E-Spion: A System-Level Intrusion Detection System for IoT Devices

Anand Mudgerikar
amudgeri@purdue.edu
Purdue University

Puneet Sharma
puneet.sharma@hpe.com
HPE

Elisa Bertino
bertino@purdue.edu
Purdue University

ABSTRACT

As the Internet of Things (IoT) grows at a rapid pace, there is a need for an effective and efficient form of security tailored for IoT devices. In this paper, we introduce E-Spion, an anomaly-based system level Intrusion Detection System (IDS) for IoT devices. E-Spion 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. These profiles are then used to detect anomalous behaviors indicative of intrusions. E-Spion provides three layers of detection with increasing detection efficiency but at the same time higher overhead costs on the devices. We have extensively evaluated E-Spion using a comprehensive dataset of 3973 IoT malware samples in our testbed. We observe a detection efficiency ranging from 78% to 100% depending on the layers of detection employed. We provide an analysis and comparison of the different layers of E-Spion in terms of detection accuracy and overhead costs. We also analyze the behavior of the malware samples in terms of our device logs at each layer.

KEYWORDS

Intrusion Detection, IoT Security, Malware

ACM Reference Format:
Anand Mudgerikar, Puneet Sharma, and Elisa Bertino. 2019. E-Spion: A System-Level Intrusion Detection System for IoT Devices. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3321705.3329857

1 INTRODUCTION

System level IDSes like anti-viruses, commonly used for traditional computer systems, employ attack signature based detection. Anomaly based detection in such IDSes is not practical as a traditional computer system runs a number of different kinds of applications which are 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.

This work was initiated at and supported by Hewlett Packard Labs.

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329857

In general, IoT devices have a main 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 etc. 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 used in DVRs, cameras, cars, televisions etc. 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 our device profiles.

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.

For traditional computing systems, there have been attempts to build behavior based detection techniques by monitoring various kinds of system data such as system events [4, 16, 22], system calls [7, 15], process events [10], function calls [25] etc. However, these techniques are difficult to use for IoT devices due to two main reasons: high overhead and low portability. Firstly, as none of these techniques are specifically designed for IoT devices, unlike our IDS E-Spion, they do not consider the periodic and consistent nature of the device's system functions. Rather they assume that each event has a causal effect on all future events. Also, IoT malware has significantly less complex and different behavior than traditional malware [6]. Due to those reasons, these IDSs tend to build complex behavior models for detection which impose high computational and storage overheads on the already resource constrained IoT devices. Secondly, these IDSs are tailored for specific operating systems and architectures like [9] for Android, [30] for Windows, [8] for iOS etc. With the heterogeneity of IoT devices in terms of operating systems and CPU architectures, portability of these IDSs to all IoT devices is difficult. In order to overcome these limitations, we have designed E-Spion to be portable to several IoT device architectures and Linux distributions. We employ a device-edge split architecture to minimize computational/storage overheads on the IoT device. In order to provide greater flexibility

and reliability, we have developed 3 layers of detection with varying levels of trade-offs between detection efficiency and overhead costs. To the best of our knowledge, E-Spion[1] is the first portable and light-weight behavior based IDS tailored for IoT devices which monitors system level data.

Contributions. To summarize, we make the following contributions:

(1) A system level IDS E-Spion, 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 higher overhead costs on the device.
(5) A realistic threat model for IoT devices and a secure scheme for storage and transfer of device logs using hash chains which we refer to as the hash-chain verifier.
(6) An extensive evaluation of our system using 3973 malware samples assembled from attacks seen in the recent years on IoT devices and an analysis of the malware samples in terms of our device logs.

The rest of the paper is organized as follows. First, we discuss in detail the design of E-Spion and its components in Section 2. Next, we define a threat model and analyze the security of our scheme in Section 3. We perform the evaluation of E-Spion in terms of detection efficiency and overhead costs in Section 4. We outline conclusions and future work in Section 5. The paper also includes an appendix with details on the malware dataset and the evaluation testbed used for our experiments. The appendix also discusses the efficiency of the different modules of E-Spion and related work.

2 E-SPION DESIGN

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

2.1 Design Overview

Our device-edge split architecture has 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 SSH channel. All the computationally intensive operations, i.e. parsing logs to extract features, authentication of logs, training classifiers, running classifier models, module management etc., are performed on the edge-server. We have employed such a local edge computing strategy to minimize workload on the IoT devices.

Our device profile is built in three layers using three types of device logs (one per each layer) obtained from three types of information: running process names, running process parameters, and system calls made by these processes. Since each of these log types has different overheads for recording, storage, and analyzing, we maintain three separate modules which handle each type of device log, namely PWM, PBM, and 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.). 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.

Our system has two phases of operation: learning stage and operation stage.

Learning Stage. The goal of this stage is to build a baseline profile for a newly connected IoT device. We assume that during this stage the device has not been infected and is operating correctly. This will be usually at the time of installation of the device on the network. Our system detects that a new IoT device has connected to the network and builds the baseline profile for it. The baseline profile is built using the three layer system level logs recorded by the *SysMon* modules running on the device. The baseline profile of a device is composed of three components (one from each layer) namely: Process White-list, Process Behavior Parameters, and System Call Behavior Parameters maintained by the PWM, PBM, and SBM modules, respectively. We discuss these modules in more detail in Subsection 2.2.

Operation Stage. The *SysMon* modules on the device continuously record logs on the device and maintain corresponding hash chains for each log. Periodically, a SSH session is opened between the device and the edge server to transfer the device logs and corresponding hashes. We define two key configuration variables for each device: *windowsize* and *interval*. The variables *interval* helps to encompass the periodic and consistent nature of the device behavior in our logs. The device constantly records logs for each module every *interval* seconds and transfers them to the edge every *windowsize* seconds. Overall, every transfer contains a number of PWM, PBM, and SBM logs equal to *windowsize/interval*.

The integrity of the logs transferred is ascertained by the hash chain verifier running at the edge-server. The hash chain verifier is discussed in detail in Subsection 2.3. A verification failure indicates either an intrusion or malfunctioning of the device. After the logs are successfully authenticated, anomaly detection is performed on the logs at the server. The logs are matched with the baseline profile of the device created during the learning stage, using white-listing and machine learning techniques. If any anomalous behavior is detected, an alert is issued. The severity of the attack is assessed according to which layers/modules of the anomaly detection engine detect the anomaly.

2.2 Anomaly Detection Engine

As stated before, our anomaly detection engine is organized into three detection modules: Process White listing Module (PWM),

[1] E-Spion roughly translates to "Spy on the Edge"

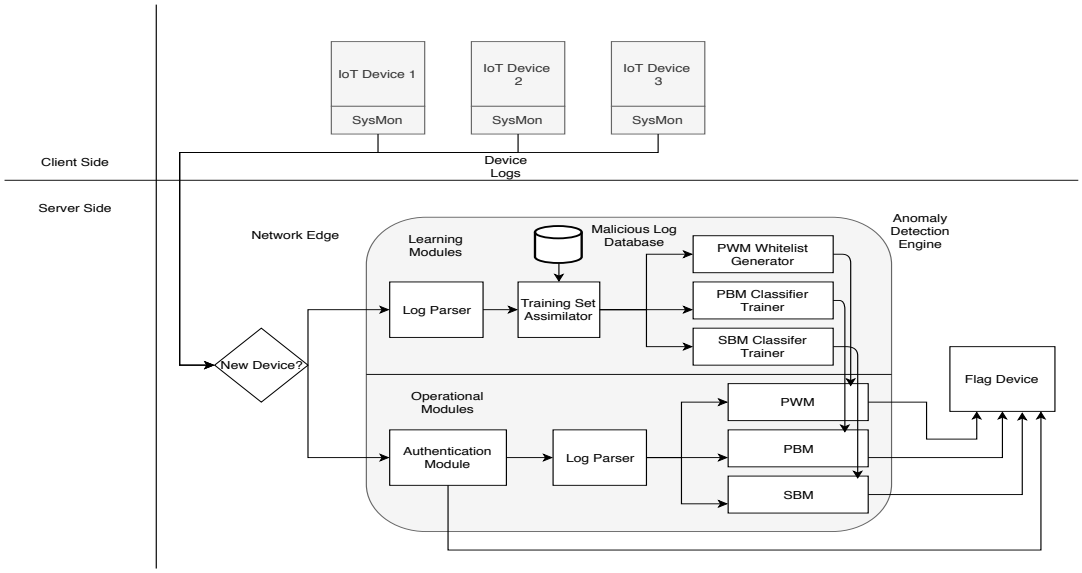


Figure 1: Design overview of E-Spion

Process Behavior Module (PBM), and System-call Behavior Module (SBM).

PWM. This module uses a white listing 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 cheaply as possible.

PBM. 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 from 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 shown in Table 2. We list the metrics used for training our classifier in Table 3. 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.

2.3 Hash Chain Verifier

This component verifies the integrity of the logs received by the edge-server. We do this by maintaining 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 [5] 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 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

Table 2: 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
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 3: 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

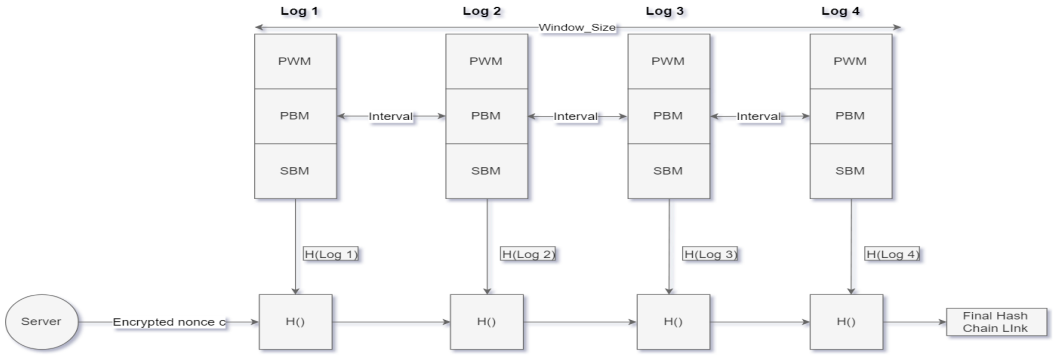


Figure 2: Hash Chain Verifier

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.

3 SECURITY ANALYSIS

In this section, we first introduce our threat model and then we analyze the security and integrity of our logs maintained by the Hash-Chain-Verifier.

3.1 Threat Model

Our threat model is similar to threat models for system logging and auditing schemes [16, 19, 22] used in traditional computer systems. Most of the previous work assumes that the system has a trusted computer base (TCB), like fully-trusted user space applications [11, 12], secure kernel based schemes [22, 26], and hardware assisted secure storage using TPM [2]. Such models are suited for traditional systems but are impractical for IoT devices as there is no guarantee of a TCB on IoT devices. Most IoT device compromises are from either guessing or brute-forcing root passwords or device-specific vulnerabilities. In both cases, the attacker is able to gain root access and, as a consequence, most malware execute with super-user (root) privileges. With root access, the attacker is able to tamper the `E-Spion` logs by removing device log records, inserting false information into these records, or removing all evidence of infection from the logs. So in order to carry out a security analysis of `E-Spion`, we introduce a new threat model for our system.

The goal of the attacker is to compromise the device protected by `E-Spion` in order to either manipulate the functioning of the device (e.g., by breaking the device, reporting incorrect data, etc.) or using the device for other malicious activities, like DDoS, crypto-mining, installing Tor nodes and packet sniffers, DNS hijacks, credential collection etc. To achieve its goal, the attacker can install malware on the device or exploit a benign application running on the device. We assume that the attacker cannot attack the device before it has

been deployed in an E-Spion enabled network (i.e. before the operational phase). Hardware Trojans and devices compromised during production are beyond the scope of this work. All communication between the device and edge-server is through an encrypted, authenticated and forward secure SSH sessions. After deployment is complete, the system works in a malicious environment in which the attacker can compromise and gain root level access to all the devices except the trusted edge-server. The attacker can manipulate all user and kernel space processes on the devices. However, it is important to note that due to the forward secrecy guarantees of SSH [3], even if the attacker can access the private key on the device, it is not able to read the value of the nonce c or logs sent in the previous sessions. The security of E-Spion against such an adversary relies on verifying the integrity of the logs being sent which is done by the hash chain verifier.

3.2 Security of Hash Chain Verifier

We claim that the point of injection of the attack will be recorded as an anomaly in the logs. The only way of evading such a recording is for the attacker to successfully compromise the device and also manipulate all the hash chain links of the logs which contain evidence of the injection of the attack in a very short interval. The purpose of this is that even if an attacker is able to successfully compromise the device (that is, to gain root access), the attacker needs to successfully modify the log files containing the anomalous behaviors and their corresponding hash values to successfully evade our system. Now once an attacker has successfully compromised the device, the evidence of the injection of the attack will be stored in one or more of the previously stored logs. In order to erase evidence of the attack, the attacker must be able to modify the previously stored logs and its hashes which are maintained as a hash chain. For this, the attacker either needs to know the value of the initial nonce c or the hash value of the previous logs in the hash chain. The attacker cannot intercept or manipulate the value of the previously sent nonce using network based attacks, like replay attacks, session-hijacking attacks etc., without breaking the authenticity, integrity and forward secrecy guarantees of the SSH session. On the device, both the nonce c and all previous log hash values are deleted after every short *interval* time. So, we see that the attacker can easily modify files (hash chain links) which are created after the compromise has happened but it is difficult for the attacker to remove/modify the evidence of the attack from the previously stored logs.

Successful forgery is only possible through a timing based attack where the attacker is able to perfectly time the attack at the start of an *interval* (10 seconds in our current prototype) while the previous hash chain link has not yet been deleted. During this time window, the attacker can make the necessary changes to the logs and then compute the masqueraded hash as the attacker has access to the previously stored hash chain link. This is practically infeasible for the attacker as the attacker has only *interval* seconds to successfully compromise the device and also modify the hash chain link generated during this time. This will become even more difficult if the time interval is further reduced. Also the evidence of the infection or attack might be present in more than one layer of E-Spion, making the attack more difficult. If the attacker does not modify all these logs, the attack will be detected once the anomalous behavior is observed in the transmitted logs. We do not provide a formal proof of security of our authentication scheme but rather argue that the attacker has a very low chance of evading our system.

As seen in the timing based attack mentioned before, it is important to note that a skilled attacker with sufficient resources would be able to bypass this authentication mechanism. A more secure way of ensuring the authenticity of the logs would be to use a trusted hardware storage mechanism, like TPM[18], on the IoT devices. The issue is that most of the IoT devices in production currently are not TPM enabled and it is infeasible to require all resource constrained IoT devices to have such hardware assisted security mechanisms. So our choice of authentication mechanism makes our system easily deployable on current IoT devices. However, it is important to note that our system can be easily extended to use hardware assisted log authentication for IoT devices that have such capabilities.

4 EVALUATION

We perform an extensive evaluation of E-Spion on a typical enterprise IoT setup with 3973 of the most recent IoT malware samples. To test E-Spion, we manually download and run 3178 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 effeciency and costs imposed by each layer of E-Spion.

4.1 Detection Effeciency

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 white-listing of process names. This reinforces our claim that most of the IoT malware is basic and does not employ any obfuscation or deception techniques. The PBM layer has a high detection rate of 97.02% but at the same time a false positive rate of 2.97%. We provide additional details about our malware dataset, testbed and per module detection efficiency in the appendix. 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.

4.2 Cost Analysis

In the following, we discuss the performance and storage overheads for the different modules of E-Spion. An important requirement for the design of E-Spion is that the system should impose minimal costs on the already resource constrained IoT devices. 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 requires minimal computational resources. We observe that the SBM module is the most computationally 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

Table 4: 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

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 largest 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 the 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 3 modules have similar overhead costs on the server side. We see that that the three modules of E-Spion have varying degrees of computational/storage overheads and detection efficiency. Finally in Table 4 we summarize our comparisons of each module of E-Spion in terms of accuracy and overhead.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced E-Spion, a system-level IDS tailored for IoT devices. E-Spion builds 3-layered baseline profiles with varying overhead costs for IoT devices using system information and detects intrusions according to anomalous behavior. E-Spion is specifically designed for resource-constrained IoT devices with an efficient device-edge split architecture and modular 3-layered design. We extensively tested E-Spion with a comprehensive set of 3973 IoT malware samples and 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[21] and Heimdal [14]. The goal is to provide more fine-grained detection and build more comprehensive device behavior profiles.

REFERENCES

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

[2] Adam M Bates, Dave Tian, Kevin RB Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel.. In *USENIX Security Symposium*. 319–334.

[3] Vincent Bernat. 2011. SSL/TLS & perfect forward secrecy. *ht tp://vincent. bernat. im/en/blog/2011-ssl-perfect-forward-secrecy. html* (2011).

[4] Timothy R Chavez. 2006. A Look At Linux Audit. In *LinuxWorld Conference and Expo, Boston, MA, April*.

[5] 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.

[6] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *IEEE Symposium on Security & Privacy*.

[7] Gideon Creech and Jiankun Hu. 2014. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *IEEE Trans. Comput.* 63, 4 (2014), 807–819.

[8] Dimitrios Damopoulos, Georgios Kambourakis, and Georgios Portokalidis. 2014. The best of both worlds: a framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones. In *Proceedings of the Seventh European Workshop on System Security*. ACM, 6.

[9] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. 2012. MADAM: a multi-level anomaly detector for android malware. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*. Springer, 240–253.

[10] Ahmed M Fawaz and William H Sanders. 2017. Learning Process Behavioral Baselines for Anomaly Detection. In *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*. IEEE, 145–154.

[11] Ashish Gehani, Basim Baig, Salman Mahmood, Dawood Tariq, and Fareed Zaffar. 2010. Fine-grained tracking of grid infections. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*. IEEE, 73–80.

[12] Ashish Gehani and Ulf Lindqvist. 2007. Bonsai: Balanced lineage authentication. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 363–373.

[13] Ashish Gehani and Dawood Tariq. 2012. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 101–120.

[14] 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.

[15] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. 1998. Intrusion detection using sequences of system calls. *Journal of computer security* 6, 3 (1998), 151–180.

[16] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 377–390.

[17] Prabhakaran Kasinathan, Gianfranco Costamagna, Hussein Khaleel, Claudio Pastrone, and Maurizio A Spirito. 2013. An IDS framework for internet of things empowered by 6LoWPAN. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1337–1340.

[18] Steven L Kinney. 2006. *Trusted platform module basics: using TPM in embedded systems*. Elsevier.

[19] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting.. In *NDSS*.

[20] Open Malware. [n. d.]. http://openmalware.org

[21] Daniele Midi, Antonino Rullo, Anand Mudgerikar, and Elisa Bertino. 2017. KalisâĂŤA System for Knowledge-Driven Adaptable Intrusion Detection for the Internet of Things. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 656–666.

[22] 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.

[23] 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.

[24] 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.

[25] Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. 2007. Analysis of computer intrusions using sequences of function calls. *IEEE Transactions on dependable and secure computing* 4, 2 (2007), 137–150.

[26] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 259–268.

[27] 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).

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

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

[30] Miao Wang, Cheng Zhang, and Jingjing Yu. 2006. Native API based windows anomaly intrusion detection method using SVM. In *null*. IEEE, 514–519.

A MALWARE DATASET

While there are different variants of IoT malware, we have built a comprehensive dataset using 3973 malware samples from the most popular malware families: Zorro, Gayfgt, Mirai, Hajime, IoTReaper, Bashlite, nttpd, linux.wifatch etc. The malware samples were collected from IoTPOT [24], VirusTotal [29], and Open Malware [20]. These malware executables are compiled for different CPU architectures and endianess. 2572 of the samples are compiled for little endian processors while 1421 of them are for big endian processors. We have used 20% (795) of our malware samples as training malware samples for the learning stage of `E-Spion` and 80% (3178) of the them for evaluating `E-Spion` in our experimental setup.

B 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).

C MODULE DETECTION EFFICIENCY AND ANALYSIS

PWM layer. 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., white-listed process). We observe that most malware invoke the prctl system call and use 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 masquerade 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 white-listed during the learning stage and none of the benign applications spawn any new unwhite-listed processes during the operation stage.

PBM Layer. As we are monitoring for anomalous activity 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 they infect 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 Figures 4a, 4b, and 4c, respectively. 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. 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 a detection accuracy of 94.2%, 96.67%, and 91.46%, respectively. Although using the metrics individually gives 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. 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.

D RELATED WORK

Intrusion Detection for IoT. Most existing IDSes for IoT devices and embedded devices, like [17, 23], 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 etc., 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 all the signatures 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 schemes detection schemes have been proposed [27, 28] which rely on detecting anomalies by inspecting packet rates, sizes, payloads, headers, node connections, energy consumption, device profiles etc. 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.

Overall, our approach is significantly different from all the previous approaches in the area as we aim to build a hybrid light-weight

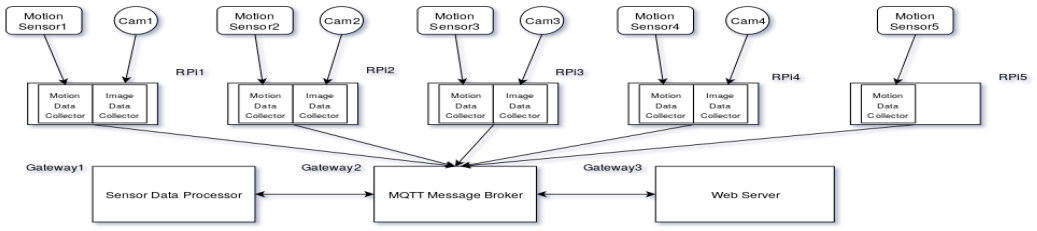
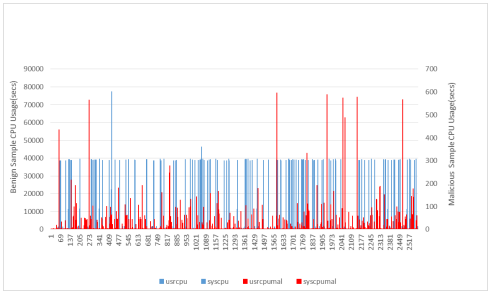


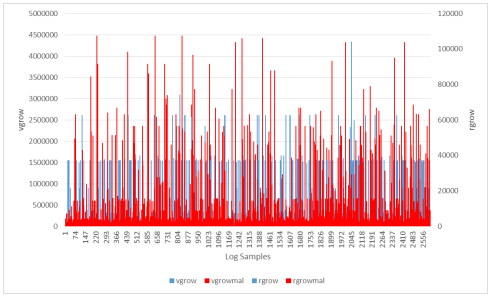
Figure 3: Testbed Implementation Details

IDS system which is able to detect anomalous behavior in terms of system level information from running running processes and system calls.

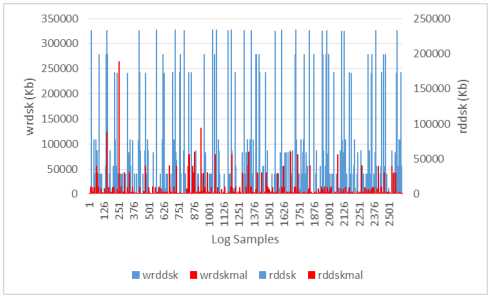
Traditional system level techniques. Even though there is little research on system level IDS technology for IoT, there has been significant research in the area of intrusion detection using system events and provenance logging for traditional computer systems. There are two main approaches. The first approach is based on system event logging and then causally connecting these events during attack investigation to build causal graphs like Auditd [4] in Linux kernels which maintains audit logs of important system events. The other approach is using provenance propagation like in PASS [22] 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 [13]. ProTracer [19] proposes a light-weight provenance "tainting" scheme which is a hybrid of both those approaches. The most comprehensive of such hybrid IDSes is RAIN [16] which achieves higher run-time efficiency by pruning out unrelated executions in their provenance graphs. Although these approaches have similar ideas compared to `E-Spion`, 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 have amounts of computational and storage costs which are too high for IoT devices. By contrast, `E-Spion` uses a simpler and more efficient 3-layered approach to model IoT device behavior using system data. Also, `E-Spion` leverages network edge-servers and is thus able to minimize the workload on the devices. This makes our approach practical and cost-efficient for IoT devices.



(a)



(b)



(c)

Figure 4: 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)