completely linear boundary for decisions. Therefore, the ideal decision boundary for our dataset would be non-linear. We observe that tree-based classifiers (decision trees and random forests) work best for our system. However, decision trees are prone to over-fitting which is the reason we choose a random forest classifier over a decision tree classifier. Although the distance-based K-NN classifier and the deep-learning-based ANN classifer give comparable results to tree-based classifiers, they take more time and computational power to build which is unsuitable for the real-time requirements of IoT networks.

Timing/Interval Choices. The configuration values for the variables *windowsize* and *interval* play a significant role in the detection efficiency and overhead costs of our system. We performed several tests with different configuration values on different devices in our testbed (see Section 5.2). We tested our system with *windowsize* of 20, 50, 100, 500, and 1,000 seconds and *interval* of 2, 10, and 20 seconds. We found that for all devices, the detection accuracy remains nearly constant (75%, 97%, 99% for PWM, PBM, SBM, respectively) for all these configurations. This can be attributed to the fact that the devices behave similarly in all these intervals and the choice of the recording intervals has minimal impact on the detection accuracy for our current dataset. However, the larger the *interval* size, the higher chance the attacker has of evading the system as discussed in [29]. Also, our system can only detect attacks after *windowsize* seconds when logs are transferred to the edge-server. If the *windowsize* is too high, then the detection time of the attack will also be higher. Large-scale time-critical networks, like vehicular networks, smart-grids, and the like, require very short attack detection time because of their real-time safety requirements. So in such networks, the *windowsize* should be small enough to detect these attacks in real-time. In terms of overhead costs, a lower *windowsize* results in higher communication overhead at the device as the logs need to be transferred more frequently. A lower *interval* results in a higher computational overhead at the device mainly because more hashing operations are required. Such computational and communication overheads can be detrimental in scenarios where resource constrained embedded devices are present in the network. So, we leave the choice of the optimal values of these configuration parameters to the system administrator as it depends on the system requirements, devices used, deployment scenario, and associated risk for each device. In our current prototype, we use a *windowsize* of 100 seconds and *interval* of 10 seconds for all the devices.

Modularity of Logs. Our system has three detection modules that provide varying degrees of detection efficiency and overhead costs. We deploy these modules independently in separate threads rather than sequentially. The outputs from these modules are combined by the module manager to provide a 3-layered detection output rather than a simple alert. This implementation is essential to weed out false positives and differentiate between benign malfunctions and attacks. For example, in case of a device malfunction, the PBM module will raise an alert due to the anomalous process behavior. However, the SBM module will not raise an alert as no anomalous system calls were issued by the process. These modular logs provide the module manager enough information to classify this incident as a malfunction rather than an attack and thus reduce false positives.

The modules have different overhead costs associated with them as well. The modular design allows activating/deactivating each module as required by the device according to its resources, associated risk, network conditions etc. For example, a resource constrained device can be configured to activate the expensive SBM module only when an alert is received from either PWM/PBM modules. On the other hand, a device with a high associated risk will be configured to have all modules active at all times. The modular design enhances flexibility and scalability in the deployment of our system.

Distributed Nature of Logs. In our current prototype, we store the learnt behavior from the learning stage for each device on the edge and we assume that the device functions benignly until the end of the learning phase. This assumption holds in our current threat model but would be a limitation for real world scenarios where devices are compromised as soon as they connect to the network or are compromised in production. To address this challenge, we added some additional functionality in our current prototype to move further towards a fog computing paradigm [6]. The edge-server stores the logs in a cloud repository accessible by other edge-servers. This repository holds learnt logs from devices across different `E-Spion` enabled networks. Such logs enable one to compare behaviors of the same devices in different networks and to detect anomalous behavior during the learning stage. A device which has already been compromised during the learning stage shows significantly different behavior when compared to the same device in another network which has not been compromised during the learning stage. This distributed nature of logs allows inter-operation between different `E-Spion` edge-servers and a fail-check in case of devices behaving maliciously during the learning phase.

5 EVALUATION

We perform an extensive evaluation of `E-Spion` on a typical enterprise IoT setup with 3,973 of the most recent IoT malware samples. In this section, we provide details on our malware dataset and evaluation testbed. We then discuss the detection efficiency of our system and finally provide an analysis of the malware samples in terms of our device logs. After which we discuss the effectiveness of our system against eight types of file-less attacks and finally look at the overhead costs associated with our system. We focus on collecting IoT malware extensively rather than simulating various network-based attacks as seen in prior work. The reason for this is that the goal of our host-based system is detecting the compromised host/device during the injection or infection stage rather than the attack stage. More details on our threat model are given in our previous paper [29].

5.1 IoT Malware

IoT malware is downloaded during the infection stage according to the device operating system and architecture. For evaluating our system, we collected different variants of IoT malware and built a comprehensive dataset using 3,973 malware samples from the most popular malware

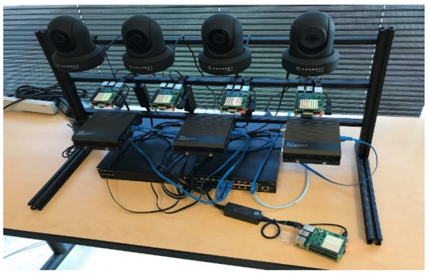

Fig. 3. Experiment Testbed.

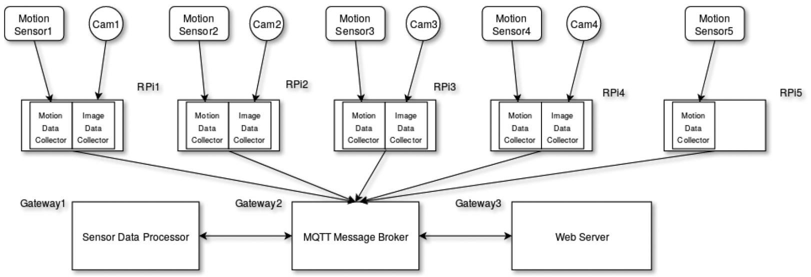

Fig. 4. Testbed Implementation Details.

families: Zorro, Gayfgt, Mirai, Hajime, IoTReaper, Bashlite, nttpd, linux.wifatch etc. The malware samples were collected from IoTPOT [32], VirusTotal [39], and Open Malware [27]. These malware executables are compiled for different CPU architectures and endianess. The collected malware executables are classified according to different device architectures in Table A.2. 2,572 of the samples are compiled for little endian processors while 1,421 of them are for big endian processors.

We have used 20% (795) of our malware samples as training malware samples for the learning stage and 80% (3,178) of them as testing data for evaluation in our experimental setup.

5.2 Evaluation Testbed

We create a typical motion sensing network using 4 webcams, 5 raspberry pi devices (4 mounted and 1 tethered), 3 HPE GL10 IoT gateways, and 1 Aruba PoE Switch (see Figure 3).

The raspberry pi devices are responsible for recording images from the cameras and movement from the sensors. The motion sensor is an accelerometer installed on the raspberry pi device and is responsible for detecting any movement of the raspberry pies. The devices communicate with each other using MQTT (Message Queuing Telemetry Transport). A high-level outline of the functions of the devices in the testbed can be seen in Figure 4.

5.3 Detection Efficiency and Analysis

To test our system, we manually download and run 3,178 malware executables on the devices in our testbed. We evaluate the performance of our system by running the malware samples sequentially. Every time a malware sample is executed, we check if our system is able to correctly flag each malicious process spawned by the malware. After every run, we restore the system with a clean OS and execute the next malware sample. In what follows, we discuss the detection accuracy for each layer of our system.

PWM Layer. The PWM layer had a detection rate of 79.09%. We see that the simplistic PWM is able to detect most of the IoT malware samples just through the simple whitelisting of process names. This reinforces our claim that most of the IoT malware is basic and does not employ any obfuscation or deception techniques. On average, each malware spawns a mean of 1.79, median of 1 and mode of 2 new processes. 20.91% of the malware spawn no new processes but rather manipulate or masquerade as a benign process (e.g., whitelisted process). We observe that most malware invokes the prctl system call and uses the PR_SET_NAME request. We also observed that some malware simply change the name of the malicious program to a benign one. Most of the malware masquerades as common system utilities such as sshd, telnetd etc. Another observation we made from the PWM layer is that some of the malware samples produce a randomly generated process name for each execution of the sample. This would seem an appropriate approach taken by attackers to bypass process name blacklisting approaches.

Another key point to note is that we do not see any false positives from the PWM layer. This can be attributed to the fact that all the processes spawned by benign applications are already whitelisted during the learning stage and none of the benign applications spawn any new un-whitelisted processes during the operation stage.

PBM Layer. The PBM layer has a high detection rate of 97.02% but at the same time a significant false positive rate of 2.97%. As we are monitoring for anomalous activity for all the running processes of the system, the PBM layer is able to capture malware masquerading as benign processes which the PWM layer is unable to detect. As this is a machine learning predictive approach, we do encounter false positives in this layer and some benign processes are incorrectly classified as malicious. We observe that most malware is very aggressive in terms of CPU and memory usage when it infects the system. Therefore, they can be easily detected by the PBM as this module is able to quickly detect anomalous behavior even for malware masquerading as benign processes.

To demonstrate the effectiveness of each metric/feature in our PBM module, we compare baseline logs against the malicious logs over time according to average CPU usage (syscpu, usrcpu), average memory sage (Vgrow, Rgrow) and average disk usage (wrdsk, rddsk) in Figure 5. The malicious logs are represented in red while the benign logs are represented in blue. We observe that the malicious logs are clearly distinguishable from the baseline logs using just a subset of the metrics.This means that these features individually (CPU, memory and disk usage) also perform well in terms of detection efficiency. So, in devices where one of the metrics is inaccessible or unavailable (for example, small embedded devices which have no disk but just flash memory), the PBM module will still be able to detect intrusions effectively. We observe that most of the malware has a typical *bursty* behavior pattern in that it remains dormant most of the time and performs its malicious activities in a burst. Benign applications on the other hand have a *consistent* behavior where their operations are periodic and constant. These results confirm our original intuition about the periodic and consistent nature of IoT device processes. We also train our PBM classifier using the CPU, memory and disk usage metrics individually and observe detection accuracy of 94.2%, 96.67%, and 91.46%, respectively. Although using the metrics individually has high detection efficiency, it also results in higher false positive rates of 5.77%, 3.32%, and 8.5% respectively. So, we use all the metrics in conjunction to minimize the number of false positives.

SBM Layer. The SBM layer has a detection rate of 100% and 0 false positives. This is the most fine grained detection module. However, it is also the most expensive. Table 5 shows a comparison between malicious and benign SBM logs according to the different system call types in terms of average number of calls made and average % of execution time taken by the system call. We can see that malicious processes use a typical combination of system calls, like connect, socket, read, write, munmap, ioctl, etc., and the behavior is highly different from benign processes.

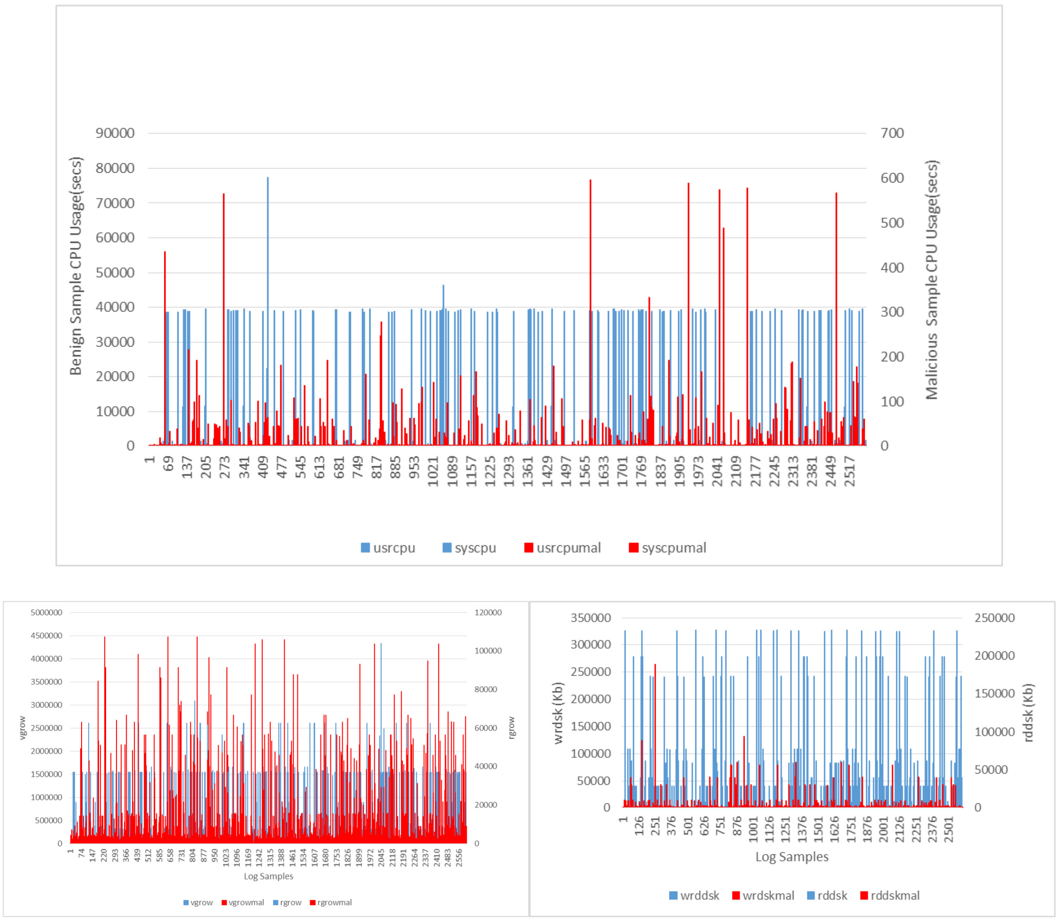

Fig. 5. Comparison between malicious vs baseline PBM log samples over time according to (a) CPU usage (usrcpu, syscpu), (b) Memory Usage (vgrow, rgrow), and (c) Disk Usage (wrdsk, rddsk).

5.4 Effectiveness Against File-Less Attacks

As seen before, we have categorized file-less attacks into eight categories. We manually performed all 8 types of attacks in our `E-Spion` monitored environment. We observe that our system is able to effectively detect all 8 types of the attacks in our evaluation testbed. The reason being that all file-less attacks propagate using either shell or system commands like passwd, rm, kill etc. on the device using root privileges. Whenever a command is issued in Unix/Linux, it creates/starts a new process. This new process is flagged by the PWM layer. Only the Type 8 attack manages to evade the PWM layer when it it uses the ssh utility to set up the port forwarding. This however is flagged by the PBM layer as it detects an anomalous use of the whitelisted ssh processes. The SBM layer also detects the anomalous use of socket, bind system calls made by the SSH processes and flags the Type 8 attacks. Even more sophisticated file-less attacks which employ benign application-based commands for infection would be detected by our PBM and SBM modules as the application's "behavior" is "anomalous" or different from the learnt baselines.

Table 5. Comparison of Malicious and Baseline Logs of the SBM Module

System Call Type	Malicious Process: Avg No of calls	Benign Process: Avg No of calls	Malicious Process: Avg Time%	Benign Process: Avg %Time
connect	10	1	29.375431	0.109712824
_newselect	6	0	7.652097	0
close	882	527	0.79684114	2.8245006
nanosleep	5	409	3.5325627	44.9521
fcntl64	53	34	0.07268107	0.3070208
socket	7	1	1.5146441	0.06425141
ra_sigprocmask	0	2	0	0.009006577
getsockopt	6	0	0.77036035	0
read	2868	984	2.2888484	7.1284814
open	22	1704	0.16306427	13.151322
execve	1	1	3.61E-05	0.0042437743
chdir	1	0	7.06E-04	0
access	11	79	0.066558525	0.848794
brk	3	33	0.005014777	0.19349128
ioctl	3464	498	0.52835435	12.714709
setsid	1	0	0.008818369	0
munmap	2769	18	13.151226	0.5787933
wait4	1	0	12.624713	0.3850142
clone	1	2	0.018954737	0.024077073
uname	1	0	0.0037215038	0.0037744138
mprotect	13	232	0.07808309	1.8957471
prctl	1	0	0.022474075	0
rt_sigaction	12	85	0.20787959	0.2348594
ugetrlimit	1	1	5.62E-04	0.003911133
mmap2	2801	291	12.589997	3.5121188
fstat64	24	656	0.026696749	2.4766097
getuid32	1	3	0.008650763	0.044761505
getgid32	1	1	0.007634516	0.026103668
geteuid32	1	2	0.0027905148	0.028454894
getegid32	1	0	0.0021606325	0.015322265
madvise	1	1	2.76E-04	0.19356513
gettid	1	0	6.24E-04	0
set_thread_area	1	0	3.96E-04	0
set_tid_address	1	0	4.08E-04	0.0020117187

5.5 Overhead Analysis

We now discuss the performance and storage overheads for the different modules of `E-Spion`. An important requirement of our design is that the system should impose minimal costs on the already resource constrained IoT devices. As we see in Table 6, most of the computational resources (CPU and memory usage) are required by the server side modules which run on the more powerful edge-server while the client side modules running on the IoT device require minimal computational resources. Table 6 also shows the overheads in terms of CPU, memory and disk usage required by the different modules on the device. We observe that the SBM module is the most computationally

Table 6. Overhead Analysis of the Modules of E-Spion

Module	Device			Edge-Server Side	
	CPU Usage(%)	Memory Usage(Kb)	Disk Usage(Kb)	CPU Usage(%)	Memory Usage(Kb)
PWM	0.01	2896	67	23	74,684
PBM	0.2	4788	776	24	739,936
SBM	0.6	3912	1340	25	363,680

Table 7. Comparison of Different Modules of E-Spion

Module	Accuracy	Computational Cost on Device	Storage Cost on Device	Slow-Down of benign applications	Computational Cost on edge-server	False Positives
PWM	Moderate	Low	Low	No	Low	None
PBM	High	High	High	No	High	Moderate
SBM	Highest	High	High	Yes	Moderate	None

expensive while the PWM is the least expensive. The PWM module requires the least amount of CPU, memory and disk on the device. In terms of CPU usage and disk usage, the SBM module is the most expensive. It is important to note that the SBM also slows down benign processes because it uses the system tool *strace*. *strace* pauses the process twice for each syscall, and executes context-switches each time between the process and *strace*.

On the edge-server side, the PBM module is the most expensive in terms of memory and PWM is the least expensive. The PBM module requires the most amount of memory for extracting features because the process behaviour logs are denser than both process whitelists (PWM) and system call behaviour logs (SBM). As the SBM module only checks for certain system calls and records a summary of these calls in its logs, extraction of features from the logs is considerably less expensive than PBM. Both SBM and PBM modules require training and operating expensive random forest binary classifiers due to which they are 5x and 10x more expensive respectively than the PWM module in terms of memory usage. In terms of CPU usage, all three modules have similar overhead costs on the server side.

We see that the three modules have varying degrees of computational and storage overheads and detection efficiency. Finally, in Table 7, we summarize our comparisons of each module in terms of accuracy and overhead.

6 RELATED WORK

Intrusion Detection for IoT. IDSes for IoT devices use different strategies for placing intrusion detection tools, namely: centralized, distributed, and hybrid. IDSes like [11, 18, 41] use the centralized IDS placement approach and generally monitor traffic passing through the border routers. Such a strategy has the advantage of detecting attacks from the Internet into or out of the IoT network but this is not enough to detect attacks involving just the nodes of the IoT network. Also, if part of the network is disrupted, a centralized IDS might not be able to monitor all the nodes. Lightweight distributed placement strategies have been proposed in [8, 24, 31], where each node is responsible for monitoring and analyzing its packet payloads, energy consumption, and its neighbors, respectively. However, these strategies impose a non-negligible computation overhead on the already resource constrained devices. Most recent IDSes [34, 35, 38] are hybrid approaches which combine centralized and distributed approaches. The Kalis IDS [28] has been designed with a flexible placement strategy, in that it can be placed on the devices, or at some gateway, or onto its own specialized device. Our system also uses a hybrid placement strategy where the data gathering module runs on the device while the analysis module runs on the edge-server.

Most existing IDSes for IoT devices and embedded devices, like [21, 25, 31], use signature-based detection schemes. These IDSes rely on network information gathered by a packet sniffer, and detect attacks using signature matching over this information. The approaches that only use signature-based detection are simpler to develop but cannot detect attacks for which the signature is unavailable. Also, it is difficult for resource-constrained IoT devices to run computationally expensive signature storing and matching schemes. Another factor is that with the high heterogeneity in terms of protocols, functionality, manufacturers, device architectures, and so on, the attack signatures/rule list becomes very large and complicated. While running through a large signature list is sustainable for a traditional network, small IoT networks incur heavy overhead and this results in poor performance of the IDS. Also, going through a large signature list usually results in a higher number of false positives. Conversely, anomaly-based techniques are more versatile, as they can detect unknown attacks, but are harder to implement and more inaccurate, potentially yielding high false positive rates. A number of such anomaly-based detection schemes schemes have been proposed [11, 18, 24, 34, 38] which rely on detecting anomalies by inspecting packet rates, sizes, payloads, headers, node connections, energy consumption, device profiles, and the like. Our system is also an anomaly-based detection scheme but focuses on building device profiles using system information gained from the running processes and system calls rather than network information.

This current article extends our previous work [29] by introducing modifications to the IDS implementation and reporting evaluation results concerning the effectiveness of the IDS against recent sophisticated file-less attacks. Overall, our approach is significantly different from all the previous approaches in the area of IDS for IoT systems as we aim to build a hybrid lightweight IDS system which is able to detect anomalous behavior in terms of system level information from running processes and system calls.

Traditional System Level Techniques. Even though there is little research on system-level IDS technology for IoT, there has been significant research in the area of intrusion detection using system events and provenance logging for traditional computer systems. There are two main approaches. The first approach is based on system event logging and then causally connecting these events during attack investigations to build causal graphs like Auditd [9] in Linux kernels which maintains audit logs of important system events. The other approach is to use provenance propagation like in PASS [30] which stores and maintains provenance data where provenance is calculated for certain entities like network sessions after which the IDS captures the program dependencies during execution. There have been attempts to build such schemes for distributed systems [16]. ProTracer [26] proposes a lightweight provenance "tainting" scheme which is a hybrid of both those approaches. The most comprehensive of such hybrid IDSes is RAIN [20] which achieves higher run-time efficiency by pruning out unrelated executions in their provenance graphs. Although these approaches have similar ideas compared to ours, they are not suitable for IoT devices. The implicit assumption made in these approaches is that each input event has a causal effect on all the output events. These systems then try to build a model of the behavior of all the benign applications using this assumption. Such an approach works well for complex applications running on traditional computer systems as there are large numbers of running processes and threads interacting with each other on these systems. However, this model would be an overkill for IoT devices. Most IoT devices have much simpler program execution and a simpler approach to modeling IoT devices is required. The other major factor to consider is that these approaches incur large computational and storage costs which are infeasible for resource-constrained IoT devices. By contrast, our system uses a simpler and more efficient 3-layered approach to model IoT device behavior using system data. Also, we leverage network edge-servers and thus are able to minimize the workload on the devices. This makes our approach practical and cost-efficient for IoT devices.

7 CONCLUSIONS AND FUTURE WORK

In this article, we have proposed a system-level IDS E-Spion tailored for IoT devices. It builds a 3-layered baseline profile with varying overhead costs for IoT devices using system information and detects intrusions according to anomalous behavior. It is specifically designed for resource-constrained IoT devices with an efficient device-edge split architecture and modular 3-layered design. We extensively tested our system with a comprehensive set of 3,973 IoT malware samples and 8 types of file-less attacks. We observed a detection rate of over 78%, 97%, and 99% for our 3 layers of detection, respectively. As part of future work, we intend to broaden our device logs by including network logs of the device by integrating our system with network-based IDSs like Kalis [28] and Heimdall [18]. The goal is to provide a more fine-grained detection and build a more comprehensive device behavior profiles.

APPENDIX

A SUPPLEMENTAL TABLES

Table A.1. System Calls Monitored by *SBM*

System Call	Description
connect	initiate a connection on a socket
_newselect	synchronous I/O multiplexing
close	close a file descriptor
nanosleep	high-resolution sleep
fcntl64	manipulate file descriptor
socket	create an endpoint socket for communication
rt_sigprocmask	examine and change blocked signals
getsockopt	get and set options on sockets
read	read from a file descriptor
open	open and possibly create a file
execve	execute program
chdir	change working directory
access	check user's permissions for a file
brk	change data segment size
ioctl	manipulates the underlying device parameters of special files
setsid	creates a session and sets the process group ID
munmap	map or unmap files or devices into memory
wait64	wait for process to change state
clone	create a child process
uname	get name and information about current kernel
mprotect	set protection on a region of memory
prctl	operations on a process
rt_sigaction	examine and change a signal action
ugetrlimit	get/set resource limits

(Continued)

Table A.1. Continued

System Call	Description
mmap2	map or unmap files or devices into memory
fstat64	get file status
getuid32	returns the real user ID of the calling process
getgid32	returns the real group ID of the calling process.
geteuid32	returns the effective user ID of the calling process.
getegid32	returns the effective group ID of the calling process.
madvise	give advice about use of memory
set_thread_area	set a GDT entry for thread-local storage
get_tid_address	set pointer to thread ID
prlimit64	get/set resource limits

Table A.2. Malware Executables Breakdown according to CPU Architecture

Architecture	Number of Malware samples
Mips	935
Arm	912
x86	576
Sparc	299
Renesas/SuperH	310
x86_x64	294
MC68000	294
PowerPC	353
Other	26

REFERENCES

[1] [n.d.]. https://linux.die.net/man/1/sha256sum.

[2] Fahd Al-Haidari, M. Sqalli, and Khaled Salah. 2013. Impact of CPU utilization thresholds and scaling size on autoscaling cloud resources. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 2. IEEE, 256–261.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. 1093–1110.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[5] Elisa Bertino. 2016. Data security and privacy in the IoT. In *19th International Conference on Extending Database Technology (EDBT 2016)*. OpenProceedings.org, 1–3.

[6] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. ACM, 13–16.

[7] Prasad Calyam, Sudharsan Rajagopalan, Sripriya Seetharam, Arunprasath Selvadhurai, Khaled Salah, and Rajiv Ramnath. 2014. VDC-Analyst: Design and verification of virtual desktop cloud resource allocations. *Computer Networks* 68 (2014), 110–122.

[8] Christian Cervantes, Diego Poplade, Michele Nogueira, and Aldri Santos. 2015. Detection of sinkhole attacks for supporting secure routing on 6LoWPAN for Internet of Things. In *IM*. 606–611.

[9] Timothy R. Chavez. 2006. A look at Linux audit. In *LinuxWorld Conference and Expo (Boston, MA, April)*.

[10] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for Linux-based embedded firmware. In *NDSS*.
[11] Eung Jun Cho, Jin Ho Kim, and Choong Seon Hong. 2009. Attack model and detection scheme for Botnet on 6LoWPAN. In *Asia-Pacific Network Operations and Management Symposium*. Springer, 515–518.
[12] Catalin Cimpanu. 2018. New Hakai IoT botnet takes aim at D-Link, Huawei, and Realtek routers. https://www.zdnet.com/article/new-hakai-iot-botnet-takes-aim-at-d-link-huawei-and-realtek-routers/
[13] Stephen Cobb. 2017. RoT: Ransomware of Things. ESET.
[14] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. 2005. Merkle-Damgård revisited: How to construct a hash function. In *Annual International Cryptology Conference*. Springer, 430–448.
[15] Fan Dang, Zhenhua Li, Yunhao Liu, Ennan Zhai, Qi Alfred Chen, Tianyin Xu, Yan Chen, and Jingyu Yang. 2019. Understanding fileless attacks on Linux-based IoT devices with honeycloud. In *17th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 482–493.
[16] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for provenance auditing in distributed environments. In *13th International Middleware Conference*. Springer-Verlag New York, Inc., 101–120.
[17] Andy Greenberg. 2017. The Reaper Botnet Has Already Infected a Million Networks. https://www.wired.com/story/reaper-iot-botnet-infected-million-networks/.
[18] Javid Habibi, Daniele Midi, Anand Mudgerikar, and Elisa Bertino. 2017. Heimdall: Mitigating the Internet of insecure things. *IEEE Internet of Things Journal* 4, 4 (2017), 968–978.
[19] Christine Hall. 2018. Survey Shows Linux the Top Operating System for Internet of Things Devices. https://www.itprotoday.com/iot/survey-shows-linux-top-operating-system-internet-things-devices.
[20] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 377–390.
[21] Prabhakaran Kasinathan, Gianfranco Costamagna, Hussein Khaleel, Claudio Pastrone, and Maurizio A. Spirito. 2013. An IDS framework for Internet of Things empowered by 6LoWPAN. In *2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, 1337–1340.
[22] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. 2017. DDoS in the IoT: Mirai and other botnets. *Computer* 50, 7 (2017), 80–84.
[23] In Lee and Kyoochun Lee. 2015. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. *Business Horizons* 58, 4 (2015), 431–440.
[24] Tsung-Han Lee, Chih-Hao Wen, Lin-Huang Chang, Hung-Shiou Chiang, and Ming-Chun Hsieh. 2014. A lightweight intrusion detection scheme based on energy consumption analysis in 6LowPAN. In *Advanced Technologies, Embedded and Multimedia for Human-centric Computing*. Springer, 1205–1213.
[25] Caiming Liu, Jin Yang, Run Chen, Yan Zhang, and Jinquan Zeng. 2011. Research on immunity-based intrusion detection technology for the Internet of Things. In *2011 7th International Conference on Natural Computation*, Vol. 1. IEEE, 212–216.
[26] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*.
[27] Open Malware. [n.d.]. http://openmalware.org.
[28] Daniele Midi, Antonino Rullo, Anand Mudgerikar, and Elisa Bertino. 2017. Kalis—A system for knowledge-driven adaptable intrusion detection for the Internet of Things. In *IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. IEEE, 656–666.
[29] Anand Mudgerikar, Puneet Sharma, and Elisa Bertino. 2019. E-Spion: A system-level intrusion detection system for IoT devices. In *2019 ACM Asia Conference on Computer and Communications Security*. ACM, 493–500.
[30] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. 2006. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*. 43–56.
[31] Doohwan Oh, Deokho Kim, and Won Woo Ro. 2014. A malicious pattern detection engine for embedded security systems in the Internet of Things. *Sensors* 14, 12 (2014), 24188–24211.
[32] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. 2015. IoTPOT: Analysing the rise of IoT compromises. *EMU* 9 (2015), 1.
[33] Vern Paxson. 1999. Bro: A system for detecting network intruders in real-time. *Computer Networks* 31, 23–24 (1999), 2435–2463.
[34] Pavan Pongle and Gurunath Chavan. 2015. Real time intrusion and wormhole attack detection in Internet of Things. *International Journal of Computer Applications* 121, 9 (2015).
[35] Shahid Raza, Linus Wallgren, and Thiemo Voigt. 2013. SVELTE: Real-time intrusion detection in the Internet of Things. *Ad Hoc Networks* 11, 8 (2013), 2661–2674.
[36] Martin Roesch. 1999. Snort: Lightweight intrusion detection for networks. In *Lisa*, Vol. 99, No. 1. 229–238.

[37] Khaled Salah, Jose M Alcaraz Calero, Sherali Zeadally, Sameera Al-Mulla, and Mohammed Alzaabi. 2012. Using cloud computing to implement a security overlay network. *IEEE Security & Privacy* 11, 1 (2012), 44–53.

[38] Nanda Kumar Thanigaivelan, Ethiopia Nigussie, Rajeev Kumar Kanth, Seppo Virtanen, and Jouni Isoaho. 2016. Distributed internal anomaly detection system for Internet-of-Things. In *2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 319–320.

[39] VirusTotal. [n.d.]. https://www.virustotal.com.

[40] Jack Wallen. 2017. Five nightmarish attacks that show the risks of IoT security. https://www.zdnet.com/article/5-nightmarish-attacks-that-show-the-risks-of-iot-security/.

[41] Linus Wallgren, Shahid Raza, and Thiemo Voigt. 2013. Routing attacks and countermeasures in the RPL-based Internet of Things. *International Journal of Distributed Sensor Networks* 9, 8 (2013), 794326.

[42] Jacob Wurm, Khoa Hoang, Orlando Arias, Ahmad-Reza Sadeghi, and Yier Jin. 2016. Security analysis on consumer and industrial IoT devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 519–524.

[43] Shanhe Yi, Zhengrui Qin, and Qun Li. 2015. Security and privacy issues of fog computing: A survey. In *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 685–695.

Received September 2019; revised December 2019; accepted February 2020