Automatic Recognition of Advanced Persistent Threat Tactics for Enterprise Security

Anonymous Author(s)

ABSTRACT

Advanced Persistent Threats (APT) has become the concern of many enterprise networks. APT can remain undetected for a long time span and lead to undesirable consequences such as stealing of sensitive data, broken workflow, and so on. To achieve the attack goal, attackers usually leverage specific tactics that utilize a variety of techniques. This paper explores the recognition of APT tactics through synthesized analysis and correlation of data from various sources. We propose a framework for detecting the APT tactics and discuss the application of machine learning techniques in this problem. Our framework can be used by the security analysts for effective detection of APT attacks. The evaluation of our approach shows that it can detect APT tactics with high accuracy and low false positive rate. Therefore, it can be used for tactic-centric APT detection and effective implementation of cyber security response operations.

KEYWORDS

Advanced Persistent Threat, Attack Tactics, Machine learning.

1 INTRODUCTION

Cyber attacks against organizations, including Advanced and Persistent Threats (APT), usually employ certain attack tactics. It is common that some attack tactics are used repeatedly in different APT attacks. Identifying these tactics may help understand attackers' potential intent, objectives and strategies, and even help with identification of specific attacker(s) or attack communities. However, the adversary groups often constantly update their tools and tactics to make detection and analysis more difficult. For instance, the DragonOK group, a well-known adversary group, has been evolving their tactics in targeted APT attacks across Asia Pacific and Japan. They began to use multiple new variants of malware "FormerFirstRAT" along with malware "IsSpace" and "Tidepool" in their 2017 tactics [1].

The emergence of new APT tactics has introduced daunting challenges to APT detection. The attackers can leverage a carefully designed combination of various *APT techniques* (e.g., spear phishing, drive by download, buffer overflow, pass the hash) to strategically achieve a goal. Hence, the detection of individual APT techniques is no longer adequate to identify the attacker's intents, objectives and strategies. That is, individual APT techniques cannot tell the "whole story" of the attacks. This inability has put real-world Cyber Security Operation Centers (CSOCs) into a highly undesired dilemma: (a) on one hand, without knowing the "whole story", CSOCs are more likely to take ineffective intrusion response actions; (b) on the other hand, correlating the detected individual APT techniques to generate the "whole story" requires significant amount of time and manual efforts.

A number of research works have explored the problem of detecting APT attacks from different angles. 1) Since APT may remain stealthy for a long time span, capturing all stages of its life cycle is not an easy task. Hence, some research works propose to detect a specific technique that is used in a stage of APT. For example, [11, 17] detect the network connections during the stage of malware command and control (C&C) communication. [17] applies supervised learning towards the web proxy logs to identify and prioritize the enterprise malicious activities, while [11] proposes unsupervised detection of C&C communications based on the web request graphs. [21] detects the APT malware infection by analyzing malicious DNS and network traffic. 2) Some other works aim to detect APT attack as a whole. For example, [7, 20] use classification models for APT detection; [9, 10, 19] reconstruct the attack by combining past security events. HOLMES [16] also leverages the correlation between suspicious information flows for APT detection. 3) Another angle of detecting APT attacks is through provenance tracking. A provenance tracking system captures the causality relationship between system objects such as processes and files. Security analysts can find the root cause of attacks by tracking system object dependencies generated by the provenance data. Because most existing provenance tracking techniques are at low system level and usually suffer from dependence explosion problem, some research works propose to partition execution to units [12, 14]. [13] further proposes to leverage the annotated application specific high level task structures to partition execution. 4) Mining logs is another technique that is commonly used for APT attack detection, although the purposes and approaches of mining may be different. [18] proposes an automated multi-stage intrusion analysis system that is based on mining various logs. The proposed system discovers the "attack communities" from the weighted graphs that are built from multiple logs. [8] proposes to use a deep neural network model, DeepLog, to detect anomaly log sequences. It models a system log as a natural language sequence, learns the log patterns from normal system execution, and reports anomaly when the log patterns deviate from the trained model.

Although extensive researches have been conducted towards detecting APT, few of them emphasize the detection of commonly used APT tactics. For approaches that focus on specific steps or techniques such as C&C communication, or that focus on log mining, the detected anomaly may or may not be relevant to APT. Other approaches, that focus on provenance tracking, event correlation or clustering, are not tactic-centric.

However, identifying tactics is essential to reveal attackers' potential objectives and strategies, and may even help to identify specific adversary groups. Therefore, this paper seeks to propose a framework that can detect APT tactics with high accuracy and general applicability. If successful, the framework is able to: (a) identify the APT techniques that are not easily detected with traditional approaches; (b) match the APT techniques to the tactics they belong to with the help of system object dependencies; (c) make the APT

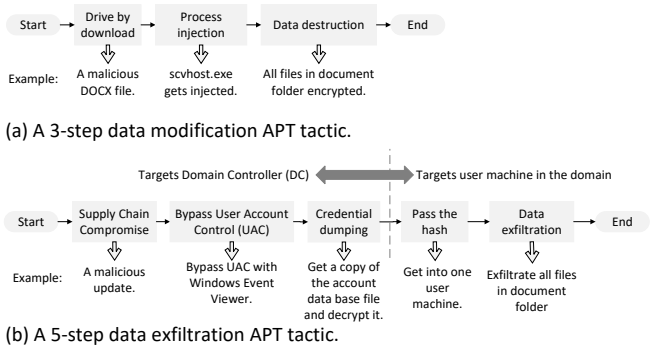

Figure 1: Two example APT tactics.

tactic and APT technique identifiers extensible and adaptive to new tactics and techniques.

The significance of this research is three-folds: 1) it is, to the best of our knowledge, the first framework which could simultaneously achieve accuracy and general applicability in detecting multiple APT tactics; 2) implementations of the framework could help CSOCs and analysts identify the attacker's intents, objectives and strategies, and provide a "whole story" of the attacks; 3) the automated APT tactic identifier could significantly reduce the manual efforts involved in detecting APT techniques.

The remaining of this paper is structured as follows. Section 2 will discuss the APT tactic with two examples, and also the differences from attack graph, which is a well-known graphical method in cyber security. Section 3 will discuss our proposed framework. Section 4 will briefly introduce our design and implementation. In Section 5, a simple five-step APT tactic is presented and used to demonstrate how our framework works. Section 6 presents our evaluation experiments results. Section 7 is the conclusion.

2 PRELIMINARY

The MITRE adversarial tactics and techniques knowledge repository [5] provides a comprehensive review of the real-world adversarial tactics and techniques. In this work, we adopt definitions for tactic and technique different from those in [5]. We define:

- *APT technique*: A specific implementation such as a hacking tool, attack script, and/or malware payload.
- *System object dependency*: The relationships among system objects, such as processes, files and user accounts.
- *APT tactic*: A sequence of APT techniques chained by system object dependencies.

Therefore, APT tactics represent attackers' strategies; and APT techniques represent the specific steps that attackers take to implement the strategies.

In this paper, we present an APT tactic as a connected graph showing the chain of techniques in a multi-step cyber attack. APT techniques are basic building blocks of an APT tactic. All the APT tactic presented in this paper are crafted based on our observation on different attack scenarios. Figure 1 presents two example APT tactics as directed graphs that consist of multiple APT techniques.

Each technique in a tactic has its post-conditions and pre-requisites. Post-conditions are the results of the technique, such as malicious processes being created, files being accessed and user account being modified. Pre-requisites describe the requirements for the technique to be matched into tactic.

In Figure 1(a) and Table 1, an APT tactic about data destruction is presented. This tactic starts from drive by download. In this technique, the download is requested by the user, but the downloaded item include functionalities that user does not expect. The user downloads the program and executes it, without knowing that the program has a trojan built in. The trojan allows a remote attacker to connect and execute malicious commands on the victim computer. With the remote access, the attacker can then escalate the privilege to the system level by process injection. Afterwards, the attacker can destruct all documents by deleting or encrypting them.

In Figure 1(b) and Table 2, an APT tactic about data exfiltration is presented. The first three techniques are launched against Windows Domain Controller (DC), whereas the last two are against another user machine in the domain. This tactic starts with supply chain compromise. For privilege escalation, the attacker bypasses the User Account Control (UAC). UAC prompts user for confirmation when a process requests for system-level privileges. By bypassing UAC, the attacker can escalate privileges without being noticed. After that, the attacker dumps users' credentials such as password hashes. These credentials can be used to launch pass the hash attacks to access other machines. In the end, the attacker downloads sensitive files from target machines.

APT tactics are fundamentally different from attack graphs. Some important differences include: (a) attack graphs represent the causality relationship between vulnerabilities and exploits, whereas APT tactics represent the strategies, techniques and procedures used by attackers; (b) attack graphs show all the possible attack routes from the attacker's machine to the target machine, whereas APT tactics focus on the attackers' chosen techniques and procedures, rather than the attack paths; (c) attack graphs are not being used by CSOCs on a daily basis, whereas APT tactics are frequently referred to by security analysts, though in an implicit and informal way according to our observation.

3 PROPOSED FRAMEWORK

To serve the validity of the proposed framework, we assume that:

- Each APT technique used in the APT tactic is identifiable through automated, semi-automated, or manual effort. In the worst case, the CSOC may have to resort to manual effort to identify a particular APT technique, we have no assumption on the maximum time used to identify an APT technique.
- The whole framework is kept safe from the attacker. All the input data is genuine, which means that the attackers cannot modify or delete them; and the attackers have no access to the framework itself in any way.

The framework presented above needs to address the following challenges:

- *Accurate identification of various adversary techniques.* Although we assume every APT technique is identifiable, some APT techniques, such as pass the hash, are hard to accurately identify with traditional methods like pattern matching and

Table 1: Detailed description of a 3-step APT tactic with system object dependency included.

Technique Name	Post-condition	Pre-requisites	Description	Identification Method
Drive by download	Process P_1 is created.	(None)	Initial intrusion by drive by download attack, resulting to a malicious process P_1 being created.	The downloaded item has some kind of backdoor built-in. Some backdoors are detectable using signature-based IDS.
Process injection	Process P_2 is created; Process P_3 is affected.	Process P_2 is a child process of or the same as P_1.	Privilege escalation by process injection. Process P_2 is the process which initiates the injection, and P_3 is the victim process. P_3 is usually a process running with system-level privilege.	Detectable by monitoring critical system processes.
Data destruction	Process P_4 is created; File F_1 is accessed.	Process P_4 is a child process of or the same as P_3.	Malicious process P_4 accesses file F_1 and makes it inaccessible to the user.	Detectable by monitoring disk I/O on sensitive files/directories.

Table 2: Detailed description of a 5-step APT tactic with system object dependency included.

Technique Name	Post-condition	Pre-requisites	Description	Identification Method
Supply chain compromise	Process P_1 is created.	(None)	Initial intrusion by supply chain compromise, resulting to a malicious process P_1 being created.	Some software distribution or update channels get infected and backdoor gets inserted to the products. Some backdoors are detectable using signature-based IDS.
Bypass User Account Control (UAC)	Process P_2 is created; Process P_3 is created.	Process P_2 is a child process of or the same as P_1.	The attacker bypass the Windows UAC to escalate its privilege to system level.	Many procedures of bypassing UAC needs to modify the Windows registry. Such procedures can be detected by monitoring the registry for specific key creation and modification.
Credential dumping	Process P_4 is created.	Process P_4 is a child process of or the same as P_3.	The attacker leverages escalated privilege to dump user credentials like password hashes.	In a Windows Domain, the DC stores the users' credentials as a database file, and very few processes are allowed to interact with this file. Dumping users' credentials can be detected by monitoring the disk I/O on this file and activities of those special processes.
Pass the hash	Process P_5 is created; User U_1 is impersonated.	(None)	The attacker leverages the password hashes to get into other machines in this domain.	Directly using hashes for authentication relies on certain authentication mechanism, which will leave traces in the network packets. Thus, it is detectable by monitoring the network traffic and inspecting the network packets.
Data exfiltration	Process P_6 is created; File F_1 is read.	Process P_6 is a child process of or the same as P_5.	The attacker, pretending to be user U_1, downloads file F_1 to his/her own machine.	Detectable by monitoring disk I/O on sensitive files/directories.

anomaly detection. Pattern matching suffers from low accuracy, and anomaly detection suffers from high false positive rate.

- *Correct match of the adversary techniques to the tactics they belong to.* Assuming that each technique can be accurately identified, the APT tactic matcher needs to match those techniques into tactics. For receiving inputs, the matcher needs to deal with the diversity of technique identifiers, such as different identification delay; for matching, the matcher needs to deal with multiple cases, namely (a) one attacker is using one tactic, (b) multiple attackers are using one same tactic, and (c) multiple attackers are using different tactics. The framework should address those problems.

For the first challenge, we propose to apply machine learning method for those APT techniques that cannot be accurately identified with traditional methods. Pass the hash is one of such techniques. It is difficult to identify with traditional methods because it leverages legitimate authentication mechanism. To apply machine learning method, a huge amount of data is needed to train an accurate neural network. Because we didn't find any open network log data sets for identifying pass the hash, we generate our own data set. Details are described in section 5. The evaluation of our trained neural network is presented in subsection 6.1.

For the second challenge, we design a framework as shown in Figure 2. It comprises three concurrent workflows, the data processing workflow, the tactic knowledge processing workflow, and system object dependency discovering workflow, to address the tactic matching problem.

The data processing workflow is as follows:

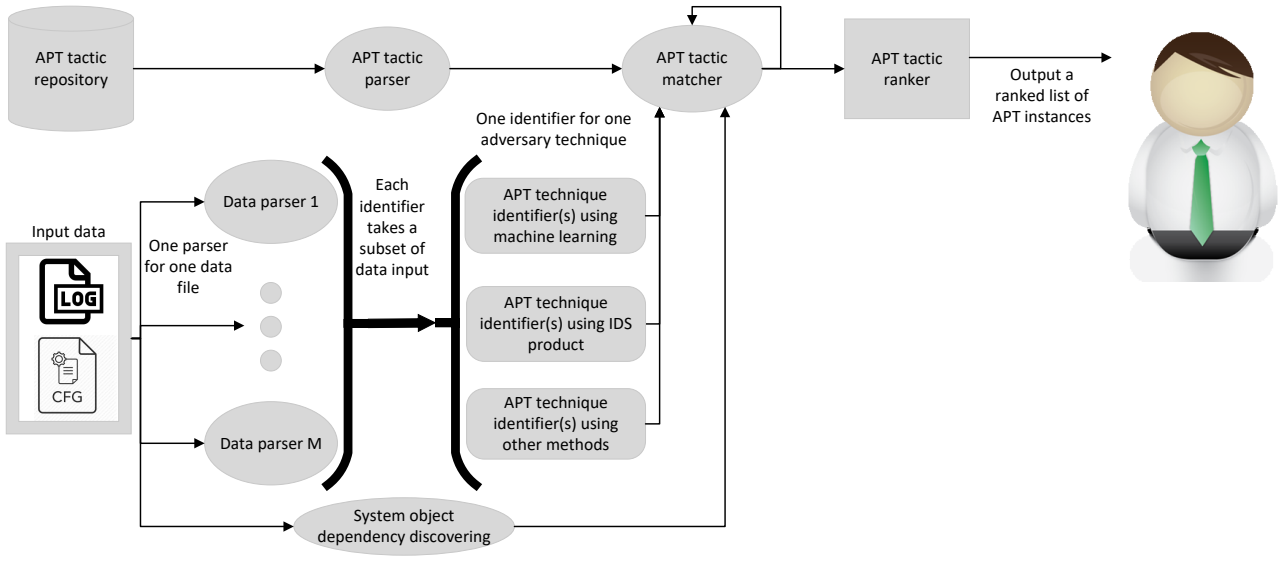

Figure 2: Proposed architecture.

(1) **Data Parsing.** The collected data sources are first fed to the corresponding data parsers.
(2) **Technique Identifying.** Based on parsed data, the APT technique identifiers determine whether certain adversary techniques exist or not.

The tactic knowledge processing workflow is as follows:

- **Tactic Parsing.** The previously seen APT tactics, which are stored in the APT tactic repository, are fed to the APT tactic parser.

The system object dependency discovering workflow is as follows:

- **Discovering system object dependencies.** A number of system logs, such as process and file I/O monitoring logs, are used to discover system object dependencies.

Finally, the results from above three workflows are taken as input for the following procedures:

(1) **Tactic Matching.** The APT tactic matcher uses the parsed tactics to match the identified adversary techniques. The matched tactics are stored as tactic instances, no matter it is fully matched or partially matched.
(2) **Tactic Ranking.** All APT tactic instances are ranked based on completeness of tactic matching.

4 DESIGN AND IMPLEMENTATION

The framework is implemented in a Ubuntu 18.04 virtual machine (referred to as detector). To isolate the detector from the IT system being monitored, it is setup as an HTTP server to receive file upload. On receiving files upload, the files are put to specific directories based on its type, such as APT tactic files, network log files or windows event log files. Other daemon programs, which are configured to monitor those directories, will trigger the framework to run once the directories' contents are changed. It is also possible to set the triggering to manual if the contents are changed at a high frequency. In this case, the daemon program monitoring the contents can raise alerts to notify the security analysts about the arrival of new input files. The analysts can then run the program at a preferred time point.

Data parser. On receiving system information, each file (e.g. a log file or configuration file) is assigned to the corresponding data parser for processing. Different data parsers are designed to deal with different types of input system files based on their syntaxes. Therefore, the number of data parser types is the same as the number of input system file types. The data parsers can work in parallel for faster processing speed.

APT technique identifier. The APT technique identifiers receive parsed data from data parsers and determine whether certain APT techniques are used. Each identifier is responsible for checking one technique. Based on different APT techniques, the identifiers may need data from different sources (i.e., different types of system information files). Hence, identifiers may take data from different sets of data parsers. The identifiers can be signature based, anomaly detection based, machine learning based, or other types. The output of the identifiers is data tuples made up of technique name and its post-conditions. Take the "Data exfiltration" technique in Table 2 as an example, the result output can be `("Data exfiltration", 23619, "D:\Documents\customer-list.xlsx")`, in which `23619` is the process ID (PID) on the user machine, and `"D:\Documents\customer-list.xlsx"` is the file. The malicious process with ID `23619` reads the sensitive file `"D:\Documents\customer-list.xlsx"` in this technique.

APT tactic repository. The APT tactic repository is a directory where all APT tactic files are stored. All the previously seen APT tactics are stored in the APT tactic repository in the same syntax. We use the graph description language DOT to describe these APT tactics. In APT tactic graphs, every adversary technique is defined as a box-shaped node, and directed edges denote the attack order.

Each DOT file can be easily visualized into directed graphs by tools like Graphviz [3]. In cases where other syntax (e.g. STIX [6]) are needed, a new APT tactic parser should be added correspondingly to parse tactic files in the new syntax.

Listing 1 is the DOT description of the APT tactic shown in Figure 1 (a) and Table 1. The example graph contains 3 technique nodes, 5 post-condition nodes, and 3 pre-requisite nodes. Each node definition starts with a *node ID*, and follows by node attributes such as *label* (the text to show in graph), *shape* (the node shape in graph) and *style* (the node style in graph). We use rounded boxes to denote the start and end nodes of the graph, boxes for APT techniques, triangles for post-conditions, and inverted triangles for pre-requisites. Post-conditions are presented as result system objects, and pre-requisites are presented as the relationships between system objects, such as "P1=>P2" stands for process P_2 is a child process of or the same as P_1, "P2->F1" stands for process P_2 writes file F_1, and that "F1->P3" stands for process P_3 reads file F_1.

Listing 1: DOT codes example.

```
digraph example_tactic {
    // nodes
    1[label="start",shape=box,style=rounded];
    2[label="Drive by download",shape=box];
    21[label="P1",shape=triangle];
    3[label="Process injection",shape=box];
    31[label="P2",shape=triangle];
    32[label="P3",shape=triangle];
    33[label="P1=>P2",shape=invtriangle];
    4[label="Data destruction",shape=box];
    41[label="P4",shape=triangle];
    42[label="P5",shape=triangle];
    43[label="P4=>P5",shape=invtriangle];
    44[label="P5->F1",shape=invtriangle];
    5[label="end",shape=box,style=rounded];

    // edges
    1->2->3->4->5;
    21->2;
    31->3;32->3;33->3;
    41->4;42->4;43->4;44->4;
}
```

APT tactic parser. From APT tactic DOT files, the APT tactic parser extracts information including: (a) the number of nodes in the tactics; (b) the types of these nodes; (c) the way that the nodes are connected; (d) the post-conditions and pre-requisites for each technique in this tactic. The information is used to create the initial "template" APT tactics. They are "templates" because no identified techniques have been matched into them at this time. We store the tactic templates in a multi-layer JSON-like data structure, which features pairs of name and values. The first layer presents technique node IDs in the DOT file, and the second layer presents node properties such as node name, post-conditions and pre-requisites. For example, the parsed APT tactic for the tactic in Figure 1(a) and Table 1 is presented below.

Listing 2: Parsed APT tactic example

```
{   '1': {  'name': 'start',
            'post-conditions': 'none',
            'pre-requisites': 'none'},
    '2': {  'name': 'Drive by download',
            'post-conditions': 'P1',
            'pre-requisites': 'none'},
    '3': {  'name': 'Process injection',
            'post-conditions': 'P2; P3',
            'pre-requisites': 'P1=>P2'},
    '4': {  'name': 'Data destruction',
            'post-conditions': 'P4; F1',
            'pre-requisites': 'P3=>P4'},
    '5': {  'name': 'end',
            'post-conditions': 'none',
            'pre-requisites': 'none'}}
```

System object dependency discovering. Some logs, such as process and file I/O monitoring logs, are used to discover system object dependencies, which are used for chaining APT techniques in tactic matching. For example, in Figure 3, there are three edges showing a process forks to create a child process, a process writes a file, and that a process reads a file. We use triple-element tuples to present these dependencies. The first and last element each represents a system object, such as a file or a process; and the middle element represents the operation, such as process forking or file reading/writing. Each system object element is a tuple which starts with a system object indicator that represents its type, followed by other items that vary according to the system object's type. For example, file writing in Figure 3 is presented as `(("P", 5414), "write", ("F", "D:\apt.dll"))`. The first element indicates a process with PID 5414, the second element indicates file writing operation, and the third element indicates the file `"D:\apt.dll"`. Therefore, this whole entry of system object dependency means that a process with PID 5414 writes the file `"D:\apt.dll"`.

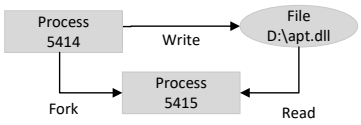

Figure 3: A simple system object dependency graph (SODG).

APT tactic matcher. The matcher takes five inputs: (a) initial "template" APT tactics from APT tactic parser; (b) APT tactic instances; (c) results from APT technique identifiers; (d) a pool of APT techniques that are identified; (e) system object dependencies. Inputs (b) and (d) are from previous run of the matcher.

Once a new APT technique is identified, the matcher will try to match it to an APT tactic if the technique is part of the tactic. Specifically, the technique can be matched to either APT tactic template(s) or partially matched APT tactic instance(s). 1) If there were partially matched APT tactic instances expecting this technique, the matcher will check the pre-requisites of the identified technique

to see if they are satisfied. The pre-requisites are checked through system object dependencies. If all pre-requisites are satisfied, the new APT technique is matched into the tactic instance, and the post-conditions are confirmed in the instance. If not, the unmatched new technique is saved to the pool and the post-conditions are discarded. 2) In many times, a given APT technique may not be matched to any partially matched APT tactic instances, but can be matched to a APT template. In this case, a new APT tactic instance is created from the template, the technique is matched into this instance, and system object(s) in the instance regarding this technique is confirmed. Otherwise, the technique is put to the pool.

Whenever an APT technique is matched into an APT tactic template/instance, the matcher goes through all techniques in the pool and start another iteration of the matching process, but this time, the matcher only tries to match the technique into partially matched tactic instances, because those techniques have already been checked against templates before put into the pool. In this way, even if a technique is not matched at the time of identification, it will still be matched once all the conditions are met afterwards.

To illustrate the algorithm, we present two examples of matching based on the tactic presented in Figure 1(a) and Table 1. Firstly, consider the case where no techniques are matched into the tactic. Assuming the identifier first reports that drive by download and the corresponding process are identified. The matcher first assumes that the post-condition P_1 is the process reported, and then checks the pre-requisites. The matcher finds out that this technique has no pre-requisites, so the technique gets matched immediately. An APT tactic instance gets created from this template, the drive by download technique in it is marked as matched, and P_1 is confirmed to be the process reported in this APT tactic instance.

In the second example, consider the case where no techniques are matched into the tactic. However, technique process injection gets identified first. Similarly, the matcher assumes the system objects reported are correct, and then checks the pre-requisites against system object dependencies. The matcher finds out that it needs P_1 and P_2, but P_1 from the previous technique is not matched yet. Therefore, the matcher decides that this technique cannot be matched and put it in the pool of unmatched techniques. After drive by download is matched, during the going through of the pool, this previously unmatched technique, process injection, will be matched if it meets the pre-requisites.

In real world, defense against APT tactics generally needs to tackle three scenarios: one attacker is using one tactic; multiple attackers are using one same tactic; multiple attackers are using different tactics. Our matcher can handle all these scenarios properly. (a) If one attacker is using one tactic, the algorithm shown in Algorithm 1 can detect it by matching every step. (b) If multiple attackers are using one same tactic, they can be differentiated by system object dependencies, such as different process ID, which is presented as pre-requisites (Table 1) used in line 6 of the algorithm. (c) If multiple attackers are using different tactics, they can be differentiated by matching different techniques as shown in line 2 of the algorithm. A more complicated situation is that multiple attackers are using different APT tactics, but these tactics share some techniques in common. At the defender's side, it is unknown which attacker is using which APT tactic, so the best practice is to list all the possible combinations. In light of this, the matcher will create APT tactic instances for all combinations while preserving the system object dependencies.

Algorithm 1 APT tactic matching algorithm

1: **Input**: APT tactic templates; APT tactic instances; one newly identified adversary technique *new_at*; a pool of identified techniques.
2: *all_candidates* ← same techniques as *new_at* found in APT tactic templates and instances.
3: **if** there exists at least one technique in *all_candidates* **then**
4: **for all** *candidate* ∈ *all_candidates* **do**
5: Assuming post-conditions are right, examine pre-requisites for each *candidate*.
6: **if** All pre-requisites are met for the *candidate* **then**
7: **if** *candidate* is in a template **then**
8: Create a new instance from this template, and match *new_at* into the position of *candidate*.
9: **else if** *candidate* is in an instance **then**
10: Create a copy of *candidate*.
11: Match *new_at* into the position of the copy of *candidate*.
12: **end if**
13: In the instance where *new_at* is matched into, identify the next adversary technique *next_at*.
14: *all_next_step_candidates* ← same techniques as *next_at* found in the pool of not matched adversary techniques.
15: **for all** *next_step_candidate* ∈ *all_next_step_candidates* **do**
16: **Call** APT tactic matching algorithm.
17: **end for**
18: **end if**
19: Put *new_at* into the pool
20: **end for**
21: **else**
22: Save *new_at* to the pool of not matched adversary techniques.
23: **end if**

APT tactic ranker. The ranker takes the APT tactic instances from matcher as input. The ranking is based on the percentage of APT tactic instances' completeness. The more completely an APT tactic instance is matched, the higher it will appear on the list. Fully matched APT tactic instances are put on top of the list.

The ranker ranks both fully matched and partially matched APT tactic instances. Assuming that the technique identifiers can correctly identify APT techniques, the existence of partially matched APT tactic instances means that either the attacker gives up this campaign, or that the attacker just decides to wait before launching remaining techniques. Therefore, partially matched APT tactics should be kept and ranked but not discarded.

In a word, the framework's workflow can be summarized as input parsing, technique identifying, tactic matching and tactic ranking. Except input parsing phase, the other three phases generate new findings at three time points respectively, and new findings at a

previous time point should lead to new findings at the next time point.

- The first time point is when a technique is identified at technique identifying phase. The new findings here are the identified techniques and their post-conditions.
- The second time point is at the end of tactic matching phase. The new findings here are are new and/or updated APT tactic instances.
- The third time point is at the end of tactic ranking phase. The new findings here are updated APT tactic instances' completeness and ranking results.

Thus, at the end of technique identification phase, if new technique(s) is identified, the tactic matcher should be automatically triggered; and at the end of tactic matching phase, if new APT tactic instance(s) is created or previous instance(s) is updated, the tactic ranker should be automatically triggered.

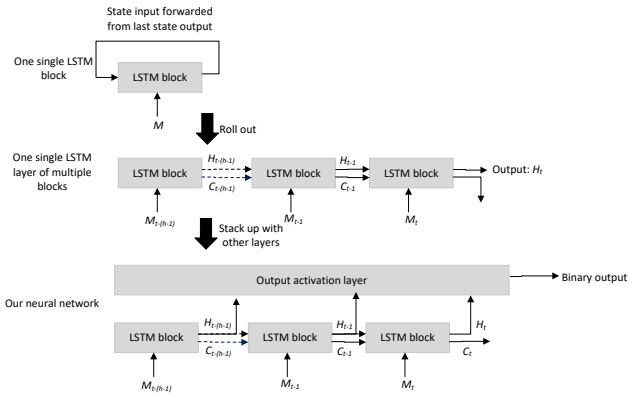

Figure 4: Our LSTM neural network.

5 CASE STUDY

The APT tactic shown in Figure 1(b) and Table 2 contains five different APT techniques. In this section, we use this tactic as a case study and describe how each APT technique can be identified, following the order they appear. We will discuss the identification of Pass the Hash in detail, as it is an example APT technique that we can apply machine learning for its identification.

Supply Chain Compromise. This technique can be identified by scanning downloaded item with Anti-Virus (AV) products. Alternatively, network intrusion detection products like Snort [4] can also detect it when the downloaded item tries to establish communication with the attacker's machine. Windows Defender can identify this technique and leave one entry in Windows event logs. The post-condition of this step is a malicious process P_1 is created due to attackers' communication with the victim machine. As an initial intrusion step, it has no pre-requisites.

Bypassing Windows User Account Control (UAC). Bypassing Windows UAC is a common privilege escalation technique towards Windows machines. Many procedures have been discovered to bypass UAC. An extensive list of available procedures can be found in the UACMe project [2]. Some procedures rely on modifying specific, user-accessible Registry entries. Therefore, by monitoring accesses to the Registry with process monitor, especially entry creation and modification, this APT technique can be identified. The post-conditions are that processes P_2 and P_3 are created, with P_3 running at system-level privilege. P_2 may act as a middle stage. In this case, P_2 may be the actual process modifying the Registry. After that, a system service reads the modified Registry and created the process P_3 with system-level privilege. Therefore, P_2 and P_3 may not have direct relation, and the pre-requisite only has requirement on P_2.

Credential Dumping. Credential dumping is the process of obtaining account credentials, normally in the form of account names and password hashes. On a Domain Controller (DC), Active Directory (AD) service maintains all the users' account names and password hashes in the domain. They are stored in a database file locked by AD. Only some specific processes are allowed to access the contents of this file. Local Security Authority Subsystem Service (LSASS) is one of such special processes. It is given the right to read the database file because its role in a DC is to provide an interface for managing local security, domain authentication, and AD processes. With escalated privilege, attackers can craft queries, send them to LSASS, and get domain user account names and corresponding hashes. Therefore, we can monitor the LSASS process to detect this APT technique. We can either match some signatures in malicious queries or detect anomaly behaviors of LSASS. The post-condition is that process P_4 that runs with system-level privilege and interacts with LSASS is created. The pre-requisite is that P_4 is a child process of or the same as P_3, because a process needs system-level privilege to interact with LSASS.

Pass the Hash. Pass the hash is a well-known technique for lateral movement. In remote login, plain-text passwords are usually converted to hashes for authentication. Some authentication mechanisms only check if hashes are matched. Pass-the-hash technique relies on these vulnerable mechanisms to impersonate a normal user with dumped hashes. We assume that a) normal users use benign client programs that are usually authenticated through other mechanisms, and that b) attackers cannot get the plain-text passwords and have to rely on hashes to impersonate a normal user. We can capture the network packets and find out which kind of authentication mechanism is used. The login session that uses those vulnerable authentication mechanisms can then be identified as pass the hash attack. The post-condition is that a malicious process P_5 with user U_1's credential is created. This step has no pre-requisites.

There are three stages during the remote login session. Each stage contains multiple network packets. For example, the second stage, authentication, can be viewed as a sequence made up of client's authentication request, server's challenge, client's challenge response and server's authentication response, as shown in Figure 5. The client first sends a session setup request to the server; then the server responds to the client with a challenge; on receiving the challenge, the client uses the challenge and credentials to do calculations and sends back the result in challenge response packet; finally, the server verifies the result and sends back authentication response indicating whether authentication succeeds or not.

No.	Source	Destination	Protocol	Length	Info
26	172.29.36.125	172.29.151.231	SMB2	239	Session Setup Request, NTLMSSP_NEGOTIATE
27	172.29.151.231	172.29.36.125	SMB2	435	Session Setup Response, Error: STATUS_MORE_PROCESSING_REQUIRED, NTLMSSP_CHALLENGE
28	172.29.36.125	172.29.151.231	SMB2	576	Session Setup Request, NTLMSSP_AUTH, User: CORP\Administrator
30	172.29.151.231	172.29.36.125	SMB2	151	Session Setup Response

Figure 5: A subset of network packets during pass the hash attack.

Pass the Hash Data Generation. The raw data (network packets) are automatically generated by protocol fuzzing. During a pass the hash attack, before a packet is sent to the server, we fuzz certain fields in the application layer (SMB/SMB2 for pass the hash) of the network packets. In this way, (a) the packet structure remains intact, so that the server will not discard the packet; (b) the authentication of pass the hash can be affected; (c) we can get a variety of network packets, and possibly, a variety of network packet sequences so we can get enough diversity in the data for machine learning. The same fuzzing method has also been applied in the generation of benign data from normal network traffic. All the network packets from malicious and benign network traffic are captured using Wireshark. Because of fuzzing, we cannot assure every pass the hash attempt or normal access attempt is successful. For failed pass the hash attempts, we remove them from malicious data. The reason is that, if it fails, the attempt does not generate any impact, so it is not really malicious. For failed normal access, we keep it in benign data, because normal user can also have typos or forget passwords.

Though the raw network logs contain network packets from both malicious and benign network traffic, there are too much redundant data that do not carry useful information for identifying pass the hash, such as timestamp. There are also fields that have fixed values, such as the header length for SMB/SMB2. Values of these two types of fields cannot help identifying pass the hash. Therefore, in the data parser for network logs, such information is removed. Only those that can help identify pass the hash are kept.

Pass the Hash Identification. To identify pass the hash with machine learning, we have two key insights:

- Network communications consist of lots of network packets sent in certain order. What happens at a previous time point can affect what happens afterwards. For example, the first several packets may be a server and a client communicating the protocol to use, and packets afterwards will use the protocol decided;
- Pass the hash relies on certain authentication mechanism to work. Therefore, there must be some differences in certain values in at least one network packet during the network communication.

With these two insights, we decide to build an LSTM neural network that takes a sequence of network packets' types, and outputs the binary label representing whether this sequence is pass the hash network traffic. We have trained a long short-term memory (LSTM) neural network for pass-the-hash identification based on network packets sent between the server (victim) and the client (attacker) during remote logins. Our neural network is presented in Figure 4. M stands for input; H stands for each LSTM block's output; and C stands for each LSTM block's state output. Subscripts stand for the time points, in which h stands for the window size. The network packets' types are defined by the values of fields of interest. If two packets have the same values in all those fields, then the two packets are presented as the same packet type number. Otherwise, different numbers are assigned. In this way, a sequence of network packets can be presented as a sequence of numbers, each standing for a packet of its corresponding packet type.

Now that each network packet is represented as a packet type number, the whole network log can be represented as a whole sequence of packet type numbers. By identifying the start packet for each benign/malicious network communication, we chomp the whole sequence into many variate-length sequences, and the beginning of every sequence is a start packet. Then, we chomp each of those variate-length sequences into one or more fixed-length sequences according to the window size and window shift step size. Depending on whether the sequence comes from a benign traffic or malicious traffic, we assign every fixed-length sequence with the corresponding binary label and get one data sample for our LSTM neural network. If a fixed-length sequence appear in both the benign and malicious data, this sequence is removed from both of them because it cannot help the classification. We then remove duplicate sequences in both the benign data and malicious data. After these two removal processes, all data samples are finally ready to be used.

Data Exfiltration. This technique can be very confidential. It can be encrypted and done through very common protocols like HTTP/HTTPS. As a result, data exfiltration is hard to be identified at the network level. Supposing that attackers are interested in sensitive data, system administrators can enforce disk I/O monitoring with process monitor on sensitive files/folders on a machine. In this way, data exfiltration can be identified at disk level.

6 EVALUATION

At high level, the framework presented can be separated into four phases:

- **Input parsing.** The raw input files are parsed by APT tactic parser, data parsers and system object dependency.
- **APT technique identifying.** APT identifiers use parsed inputs to identify APT techniques.
- **APT tactic matching.** With the help of system object dependencies, the APT tactic matcher matches identified APT techniques into APT tactics.
- **APT tactic ranking.** After tactic matching, the APT tactic ranker ranks all APT tactic instances based on completeness.

We have evaluated each phases and the results show that our framework can correctly detect APT tactics and pick out the one(s) that is fully matched. Specifically, we answer the following questions:

- **RQ1.** How accurately can technique identifiers identify APT techniques?

Table 3: 384 APT tactics in the repository.

Initial intrusion	Privilege escalation	Credential access	Lateral movement	Impact
Supply chain compromise	Bypass User Account Control	Credential dumping	Pass the hash	Data exfiltration
Exploit public-facing application	DLL search order hijacking	Account manipulation	Logon scripts	Data manipulation
External remote services	New service	Private keys		Data destruction
Spearphishing link	Process injection			Endpoint denial of service

- **RQ2.** Given identified APT techniques, how correctly can the matcher match them into APT tactics and generate instances?
- **RQ3.** Given matched APT tactic instances, can the ranker rank the fully matched APT tactic instances higher than others in the ranked list?
- **RQ4.** How much time and memory does the framework need for each phase?

For evaluation experiments, we first prepared 385 APT tactics in the APT tactic repository. One is a 4-step tactic referred to as tactic001 in Table 4. The other 384 are 5-step tactics presented in Table 3. Each tactic can take any one process belonging to the same technique (column). Therefore, Table 3 provides $4*4*3*2*4 = 384$ APT tactics. Some APT tactics may be very similar at high level. For example, replace one technique in tactic A and get tactic B. The old technique and the new technique are for the same purpose, but done in different ways (e.g. lateral movement by logon scripts or pass the hash). We treat A and B as different APT tactics because different techniques need different technique identifiers to identify them, and those identifiers may use different method and/or system object dependency for identifying.

Out of the 385 APT tactics in the repository, we launched 7 of them in evaluation experiments, presented in Table 4. 3 kinds of logs from those 7 APT tactics are collected, which are network logs (captured by Wireshark), process monitor logs (exported from Prcess Monitor [15]), and windows event logs (exported from Windows Event Viewer). Our test bed, towards which attacks are launched, consists of one Windows AD DC (Windows Server 2012 R2) virtual matching hosting a Windows domain, joined by another Windows 7 virtual machine. The raw input files include 385 APT tactic DOT files, 5.84GB of network logs, 12.6GB of process monitor logs, and 141MB of Windows event logs.

6.1 APT technique identifying

For APT technique identifying, we focus on evaluating one specific technique, which is pass the hash identification. We choose this because other identifiers involve commercial IDS or manually crafted patterns, which have little point in evaluating.

For pass the hash identification, we build an LSTM based neural network. It takes parsed network log files of packets sent between hosts as input, and produce binary results showing whether pass the hash attack is presented in a packet sequence.

With different parameters such as training batch size, LSTM window size and window shift step size, we have trained a total of 144 neural networks. Of all the data fed to the neural network, about 60% are used for training, about 20% are used for validation, and the rest about 20% are used for testing. Every trained neural network is evaluated by false positive rate, false negative rate and F1 score. With different parameters, the number of benign data samples and malicious data samples can also change accordingly. As a result, whether the final dataset is balanced or not is kind of unpredictable, so accuracy is not very helpful. What is more, the true negative samples, which are noises, are not of interest for us, and F1 score does not take true negatives into calculation. Therefore, we choose F1 score as the main criteria.

The best-performing neural network, which has the highest F1 score of 0.9763 on test set, is fed with 2085 benign data samples and 2830 malicious data samples. The false positive rate and false negative rate on the test set of this neural network are 4.437% and 0.252%, respectively. Though the false positive rate is not ideal, we will show that, in tactic matching phase, those false positives cannot produce fully matched APT tactic instance.

Result 1: *The LSTM-based APT technique identifier for pass the hash, which is hard to detect via traditional methods, can identify the technique with low false negative rate, but the false positive rate is not ideal.*

6.2 APT tactic matching

As stated earlier, the number of launched APT tactics in our experiments is small. However, even if there are very few APT tactics happening, the CSOCs need to be aware of all possible APT tactics. With this insight, we evaluated the APT tactic matcher with all the 385 APT tactics in the repository. The goal of evaluating APT tactic matcher is to see whether it can correctly and fully match APT tactic that is actually launched in the following three cases: (a) one attacker is using one APT tactic; (b) multiple attackers are using one same APT tactic; (c) multiple attackers are using multiple different APT tactics. There may be some partially matched APT tactics. This is natural because some APT tactics share the same technique at some point, but the APT tactic matcher just tries its best to match an APT technique into tactic. How much importance should be given to the tactic instance is not evaluated by the APT tactic matcher, but the APT tactic ranker, which will be evaluated in the next subsection.

Case A: One attacker using one APT tactic. In this scenario, tactic001 presented in Table 4 is launched for once in our test bed. The matcher successfully match the target tactic in full. The outputs contains 1 fully matched APT tactic instance of the target tactic and other 123 partially matched instances.

Case B: Multiple attackers using one same APT tactic. In this scenario, tactic001 presented in Table 4 is launched for three times with some different parameters, like PIDs and file names. The

Table 4: A list of APT tactics that were launched.

APT tactic name	1st technique	2nd technique	3rd technique	4th technique	5th technique
tactic001	Supply chain attack	DLL search order hijacking	Logon scripts	Data modification	(None)
tactic002	Supply chain attack	Bypass User Account Control	Credential dumping	Pass the hash	Data exfiltration
tactic003	Supply chain attack	Bypass User Account Control	Credential dumping	Pass the hash	Data modification
tactic004	Supply chain attack	Bypass User Account Control	Credential dumping	Pass the hash	Data destruction
tactic005	Supply chain attack	DLL search order hijacking	Credential dumping	Pass the hash	Data exfiltration
tactic006	Supply chain attack	DLL search order hijacking	Credential dumping	Pass the hash	Data modification
tactic007	Supply chain attack	DLL search order hijacking	Credential dumping	Pass the hash	Data destruction

matcher outputs 9 fully matched APT tactic instances of the target tactic and 375 partially matched instances.

The results show the existence of "duplicates" in fully matched APT tactic instances. The reason behind is that some techniques in the tactic instances can be replaced with others. In Case B, the last two techniques of tactic001 are interchangeable among the 3 attacks. When we launched the 3 attacks, the vulnerable process for DLL search order hijacking remains the same, which made the last two techniques interchangeable. As a result, the final output of fully matched APT tactic instances becomes $3 * 3 = 9$. This is reasonable because at the defender's side, CSOCs have no idea which attacker is aiming for what in their IT system, so the best practice is to list all possible combinations.

Case C: Multiple attackers using multiple different APT tactics. In this scenario, the APT tactics presented in Table 4 are each launched for once. Note that these APT tactics share some common techniques. The matcher outputs 94 fully matched APT tactic instances and 5468 partially matched instances. The 94 fully matched APT tactic instances include 4 instances of tactic001, 6 of tactic002, 6 of tactic003, 6 of tactic004, 24 of tactic005, 24 of tactic006, and 24 of tactic007.

The similar thing in Case B also happens to Case C. In Case B, the interchangeability results from three attackers using one same tactic; in Case C, the interchangeability results from shared APT techniques among the 7 tactics. For tactic001, its first two techniques (supply chain attack and DLL search order hijacking) are interchangeable among tactic001, tactic005, tactic006, and tactic007, so its fully matched instance number is 4. For tactic002, tactic003 and tactic004, their last two techniques (pass the hash and data exfiltration/modification/destruction) are interchangeable between tactic002 and tactic005, tactic003 and tactic006, and tactic004 and tactic007 respectively. The reason is that pass the hash technique has no prerequisites, so there is no way to chain the first three techniques together with the last two. As a result, the numbers of fully matched instances for tactic002, tactic003 and tactic004 are all $3 * 2 = 6$. For tactic005, tactic006 and tactic007, the first two, the middle one, and the last two techniques are all interchangeable. How the first two and the last two techniques are interchangeable have been discussed earlier; and the middle one technique (credential dumping) is interchangeable among tactic005, tactic006 and tactic007 because they use the same vulnerable process for DLL search order hijacking. Thus, their fully matched instance numbers are all $4 * 3 * 2 = 24$.

One thing worth noting is that after the technique identifying phase, we blindly feed identification results to tactic matching phase. This means that the false positives of pass the hash identification are treated as true positives and used for tactic matching. CSOCs may realize that some of those are false positives, but at the run time, without further inspecting the data, they cannot know that whether the outputs of pass the hash identifier contains false positives or not. In spite of this, the number of fully matched APT tactics is still in consistence with our expectations, which means that, during tactic matching, those false positives cannot be used to produce fully matched APT tactic instances because they do not meet the pre-requisites of system object dependencies.

To further assure the matcher's resilience to false positives from technique identifying, we carry out an additional experiment. We reproduce Case C, but, this time, we add a filter between the pass the hash identifier and the APT tactic matcher, so that the matcher only gets false positives from pass the hash identifier. Other parts remain the same, and we find out that the matcher only outputs 4 fully matched APT tactic instances of tactic001, and 2126 not fully matched APT tactic instances. Therefore, the APT tactic matcher is resilient to the false positives from technique identifiers.

Result 2: *The APT tactic matcher is resilient to false postives from technique identifiers and can correctly match identified APT techniques into tactics and create APT tactic instances in all three cases.*

6.3 APT tactic ranking

After APT tactic matching, the matcher outputs APT tactic instances, which can be partially of fully matched, to the APT tactic ranker. The APT tactic ranker then ranks these APT tactic instances by completeness. The goal of evaluating APT tactic ranker is to see whether it can correctly pick out fully matched APT tactic instances and put them to the top of the list.

The evaluation results show that, in all of the three Cases A, B and C, the ranker successfully ranks the fully matched instance(s) to the top. In Case B and C, because there are many "duplicate" APT tactic instances with the same 100% completeness, there can be many APT tactic instances, which are the same APT tactics with different system objects, on the top of the ranking list.

Result 3: *The APT tactic ranker can rank the fully matched APT tactic instances on top of the ranked list.*

6.4 Time and memory usage

To evaluate the efficiency of our framework, we have conducted offline measurements of the time and max memory usage for each of the four phases. The machine used for this experiment is a workstation with Intel(R) Xeon(R) E5-2650 v3 processor and 62GB of RAM. The results are presented in Table 5.

Table 5: Max memory and time taken for each phase.

Phase		Max memory taken (MB)	Time taken (s)
Input parsing		38794	1366
APT technique identifying		3812	74
APT tactic matching	Case A	19	1
	Case B	20	1
	Case C	51	13
APT tactic ranking	Case A	16	1
	Case B	17	1
	Case C	49	1

It is shown that, the most resource-consuming phase is the input parsing phase. In this phase, the memory usage can be as high as about twice the size of input files. The memory usage is high because in our Python implementation, we used the data structure of lists a lot. In Python, lists have many redundancies between adjacent elements. To make matters worse, Python does not have any means for memory recycle, which means once some memories are used, they will never be released, until the whole program exits. Therefore, in our Python input parsing implementation, after loading large files into the memories, the program cannot release them after the files' parsing have been finished. If the framework is implemented with other programming languages, like Java, the memory can be better managed.

Result 4: *The most resource-consuming phase is input parsing. Other phases consumes acceptable memories and short time. The memory usage and time consumption have the potential to be further reduced.*

7 CONCLUSION AND FUTURE WORK

In this paper, we propose a framework for detecting APT tactics from logs and configuration files. The framework takes previously seen APT tactics, logs and system configuration files as input, and generates a ranked list of APT tactics based on completeness. We also present a detailed case study of a simple 5-step APT tactic, describing how to identify each APT technique in the tactic. Finally, we present the evaluation results of our framework, which clearly shows that the framework can correctly detect APT tactics.

Currently, we implement the framework on one virtual machine. To further improve the efficiency, we plan to implement, validate and evaluate this framework in a cloud environment with multiple virtual machines. Every virtual machine will be dedicated to one task for efficiency. Manual work can be significantly reduced because system administrators only need to care about feeding data into the cloud. The other workloads, including file parsing, technique identifying, and tactic matching will be all completed by the framework automatically.

Our framework matches APT tactics from the repository. However, attackers may update their attack tactics. They may replace old adversary techniques for new ones or add/remove techniques according to their purposes. In the future, we plan to add another component, APT tactic updater, to our framework to automatically handle tactic updating. The updater will take tactic templates from tactic parser, partially matched tactic instances and the pool of unmatched adversary techniques from tactic matcher. Then it can decide when some changes should be made in original tactics, how it should be made, and finally update tactics in the repository.

Another thing worth noticing is that, we simply rank the APT tactics instances based on the completeness. The ranking could be based on more complicated algorithm. For example, CVSS scores can be taken into consideration, and an "impact score" can be calculated for each APT tactic instance. Or another model may be proposed to assess how likely the APT tactic is being used by the attacker. Because this is about APT impact assessment and is out of the scope of our paper, here we do not dig deeper into the ranking problem.

DISCLAIMER

This paper is not subject to copyright in the United States. Commercial products are identified in order to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the identified products are necessarily the best available for the purpose.

REFERENCES

[1] [n. d.]. Evolving Playbooks in Targeted APT Attacks across Asia Pacific and Japan - Security Boulevard. https://securityboulevard.com/2018/05/evolving-playbooks-in-targeted-apt-attacks-across-asia-pacific-and-japan/

[2] [n. d.]. GitHub - hfiref0x/UACME: Defeating Windows User Account Control. https://github.com/hfiref0x/UACME

[3] [n. d.]. Graphviz - Graph Visualization Software. https://www.graphviz.org/

[4] [n. d.]. Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/

[5] 2019. MITRE ATT&CK™. https://attack.mitre.org [Online; accessed 27. Mar. 2019].

[6] Sean Barnum. 2014. Standardizing cyber threat intelligence information with the Structured Threat Information eXpression (STIX™). *MITRE Corporation* (2014), 1–20.

[7] Saranya Chandran, P. Hrudya, and Prabaharan Poornachandran. 2015. An efficient classification model for detecting advanced persistent threat. In *2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015*. IEEE, 2001–2009. https://doi.org/10.1109/ICACCI.2015.7275911

[8] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1285–1298.

[9] Ibrahim Ghafir, Khaled Rabie, Liangxiu Han, Vaclav Prenosil, Francisco J. Aparicio-Navarro, Robert Hegarty, and Mohammad Hammoudeh. 2018. Detection of advanced persistent threat using machine-learning correlation analysis. *Future Generation Computer Systems* 89 (2018), 349–359. https://doi.org/10.1016/j.future.2018.06.055

[10] Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, and Scott Stoller. 2017. SLEUTH : Real-time Attack Scenario Reconstruction from COTS Audit Data. *Proceedings of the 26th USENIX Security Symposium* (2017), 487–504.

[11] Pavlos Lamprakis, Ruggiero Dargenio, David Gugelmann, Vincent Lenders, Markus Happe, and Laurent Vanbever. 2017. Unsupervised detection of APT C&C channels using web request graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10327 LNCS. 366–387. https://doi.org/10.1007/978-3-319-60876-1_17

[12] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *NDSS*.

[13] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2017. {MPI}: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 1111–1128.

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

[15] markruss. 2019. Process Monitor - Windows Sysinternals. https://docs.microsoft.com/en-us/sysinternals/downloads/procmon [Online; accessed 26. Aug. 2019].

[16] Sadegh M. Milajerdi, Rigel Gjomemo, Birhanu Eshete, R. Sekar, and V. N. Venkatakrishnan. 2018. HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows. *2019 IEEE Symposium on Security and Privacy (SP)* (2018). arXiv:1810.01594

[17] Alina Oprea, Zhou Li, Robin Norris, and Kevin Bowers. 2018. MADE: Security Analytics for Enterprise Threat Detection. *ACSAC* (2018). https://doi.org/10.1145/3274694.3274710

[18] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*. ACM, 583–595.

[19] Joseph Sexton, Curtis Storlie, and Joshua Neil. 2015. Attack chain detection. *Statistical Analysis and Data Mining* (2015). https://doi.org/10.1002/sam.11296

[20] Sana Siddiqui, Muhammad Salman Khan, Ken Ferens, and Witold Kinsner. 2016. Detecting Advanced Persistent Threats using Fractal Dimension based Machine Learning Classification. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics - IWSPA '16*. ACM Press, New York, New York, USA, 64–69. https://doi.org/10.1145/2875475.2875484

[21] G Zhao, K. Xu, L Xu, and B. Wu. 2015. Detecting APT malware infections based on malicious DNS and traffic analysis. *IEEE Access* 3 (2015), 1132–1142. https://doi.org/10.1109/ACCESS.2015.2458581