ADROIT: Detecting Spatio-Temporal Correlated Attack-Stages in IoT Networks

Dinil Mon Divakaran*, Rhishi Pratap Singh†, Kalupahana Liyanage Kushan Sudheera†,
Mohan Gurusamy† and Vinay Sachidananda*
*Trustwave, †National University of Singapore
dinil.divakaran@trustwave.com, rhishi@comp.nus.edu.sg, kushan@nus.edu.sg,
elegm@nus.edu.sg, vinay.sachidananda@trustwave.com

***Abstract*—As IoT devices become increasingly deployed for personal as well as commercial purposes, the cyber threat landscape is also changing with recent years witnessing attacks with higher intensity and sophistication. Attacks consists of multiple stages, such that the individual attack-stages not only happen at different times but are also dispersed spatially across large number of IoT devices residing in multiple networks. These characteristics make it challenging to detect the attack-stages using solutions that are localized in space and time. This work looks into the problem of detection of attack-stages in IoT networks. We develop `Adroit`, a system that correlates anomalies across different networks and different time-windows, using a scalable network architecture. In `Adroit`, network traffic of devices is processed locally to detect potential anomalous behavior. Alerts on the anomalies are regularly sent to a security manager residing in the cloud, which employs a well-known data mining approach, FIM, to extract attack patterns. Results from our preliminary experiments conducted using an OpenStack environment are encouraging — `Adroit` is able to detect attack-stages with high accuracy while filtering out much of false alerts.**

I. Introduction

The unprecedented growth in the number and type of IoT devices as well as device-centric applications makes IoTs an attractive attack surface, thereby introducing new challenges to cyber security and privacy [11]. Recent times have witnessed IoT devices being compromised and exploited for launching large-scale attacks causing huge losses [7, 10]. A cyber attack consists of multiple stages such as social engineering campaign, reconnaissance, intrusion, infection, C&C communication, and launch of targeted attacks (e.g., TCP SYN flooding, DNS-based DDoS attacks, data exfiltration, etc.). To deal with this rapidly evolving threat landscape, it is important to detect different stages of large-scale attacks (referred to as *attack-stages*) as early as possible. Such a detection solution is useful in many ways, providing early warning of malware spread, extraction of signatures, timely prediction of attack-stages and quick mitigation. To develop an effective detection solution, there are two challenges we need to overcome:

1. Spatial dispersion: Threats and attacks target as well as originate at different network premises. This means, a traditional detection solution residing at one network premise will observe only malicious activities related to its own network [9], and would not have visibility of similar or related activities in other networks. For example, a short burst of packets to a new destination from one device might be missed or ignored, but if the same pattern is observed at multiple premises, that could be an indication of a sophisticated low-rate DDoS attack [5].

2. Temporal dispersion: The malicious network activities due to the attack-stages can happen at different times. A bot can remain infected for a long duration, and during that time, it may engage in multiple rounds of one or more attack-stages such as scanning for and exploiting other vulnerable devices, loading of malware, exflitration of data, etc. A particularly interesting example is that of establishing communication with a C&C server — attempts of a newly infected bot to locate its C&C server (e.g., using a static list of IP addresses or DNS) may happen over hours or days to evade detection.

Detecting malicious activities in silo does not provide the broader view required to connect the dots and recognize the larger threat taking shape across different networks and at different times. While these challenges existed before, they are amplified in the IoT era, as demonstrated by Mirai attack [7] and Hajime botnet [10]. A naive approach to detect different stages of an attack from various networks is to have a (cloud-based) centralized solution, to which all traffic flows from all devices in deployed networks are sent to. The centralized solution can then process the network traffic to look for specific patterns of attack-stages (see [6, 12]). But such an approach has multiple disadvantages: (i) The amount of data from all devices that needs to be sent to and processed at the centralized location is going to be humongous. The constrained resources in this case (based on the available budget) are not just the computational power and storage requirements at the cloud, but also network bandwidth required to stream data from enterprises to the cloud. (ii) IoT device communication consists of private and confidential data. It is now well-known that even traffic analysis that excludes payloads leaks sensitive information [2]. Therefore, users would be unwilling to store device traffic in the cloud, where there is risk of information leak. (iii) Models to detect known specific patterns might miss out new attack-patterns; for example, the manifestation of vulnerability exploitation of Wannacry ransomware in network traffic is different from that of Mirai botnet.

To tackle the above challenges, we propose `Adroit`, a system that correlates suspicious activities across space and time to detect patterns related to attack stages. `Adroit` is build on a distributed network architecture depicted in Fig. 1 (similar

Workshop on Decentralized IoT Systems and Security (DISS) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-64-9
https://dx.doi.org/10.14722/diss.2020.23006
www.ndss-symposium.org

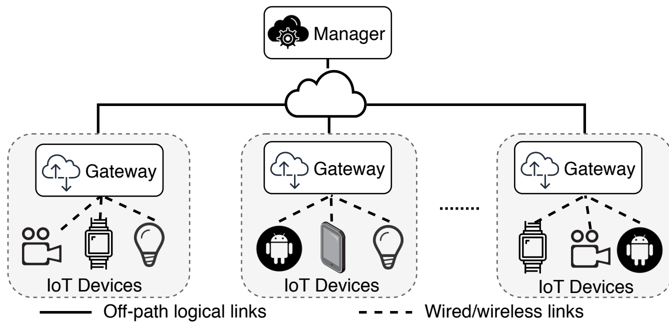


Fig. 1: Distributed network architecture for IoT security

to recent works [14, 3]). A *gateway* resides at the perimeter of a network premise[1], connected to all IoT devices in that network. Gateways have the capability to passively monitor network traffic of the connected devices. We assume gateways have reasonable computational and storage resources, as is the case with home access routers today. The gateways at the premises are all connected to a *security manager* residing in the cloud or an ISP datacenter; therefore the links (in the figure) connecting the gateways and the manager are only logical links. The manager is essentially a software logic that could be deployed either independently as a standalone software, or as an application that is integrated with the SIEM (Security Information and Event Management) engine [4], or even as an SDN application (thereby assisting in real-time mitigation).

They key properties of `Adroit` are the following:

1. Traffic of IoT devices are processed only *locally* at the gateways. Only alerts (information on anomalies generated at the gateways) are sent to a *security manager*. Thus, in comparison to sending network traffic of all devices to a centralized entity, `Adroit` reduces the bandwidth required by orders of magnitude.

2. `Adroit` minimally leaks private information of devices. As we will see in Sec. II-A, a gateway at a network premise constructs, stores and updates normal profiles of IoT devices. Alerts are generated only for those traffic flows that do not match against the profile, i.e., for 'anomalous' traffic.

3. `Adroit` takes an unsupervised approach for detecting the various attack-stages. We employ a well-know data mining technique—FIM (frequent itemset mining)—at the security manager, to efficiently and automatically extract patterns corresponding to attack-stages (Sec. II-B). This requires no knowledge of specific attack-stages, and therefore has the potential to detect new malicious activities.

4. The distributed architecture of `Adroit` removes the constraint arising from the spatial dispersion of IoTs and network premises, and allows us to correlate alerts from different gateways at the security manager. Furthermore, we also develop a window-based mining approach, that extracts patterns across past time-windows based on the patterns detected in the current time window (Sec. III). Thus, `Adroit` performs both spatial and temporal correlation of alerts.

[1]A *premise* could be a home consumer or an enterprise.

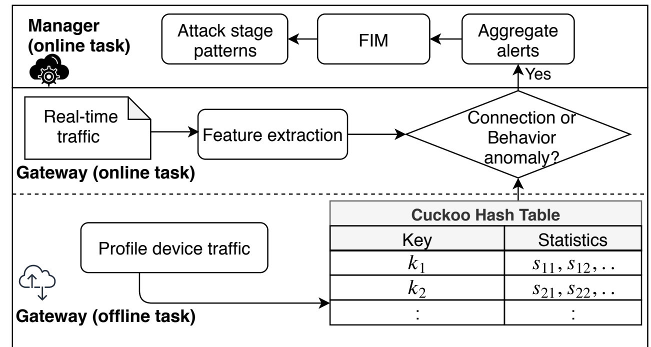


Fig. 2: Functional block diagram of `Adroit`

II. PROFILING AND ANOMALY DETECTION AT GATEWAYS

Fig. 2 shows the functional block diagram of `Adroit`. Below, we discuss the tasks carried out by a gateway.

A. *Profiling IoT devices*

We define a profile of an IoT device as a concise representation of the traffic characteristics of that device. We assume most IoT devices (such as smart camera, smart lock, etc.) use a specific set of protocols, as a device serves very specific functionality. The set of protocols and applications a device uses are highly dependent on the device functionality. Yet, the mappings of devices to protocols and servers are deterministic in the sense that, we do not expect them to change, unless there is a firmware/software update [9, 14]. Exceptions to this assumption are IoT hubs, e.g., Google Home.

As mentioned earlier, we assume a gateway has sufficient memory, and therefore it can store profiles of IoT devices in the memory for fast processing. Intuitively the right time for profile construction is when a device connects to the network (gateway) for the first time. Since the normal behavior of IoT device can change due to system updates, profiling also needs to be carried out multiple times during the lifetime of the device. It is reasonable to assume that, such profile updates need to be performed only in the timescale of days, and not every minute/hour. Therefore, the process of profiling devices can be considered as an offline task.

Cuckoo hash table for IoT profiles: One could use a hash table to store the profiles of IoT devices at a gateway. Once the profile table is built, it will be used for real-time detection of anomalies in IoT communications. For anomaly detection at gateways, as will be seen in Sec. II-B, the only operation performed on a profile table is the `lookup` operation. A traditional hash table cannot guarantee a constant `lookup` time, whether the scheme used is chained hashing or linear probing. This is where the Cuckoo hash table (CHT) [13] is useful. In a CHT (with two hash functions), exactly two bucket locations are accessed to perform a `lookup` operation in the worst case. To achieve this constant time lookup, the compromise made in the CHT design is for the `insert` operation [13]. But since `insert` operation is only required for constructing a profile, and is therefore only used offline, the overhead for `insert` operation is tolerable.

For an IoT device d, a profile table $\mathcal{P}^d$ is created by processing network traffic flows of that device during a pre-

TABLE I: Profile of D-Link socket

dstIP	dstPort	Protocol	Dir	Count		Size	
				mean	std	mean	std
dns.google.	53	UDP	Out	2	0	219.8	4.3
api.dch.dlink.com.	80	TCP	Out	10	0	1227	0
api.dch.dlink.com.	443	TCP	Out	22.6	2.26	5792.4	955.4
ntp1.dlink.com.	123	UDP	Out	2	0	152	0
r0802.dch.dlink.com.	443	TCP	Out	124.4	9.39	5212.9	974.8
tzinfo.dch.dlink.com.	80	TCP	Out	10	0	824	0
wrpd.dlink.com.	80	TCP	Out	10	0	1202	0

defined interval. For this purpose, a traffic flow is identified by the common 5-tuple of source and destination IP addresses, source and destination ports and protocol, such that two flows with the same 5-tuples are separated in time by a threshold (say, five seconds). We refer to the 5-tuple flow identifier as `fid`. Though, we process traffic in 5-tuple flows, the profile table stores only session-level information. We define a session as an aggregation of 5-tuple flows, localized in time, and having the same 4-tuple of `srcIP, dstIP, dstPort, Protocol`. In other words, multiple connections between two end-points for the same service (e.g., HTTP) are aggregated into one session by dropping the source port (which changes randomly with every new connection). In practice, it is also useful to separate incoming and outgoing connections; this can be achieved either by using two tables or by adding another attribute in addition to the 4-tuple key—`Dir`, which will indicate the direction of the connections in a session.

Essentially, the profile table of a device is indexed using the session identifier `sid` = {`srcIP, dstIP, dstPort, Protocol, Dir`}. In the indexed slot corresponding to a session in the CHT $\mathcal{P}^d$, we store statistical information such as mean and standard deviation of different features such as flow-size in packets (count), flow-size in bytes (size), etc. Table I provides the profile table of a D-Link socket.

B. Anomaly detection at gateways

There are two kinds of anomalies we detect at a gateway:

Connection anomaly: A connection to a new external IP address or use of a new application (port) not found in the profile table is suspicious. Such a new connection will not have an index in $\mathcal{P}^d$ for the specific device d. To detect this, for a packet corresponding to device d, the gateway extracts the values for the `sid` fields (`srcIP, dstIP, dstPort, and Protocol` from the header; `Dir` from the direction of the first packet of the flow), and performs a `lookup` on the profile table $\mathcal{P}^d$ with `sid` as the key. If the `lookup` fails, it is a connection anomaly. All connection anomalies for device d are stored in a hash table $\mathcal{C}^d$; different from $\mathcal{P}^d$ the 5-tuple `fid` is used as an index in $\mathcal{C}^d$. Therefore, an alert for an anomaly corresponds to a flow.

Behavior anomaly: It is possible that a connection finds a matching key in the profile table $\mathcal{P}^d$, and yet there is a statistical anomaly. For example, an attacker might compromise a device and host the data extfiltration service on the same cloud platform which hosts the device's application. This category of anomalies consist of flows that already map to a valid key in the profile table $\mathcal{P}^d$, but yet deviates from the "normal" behavior. To detect behavior anomalies, a gateway maintains a hash table $\mathcal{B}^d$ for device d, consisting of all the *active* 5-tuple flows — flows that had at least one packet in the last t seconds (t is a configurable parameter). Once the flow becomes inactive (meaning no packet was transmitted in the last t seconds), the gateway uses z-score to compare the flow features (sizes of flow in bytes and packets as well as flow duration) to the corresponding session features in $\mathcal{P}^d$. If the z-score is high, the flow is deemed as statistical anomaly, and an alert is generated. Once a flow becomes inactive, it is removed from $\mathcal{B}^d$.

A gateway temporarily stores anomalies in tables $\mathcal{C}$ and $\mathcal{B}$ (the superscripts are dropped for readability). At every predefined interval, a gateway sends all alerts stored in both tables to the security manager. Subsequently, both the hash tables are cleared of all entries. Fig. 3 gives several examples of alerts seen at a security manager. The first six fields corresponds to the 5-tuple `fid` and direction `Dir`, while the last column is the size of the flow in discrete categories.

III. Attack-stage detection at security manager

The security manager and gateways connect and communicate via secure web sockets. The manager gathers alerts from all connected gateways and stores them in a database. We now describe the process of attack-stage detection at the manager.

A. FIM based pattern extraction

The challenge at the manager is to process the alerts and extract out only meaningful patterns corresponding to attack-stages. Observe that, not all alerts are related to attack-stages, or even malicious. Alerts can be generated by a gateway, if the profile did not capture the normal behavior perfectly. This can happen for a number of reasons, say, due to changes in firmware, applications or user behavior. Besides, random scans unrelated to attacks take place frequently around the Internet for research and other purposes. We refer to all such alerts not related to attack-stages as *false alerts*. Therefore, the manager is tasked to mine for attack patterns by automatically filtering out false alerts (noises). We argue that there are two ways to identify noises: (i) if alerts corresponding to such noises are low in number, and (ii) if there are no 'similar' alert(s) across more than one gateway (network premise).

To achieve our goal, we propose to employ FIM (frequent itemset mining) [1], a data mining technique used to extract recurring patterns across all transactions (in our case, alerts). In the FIM language, each field of an alert is an item, and a set of k items is called an k-itemset, where k is the length of the itemset. For instance, each incoming alert in Fig. 3 has seven items, whereas, the initial four extracted itemsets (in the bottom table) have five items ($k = 5$) and the last itemset has six ($k = 6$). Given a list of n alerts, an itemset (i.e., a pattern) is called a frequent itemset, if it appears in at least $\theta \times n$ alerts, where $\theta, 0 \leq \theta \leq 1$, is called the *minimum support*. Therefore, the goal of FIM is to mine frequent itemsets in an alert database.

We can mine all possible frequent itemsets, also called as lattice, in the alert database using a fundamental FIM algorithm such as Apriori [1]. Though, the complete lattice provides a comprehensive overview of all the patterns, the extent of patterns is exhaustive, and the complexity associated with the generation of the lattice can go up to $O(\eta \times 2^{\eta-1})$, where η is the total number of itemsets [1]. Moreover, many patterns are closely related, and generally, lower length itemsets are

Incoming Alerts

#	srcIP	dstIP	Protocol	srcPort	dstPort	Dir	sizeBin
1	scanner1.com	10.6.1.12	TCP	45678	23	In	Small
2	scanner2.com	10.6.1.12	TCP	56897	23	In	Small
3	scanner3.com	10.6.2.2	TCP	55001	23	In	Medium
4	scanner3.com	10.6.5.173	TCP	45877	23	In	Medium
5	10.6.2.2	cnc.com	TCP	23669	48000	Out	Medium
6	10.6.5.173	cnc.com	TCP	56814	48000	Out	Medium
:	:	:	:	:	:	:	:
31	10.6.2.2	victim1.com	TCP	23456	80	Out	Medium
32	10.6.5.173	victim1.com	TCP	35689	80	Out	Medium
33	victim2.com	dns.server	UDP	13074	53	Out	Small
34	victim2.com	dns.server	UDP	18869	53	Out	Small
	:	;	:	:	:	:	:
101	10.6.2.13	firmware1.com	TCP	49225	80	Out	Large
102	10.6.13.144	random1.com	TCP	48369	443	Out	Medium
103	firmware2.com	10.6.19.66	UDP	23698	69	In	Large
:	:	:	:	:	:	:	:

Alerts related to attack stages (rows 1–34)

False alerts, not related to attack stages (rows 101 onward)

FIM

Extracted Itemsets

#	srcIP	dstIP	Protocol	srcPort	dstPort	Dir	sizeBin
1	*	10.6.1.12	TCP	*	23	In	Small
2	scanner3.com	*	TCP	*	23	In	Medium
3	*	cnc.com	TCP	*	48000	Out	Medium
4	*	victim1.com	TCP	*	80	Out	Medium
5	victim2.com	dns.server	UDP	*	53	Out	Small
:	:	:	:	:	:	:	:

Fig. 3: Example of alerts received at the manager (top table), and maximal frequent itemsets (MFI) mined (bottom table).

subsets of higher length itemsets, and thus, redundant. For instance, in the case of lattice, there would be redundant patterns such as `<<*,*,TCP,*,23,In,*>>` (where $\star$ stands for any value for that item) in addition to the first pattern `<<*,10.6.1.12,TCP,*,23,In,Small>>`, among the mined patterns listed in the bottom table of Fig. 3.

As alternatives, we can generate subsets of the lattice in the forms of closed frequent itemset (CFI) and maximal frequent itemset (MFI), where the itemsets in the former do not have any superset with the same support, while the itemsets in the latter do not have any superset which is frequent. Both CFI and MFI have significantly lesser number of itemsets than the lattice, while MFI is itself a subset of CFI. In terms of contained information, the patterns in the MFI have much more information as they are generally of higher length, and the number of patterns and complexity are lowest. Thus, in this work, we mine for MFI.

We now revisit Fig. 3. The patterns shown in the bottom table are obtained when mined for MFI. For instance, when mined with a minimum support count of 2, alerts 31 and 32 form a pattern of `<<*,cnc.com,TCP,*,48000,Out,Medium>>`, and it can be interpreted as multiple IoT devices attempting to connect with `cnc.com`. However, in the case of false alerts, i.e., not related to the attack-stages, a clear pattern is not visible. `Protocol` and `Dir` are the only frequent items; but this itemset is of very small length and will disappear if itemsets mined for are MFI.

B. Pattern search algorithm

Pattern mining is performed on an aggregation of alerts, which is collected over a time-window. The length of the window obviously influences the time to detect, and we keep it as a configurable parameter. When the attack detection is executed across multiple time-windows, it enables `Adroit` to take temporal correlation into account. Furthermore, this makes it possible to have different minimum support in different time-windows. Note that, minimum support is the most important parameter that will decide the accuracy of our detection system. If it is too low, false alerts would be extracted as patterns (false positives); and if it kept too high, some of the attack-patterns might not be extracted. Hence, we come up with a detection algorithm, presented in Algorithm 1, that dynamically adapts the minimum support across intervals.

Algorithm 1 Pattern mining at time-slot τ with look-back

Input: $\mathcal{F}$: mined patterns (an array), $\mathcal{A}$: alerts, θ_l: lower bound of minimum support, Δ^-, Δ^+: decrement and increment step sizes of minimum support, T_w: look-back time-slots

1: $\mathcal{F}[\tau] \leftarrow$ `MFI_Iter`(`any_pattern`, $\mathcal{F}, \mathcal{A}[\tau], \theta, \theta_l$) ▷ mine for any maximal frequent itemset in alert database at time τ while reducing θ iteratively until θ_l
2: **for each** $t \in \{\tau, \ldots, \tau - T_w\}$ **do**
3:     **for each** $\mathbf{I} \in \mathcal{F}[\tau]$ **do**
4:         $\theta' \leftarrow (\theta - \Delta^-)$
5:         $\mathcal{A}' \leftarrow$ `filterAlerts`($\mathbf{I}, \mathcal{A}[t]$); ▷ filter the alert database by pattern $\mathbf{I}$
6:         $\mathcal{F}' \leftarrow$ `MFI_Iter`(`new_pattern`, $\mathcal{F}, \mathcal{A}', \theta', \theta_l$) ▷ mine for any new pattern in filtered alert database $\mathcal{A}'$ while reducing θ' iteratively until θ_l
7:         $\mathcal{F}[t] \leftarrow \mathcal{F}[t] \cup \mathcal{F}'$ ▷ add new patterns
8:     **end for**
9: **end for**
10: $\theta \leftarrow (\theta + \Delta^+)$ ▷ increase for next time-slot

To vary the minimum support, the algorithm is guided by the patterns that are detected. The algorithm begins by initializing the minimum support with a high value. Once a pattern is detected (line 1), assuming it captures one of the attack-stages, the algorithm explores for more patterns with a lower minimum support. However, mining on all alerts is not only computational expensive, but may also risk extracting patterns of false alerts. To overcome this, we limit mining on alerts *related* to the already detected patterns (line 5). That is, we first prune the alert database by filtering out alerts that do not have one of the IP addresses of the detected pattern. We highlight that, our mining solution is modified to also mine for IP subnets from IP addresses (in either the `srcIP` or `dstIP` fields). Therefore, the pruning process is based only on IP addresses or IP subnets of previously mined patterns. Second, this conditional mining (line 6) is carried out on pruned alerts of not just the current timeslots, but also for T_w previous time-slots (line 2-9), thereby considering correlation across time. Essentially, the algorithm does a *look-back* in time for more patterns conditioned on the patterns detected in the current time-slot. Moreover, to avoid the minimum support being set to a very small value (and thereby increase the false alert patterns), we increase the minimum support by a constant (line 10) at the end of the current time-slot.

IV. Performance evaluation

A. Experimental setup

For our experiments, we emulate a Mirai-like environment [7] as illustrated in Fig. 4, using OpenStack. The entities

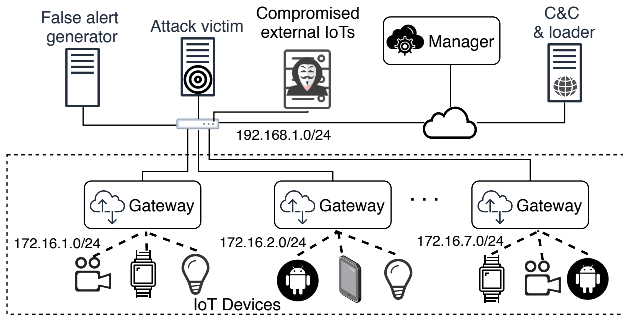


Fig. 4: Experimental setup

emulated are seven gateways, a security manager, 65 IoT devices, 1 attack victim, 2 compromised external IoTs, a C&C server, a loader and a false alert generator. False alert generator is a set of VMs generating harmless and independent scans-based alerts, the intensity of which is configurable. Note that, the challenge of detecting meaningful anomalies (attack-stages) increases with the intensity of noises, which, as explained previously, can arise due to a host of reasons such as network errors, changes in user or application behavior, firmware updates, false positives, etc. The gateways and the security manager run the algorithms that we described in the previous sections. Each gateway continuously monitors the traffic of the connected IoT devices. Alerts generated at the gateways are sent to the manager at every 20 second interval. We implemented the FIM algorithms using the Java-based SPMF software platform [8].

We emulated different stages of a Mirai-like botnet, specifically scans, brute-force login attempts, malware downloads, C&C communications, HTTP DDoS and reflective DNS DDoS attacks on the victim. Scan and brute-force login activities happen throughout the experiment. As a new IoT is compromised, malware is downloaded and communication with C&C servers are made; subsequently the victim server is attacked. DDoS attacks on victim lasts for ≈ 5 minutes. With this setup, we conducted two sets of experiments as follows.

Experiment 1: The goal of this experiment is to analyze the ability of `Adroit` in detecting attack patterns using spatial correlation. To this end, we compare local v/s global detection capabilities; in other words, if mining was performed only at the gateways instead of the security manager, how would the system perform in comparison to mining only at the manager. Therefore, the experiment was run twice, once where FIM was executed only at the gateways, and the second time only at the manager. In each case, FIM is run only once, at the end of the experiment ($\sim$45 minutes). Thus, we also anayze the effectiveness of FIM in mining relevant patterns. Here, we consider multiple false-alert levels, in which level 0 refers to the case, where there is no false alert. The number of false alerts are nearly two times and eight times the number of alerts related to attack-stages, for levels 1 and 2, respectively.

Experiment 2: Our aim with this experiment is to analyze detection capability of `Adroit` for different settings/variants of Algorithm 1, when attack-stages are temporally dispersed. We set a false-alert level of 1. Mining is done for MFIs across multiple time windows with window-size (τ) set to 5 minutes. Here, 80% of the attack-stages generated are spread across 6 time-windows.

B. *Result analysis*

Experiment 1: Fig. 5 plots the detection accuracy, in terms of F_1-score[2], for security manager, individual gateways, and aggregate gateways (*Gateway-all*), for different values of minimum support count (not fraction) varied from 2 to 100 in steps of 2. Note, false-alert level 0 corresponds to the scenario where there are only alerts related to attack-stages. In this scenario (Fig. 5a), clearly, the security manager outperforms the gateways, individually and also collectively, in detecting attack patterns. This observation remains true also for the other two scenarios (Fig. 5b and 5c), but with higher minimum support. Comparing the best cases, the manager outruns *Gateway-all* by 7%, 11%, and 7.5%, in Fig. 5a, 5b, and 5c, respectively. The manager mines on alerts received from all gateways; hence if alerts of an attack-stage is spread spatially across different gateways, the manager has the capability to mine them *together* and extract patterns that would be otherwise missed at gateways due to lower count in alerts. Nevertheless, gateways are also able to achieve good performance (in terms of F_1-score), for attack-stages targeting their respective networks, thereby demonstrating the capability of FIM in mining attack patterns. In general, we observe that detection capability is dependent on the minimum support and having a static minimum support is not advisable.

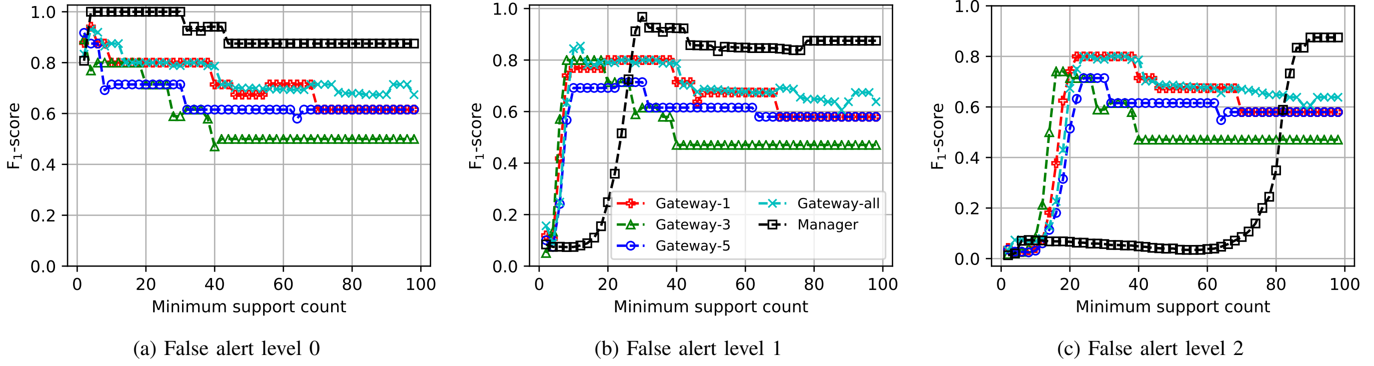
Experiment 2: Fig. 6 plots the F_1-score and the number of attack-patterns detected (True Positives) for the different time-slots. The four bars correspond to four different approaches for mining. In both *constant minimum support* and *search w/o look-back* algorithms, mining is performed only in the current window (i.e., the algorithm goes up to line 1 of Algorithm 1), yet, in the former, the minimum support does not change. Both *search w/ look-back* variants execute Algorithm 1 fully, with the only difference being the look-back time-slots (T_w). From the figure, it is evident that the proposed pattern search algorithms with look-back have extracted significantly higher amounts of attack related patterns, in comparison to other two variants (in terms of True Positives, more than 2.7×), and importantly, across multiple time windows. Not only have they extracted more attack patterns, but also achieve higher F_1-score in many time windows. *Constant minimum support* algorithm does not dynamically adapt to the intensity of attacks, and thus, fails to extract a significant amount of true positives. Although, *search w/o look-back* dynamically changes the minimum support, it does not consider the temporal dispersion of attack-stages, thereby missing out patterns related to the ones detected in the current time-window. The look-back algorithm dynamically adapts, consequently achieving a significant performance improvement.

V. Discussions

A. *Deployment scenarios for* `Adroit`

1. Securing consumers: We envision the IoT gateways in `Adroit` to be integrated with home routers deployed by ISPs. This gives ISPs capability to run detection and mitigation solutions at home gateway. While an ISP is not supposed to use private information of consumers (such as, packet payloads) for its own purpose, consumers who want an ISP solution to protect its devices might consent to the solution inspecting

[2] F_1-score is the harmonic mean of precision and recall.



(a) False alert level 0

(b) False alert level 1

(c) False alert level 2

Fig. 5: Performance of FIM at gateways and manager.

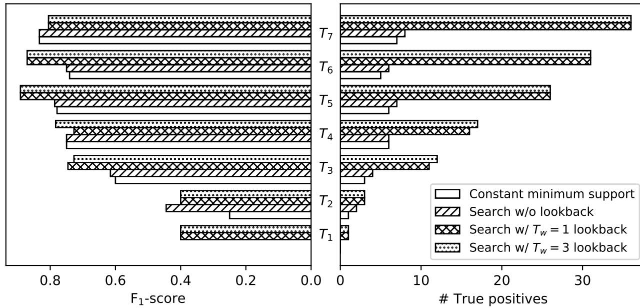


Fig. 6: Performance with different variants of Algorithm 1.

traffic. Yet, a solution in which consumer's private information does not leave the premise is preferred, to minimize risk of information leak. In `Adroit`, only alerts (i.e., meta-information) relating to anomalies leave consumer premise.

2. Securing enterprises: Observe that, the distributed architecture of `Adroit` is also similar to the SIEM (e.g., IBM QRadar) architecture we have today. A log collector deployed at an enterprise collects, processes and sends filtered alerts coming from different security appliances (firewall, IDS, OS event logs, etc.) to the SIEM engine in the cloud. Our solution can be seen as an extension of SIEM into the IoT market.

B. Next steps

In this work, we developed `Adroit` and demonstrated its capability in detecting attack-patterns with low false-positive rate. Going ahead, we plan to carry out extensive evaluation of `Adroit`, across larger timescales and considering botnets beyond Mirai. From a solution perspective, our next step is to identify the detected patterns; i.e., classify the detected patterns into one of the specific attack-stages.

Acknowledgment

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Corporate Laboratory@University Scheme, National University of Singapore, and Singapore Telecommunications Ltd.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. VLDB '94*, pages 487–499, 1994.

[2] Noah Apthorpe, Dillon Reisman, and Nick Feamster. A Smart Home is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic. In *Workshop on Data and Algorithmic Transparency (DAT'16)*, 2016.

[3] Ketan Bhardwaj, Joaquin Chung Miranda, and Ada Gavrilovska. Towards IoT-DDoS Prevention Using Edge Computing. In *USENIX HotEdge Workshop*, 2018.

[4] S. Bhatt, P. K. Manadhata, and L. Zomlot. The Operational Role of Security Information and Event Management Systems. *IEEE Sec. Privacy*, 12(5):35–41, 2014.

[5] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. Tuple Space Explosion: A Denial-of-Service Attack against a Software Packet Classifier. In *Proc. ACM CoNEXT*, 2019.

[6] Dinil Mon Divakaran, Fok Kar Wai, Ido Nevat, and Vrizlynn Thing. Evidence Gathering for Network Security and Forensics. *Digital Investigation*, 20:S56 – S65, 2017.

[7] Manos Antonakakis *et al.* Understanding the Mirai Botnet. In *Proc. USENIX Security*, 2017.

[8] P. Fournier-Viger *et al.* The SPMF Open-Source Data Mining Library Version 2. In *Machine Learning and Knowledge Discovery in Databases*, pages 36–40, 2016.

[9] Ayyoob Hamza, Hassan H. Gharakheili, Theophilus A. Benson, and Vijay Sivaraman. Detecting Volumetric Attacks on IoT Devices via SDN-Based Monitoring of MUD Activity. In *Proc. ACM SOSR*, pages 36–48, 2019.

[10] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and Analysis of Hajime: a Peer-to-peer IoT Botnet. In *Proc. NDSS*, 2019.

[11] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric. All Things Considered: An Analysis of IoT Devices on Home Networks. In *USENIX Security*, 2019.

[12] I. Nevat, D. M. Divakaran, S. G. Nagarajan, P. Zhang, L. Su, L. L. Ko, and V. L. L. Thing. Anomaly Detection and Attribution in Networks With Temporally Correlated Traffic. *IEEE/ACM Transactions on Networking*, 26(1):131–144, Feb 2018.

[13] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[14] V. Thangavelu, D. M. Divakaran, R. Sairam, S. S. Bhunia, and M. Gurusamy. DEFT: A Distributed IoT Fingerprinting Technique. *IEEE Internet of Things Journal*, 6(1):940–952, Feb 2019.