Automatic Inference of Application Reputation via PageRank on System Monitoring Data

Pengcheng Fang
Case Western Reserve University

Xusheng Xiao
Case Western Reserve University

Lingming Zhang
University of Texas at Dallas

1. ABSTRACT

Advanced Persistent Threat(APT) attacks have plagued even well-protected businesses with significant financial losses. These attacks are complicated and stealthy, which can remain undetected for years. Therefore, enterprises are seeking solution to reconstruct the multiple steps of these sophisticated attack. We purpose a solution to automatically reconstruct the steps of APT. This solution includes: (1) ubiquitous system monitoring for collecting the attack provenance for a long period of time and (2) dependency analysis for causally connecting the dots of multiple steps of APT and (3) reconstruct the attack step from the dependency analysis. Although there are several kernel audit frame for Linux and Windows and algorithms to reconstruct the dependency graph based on the audit frame output, it still relies heavily on security analysts to reconstruct steps of APT. The main contribution of this project is that adopting PageRank to requirement(3) make it is possible to automatically reconstruct the process of APT.

Keywords

System Monitoring; GraphSplit; Dependency Graph

2. INTRODUCTION

Advanced Persistent Threat (APT) attacks[6, 20] have caused many well-protect firms with significant financial losses[5, 4, 22, 15] and become the reason for people and companies seek safer and more comprehensive information security protections. Yahoo data breach leaked over 1 billion accounts' sensitive information. This security event caused Verizon to cut the price of the deal with Yahoo by 350 million[3, 20]. These APT attacks often consist of many individual attack steps across many hosts and leverage advanced tools and malware to penetrate into an enterprise[6, 20]. These complex strategies and customized tools make APT hard to be prevented. Even though these attacks can be powerful and stealthy, one typical constrains from the attacker side is that such an attack consisted by multiple steps could leave the multiple attack footprints as "dots".

In order to achieve the version of connecting the dots, the first challenge is to collect and store the dots. This goal can be achieved by the system kernel listening tool which can gather and output the information of system calls and provides a comprehensive way to capture system behaviors[10, 11]. Unlike its alternatives (file accesses, firewall and network monitoring) that provide partial information and are application-specific, system monitoring covers all activities among system entities(process, files, sockets, and pipes) over time, providing a global view of system activities.

The second challenge for achieving the vision of connecting the dots is to connect dots even interleaved with multiple system activities. The state-of-art[7, 10, 11] techniques take processes as subjects and other system entities as objects. System call events happen between system entities establish edges between subjects and objects. If an anomalous is detected, a dependency analysis is applied to connected system call events via causality. In order to finish the dependency analysis quickly, an efficient design storage and index of log data is necessary.However, APT attacks could remain stealthy for more than half a year [2], and monitoring attack provenance on every host in an enterprise for such a period of time (˜0.5–1.0 GB per host per day) is burdensome and laborious. Therefore, gathering and storing data for dependency analysis in an enterprise is an daunting task. To mitigate the problem of overwhelmingly big-data of attack derived from system monitoring data for dependency analysis, the Backtracking and Causality Preserved Reduction(CPR) are applied to shrink the size of dependency graph.

The logistics behind the Backtracking is that the events that happened after the anomalous event can not be part of multiple attack steps of APT, and therefore these events can be removed from the dependency graph. The insight of Causality Preserved Reduction is that some events have identical dependency impact space(same subject and same objects), and therefore they can be safely aggregated.

The third challenge is that although dependency analysis reveals the causality of an anomalous event as a dependency graph, they only scratch the surface of how causality analysis can be used to enhance the defense of APT attacks: it is still slow and laborious for security analysts to filter unrelated events and recover the attack sequence from the large dependency graph. In APT attacks, many steps require downloading and installing customized applications and libraries to enterprise system to succeed, such as malware injection and privilege escalation[8, 18, 21] and blocking applications and libraries from entering the enterprises can counter these

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

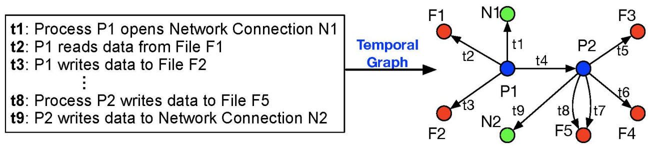

t1: Process P1 opens Network Connection N1
t2: P1 reads data from File F1
t3: P1 writes data to File F2
⋮
t8: Process P2 writes data to File F5
t9: P2 writes data to Network Connection N2
Temporal Graph
F1
N1
F3
P1
P2
F2
N2
F5
F4
t1
t2
t3
t4
t5
t6
t7
t8
t9

Figure 1: System monotoring Data

attacks. In this project, PageRank[16] is used to infer reputation of all system entities. The libraries or applications from unknown source has lower reputation compared with those from reliable channels, such as Google, Apple, Samsung and so on. After that, we can reconstruct steps based on each entity's reputation.

3. BACKGROUND

3.1 System Monitoring

System monitoring data consists of various system activities in the form of events along with time [10, 11, 7, 9]. Each event can naturally be described as a system entity (subject) does some operation on another system entity(object). For example, a process reads a file or a process accesses a network connection. An APT attack needs multiple step to succeed , such as target discovery and data exfiltration, as illustrated in the cyber kill chain[1]. Therefore, multiple attack footprints might be left as "dots", which can be captured precisely by system monitoring. System monitoring data for Windows and Linux can be collected via ETW event tracing[13] and Linux Audit Framework[17]. The audit framework used in this project is a popular and opensource audit application: Sysdig.

3.2 Sysdig

Sysdig is a popular system and container monitoring tool. It improve the system and container visibility, combined with Kubernetes, Docker and Mesos integration, better record enterprises's services. It gives system manager an easy access to the actual behavior of the system and container. Far too often, system-level monitoring still involves logging into a machine with SSH and using plethora of dated tools with very inconsistent interfaces. Sysdig instruments physical and virtual machine at the operating system level by installing into the Linux kernel and capturing system calls and other system events.

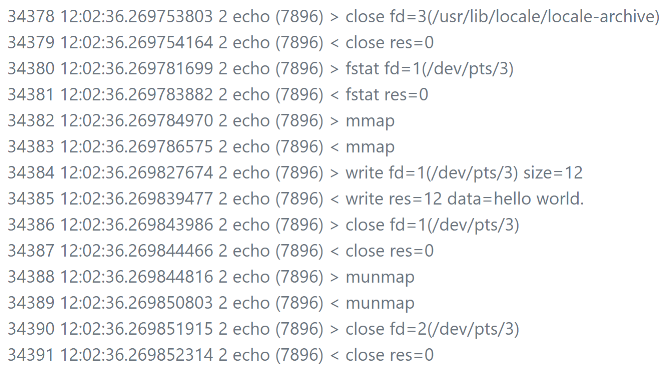
3.3 Output data description

The raw output data already includes many useful information, however it is still not enough to build the dependency graph of all the system behaviors. According to the Sysdig user guide, the output data format is defined as:

```
%evt.num %evt.rawtime.s.%evt.rawtime.ns %evt.cpu
%proc.name (%proc.pid) %evt.dir %evt.type cwd=%proc.cwd
%evt.args latency=%evt.latency
```

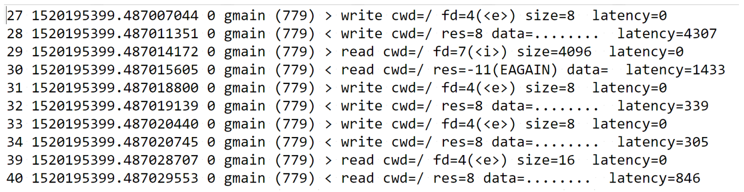
where:

- evt.num is the incremental event number.
- evt.rawtime.s is second part of the event timestamp.
- evt.rartime.na is nanosecond part of the event timestamp.
- evt.cpu is the CPU number where the event was captured



```
34378 12:02:36.269753803 2 echo (7896) > close fd=3(/usr/lib/locale/locale-archive)
34379 12:02:36.269754164 2 echo (7896) < close res=0
34380 12:02:36.269781699 2 echo (7896) > fstat fd=1(/dev/pts/3)
34381 12:02:36.269783882 2 echo (7896) < fstat res=0
34382 12:02:36.269784970 2 echo (7896) > mmap
34383 12:02:36.269786575 2 echo (7896) < mmap
34384 12:02:36.269827674 2 echo (7896) > write fd=1(/dev/pts/3) size=12
34385 12:02:36.269839477 2 echo (7896) < write res=12 data=hello world.
34386 12:02:36.269843986 2 echo (7896) > close fd=1(/dev/pts/3)
34387 12:02:36.269844466 2 echo (7896) < close res=0
34388 12:02:36.269844816 2 echo (7896) > munmap
34389 12:02:36.269850803 2 echo (7896) < munmap
34390 12:02:36.269851915 2 echo (7896) > close fd=2(/dev/pts/3)
34391 12:02:36.269852314 2 echo (7896) < close res=0
```

Figure 2: Raw Data Output



```
27 1520195399.487007044 0 gmain (779) > write cwd=/ fd=4(<e>) size=8  latency=0
28 1520195399.487011351 0 gmain (779) < write cwd=/ res=8 data=........  latency=4307
29 1520195399.487014172 0 gmain (779) > read cwd=/ fd=7(<i>) size=4096  latency=0
30 1520195399.487015605 0 gmain (779) < read cwd=/ res=-11(EAGAIN) data=  latency=1433
31 1520195399.487018800 0 gmain (779) > write cwd=/ fd=4(<e>) size=8  latency=0
32 1520195399.487019139 0 gmain (779) < write cwd=/ res=8 data=........  latency=339
33 1520195399.487020440 0 gmain (779) > write cwd=/ fd=4(<e>) size=8  latency=0
34 1520195399.487020745 0 gmain (779) < write cwd=/ res=8 data=........  latency=305
39 1520195399.487028707 0 gmain (779) > read cwd=/ fd=4(<e>) size=16  latency=0
40 1520195399.487029553 0 gmain (779) < read cwd=/ res=8 data=........  latency=846
```

Figure 3: Self-defined Data format

- proc.name is the name of the process that generated the event.
- proc.pid is the process id that generated the event
- evt.dir is the event direction, > for enter events < for exit events.
- evt.type is the name of system event e.g. 'open' or 'read'.
- proc.cwd is the current working directory of the event.
- evt.args is all the event agruments, aggregated into a single string.
- evt.latency is the delta between an exit event and the correspondent enter event in nanoseconds.

Currently, the information we try to gather includes events start time ,exit time, the id and name of the process that generated the event, the event arguments and duration. Based on the observation, the output data also includes many *switch* events, which is generated by the CPU task scheduling algorithm. This information is too detail and unnecessary for the dependency graph analysis. The goal of system monitoring is to record the legal user's or unknown attacker's operation. Therefore, this kind event can be safely neglected. In this project, there are three type system events need to be processed. The first type is about how a process is created. The relevant events are clone and fork in Linux system. The second type is file operations. The relevant event about this type are read, write and writev. The third type event is about network communication. The relevant system events are sendto. Through the filter provided by Sisdig, we can only gather the interested events.

3.4 Dependency Anlysis

Dependency analysis [10, 19, 11] plays an important role in security applications, such as finding the entry points of attacks (*causal analysis*) and investigating ramification of attacks (*impact analysis*). Given events evt_1 and evt_2, to construct a backward (forward) dependency $evt_1 \xrightarrow{backward}$

evt_2 ($evt_1 \xrightarrow{forward} evt_2$), evt_1 must occur after (before) evt_2.

Backward dependency analysis can be used to analyze the *causality* of an anomalous event, such as the creation of an unknown executable in a system. Backward dependency analysis finds the origins of the executables by searching the events backward in time: (1) start the investigation of the processes that create the executables; (2) from the processes, trace back to see which executables the processes executed; (3) if the executables are from a trusted source, then we can stop the search; otherwise, the tracking continues until it finds the origin (such as installer files or downloaders). As another example, forward dependency analysis helps users understand the ramification of malware. Assume that we identified a malicious script in the web server. To understand how it infects a victim machine, we can start the investigation of the process that executed or read from the script, and trace whether they propagate the malicious scripts across hosts In this project, the goal of this step is to reconstruct the time line in an attack. However, modern operation system is very complex, even through the user does not do any specific operation, it still has many service process running at the backend. That will generate many log records in the audit tool's output. In other words, these events will be represented as many edges and vertexes that we don't need. So we need to run backtrack[10, 11] from at least one *detection point*[12], such as a modified, extra, or deleted file, or a suspicious or missing process. There are two standards to decide which system entity or operation is relevant to *detection point* or not. The first one is whether there is a path existing between two vertexes or not. That means the source entity can use several operations to affect the state of the object entity. The second one is that the time intervals can be used to reduce false dependencies. For example, a process that reads a file at time 10 does not depend on writes to the file that that occur after time 10. The figure in the first column of Fiture 4 is the dependency reconstruction about how a python script read data from source file and then write data to another file. It is not hard to notice there are multiple edge existing between the same pair of vertexes. It gives us ground to further reduce the size of this dependency graph.

3.5 Causality Preserve Reduction

From the graph processed by the Backtrack, it is not hard to notice that there are always multiple edges between the same pair of vertexes. This is because the number of operations is not equal to the log record number. For example, If one user try to read a file, the system will not read all data from disk at one time, it will read it hundreds or thousands times. In other words, many edges are corresponding to the same event. In a real-world scenario, an average desktop computer produces more than 1 million events per day, while a server could yield 10 to 100 times the volume. Reducing the data volume is key to solving the scalability problem. Comparing the first and second column of Fiture 4, there is no more duplicate edges between the same pair of vertexes.

3.5.1 Design

In this section, we first present a formal definition of important terms and concepts used in our design. First, two events have *causality dependency* with each other if an event

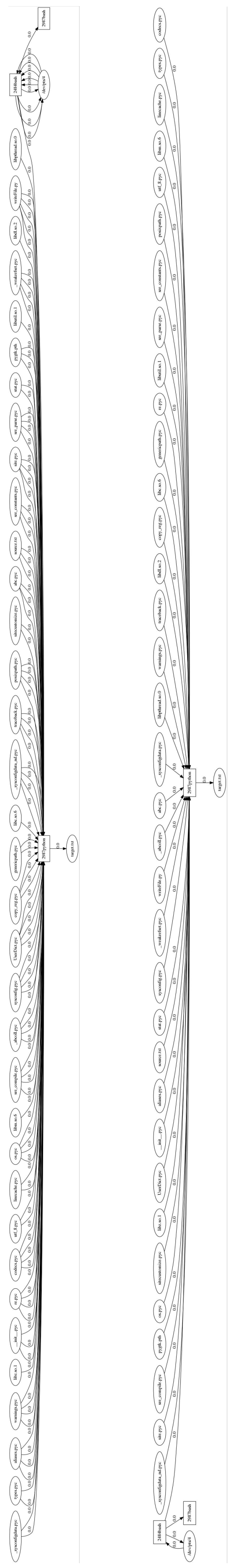


Figure 4: Backtracking And Causality Preserve Reduction Result

has information flow that affect the other event.
DEFINITION 1: **Causality Dependency**

For two event edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$, they have causality dependency, if

- $v_1 = u_2$
- $te(e_1) < te(e_2)$

if e_1 has information flow to e_2 and e_2 has information flow to e_3, then e_1 and e_3 have causality dependency. Our primary reduction scheme aims at preserving causality dependency while performing the data reduction. For exampleïijŇ assuming there are multiple edges between the vertex u and v, we need to check whether the merge of edges will break the backward trackability and forward trackability [23].We can merge the edges, if that merge will not loss the information about causality dependency.

4. APPROACH

In this part, we will present two parts of our work. The first one is about how to process the log file and get what kinds of information. The work flow of automatic dependency analysis is illustrated in Figure5.

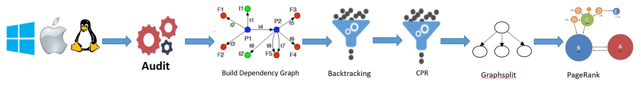


Figure 5: Overview of automatic dependency analysis

4.1 Enhanced Dependency Analysis (Milestone 1)

In this section, we describe the

Log Parsing. After gathering the data, we need build the dependency graph based on the log file. There are three type vertexes representing three kind system entities: process, file and network communication. The identifier for process is its name and id. For file, it is identified by its absolute file path, for network communication, it is distinguished by the communication two sides' ip and port number.The attributes for these three system entities are showed in Table 1. We use three kind shape to represent these three system entities. According to the entities included in the system event, the parser need to process five kinds edge : process to process, process to file, file to process, network to process, process to network. The type of edge is decided by the event arguments. The output of Sysdig will use *fd* as its identifier or key word. In the value part, it will use different characters to represent different objects.For example, it use *f* to represent file, *6* to represent IPV6 sockets,*4* to represent IPV4 sockets. This kind information is necessary, without this, it is hard to know the start or end of information flow is a file or an ip address. Through this way, the parser can know the kind of entities generating this event. For process to process event, we can directly know objects generating this event is two processes, because its event type is unique : fork or clone. In the conclusion, the parser will establish the dependency relationship between two system entities. This dependency relationship is specified by three parts: a source object, a sink object and a time interval$(t_s(e), t_e(e))$.Table2 shows the information that we collect in the log file.

Table 1: Representation Of System Entities

Entity	Attributes	Shape in Graph
File	Absolute Path	Ellipse
Process	PID and Name	Square
Network Connection	IP and Port	Parallelogram

Table 2: Representative attributes of system events

Categories	Attributes
Operation	read/write writeto fork clone
Time/Sequence	Start/End Time
Argument	IP Port data size

Limitations of Existing Work. In the existed research work, the information for the dependency analysis only includes file name, process, IP addresses and attributes of events including event origins, operations (file read/write), and other security-related properties. However, to infer application reputation, only these attributes are not enough. The reason is that they are mainly used to form a dependency graph with nodes being entities and edges representing data flows; but to infer reputations, each edge carries different weights in impacting the reputation propagation as shown in Figure 5. Moreover, dependency analysis includes filters based on the existing works [10, 11], which are not optimized for inferring reputations. Therefore, it is *necessary* to collect more attributes for the purpose of assigning weights to edges in the produced dependency graph and include more specialized filters to filter out unrelated events in the graph.

Relative Time to the POI event. Currently, we finished the building the dependency graph with additional necessary information. The first one is the relative time to the POI event(Detection Point).Once an application or a library is introduced to a system, it may get modifications via application updates, manually customizations, or malicious alternations. That is, a file node in the dependency graph can have multiple *write* edges (incoming edges) representing initial creation and subsequent updates. Intuitively, the write comes latter carries more weights in impacting the application reputation. Therefore, *the time when an event occurs* is one factor that impacts the reputation propagation: the latter it happens, the more weight it carries. As such, given a POI event, we propose to compute *the relative time to the start time of the POI event* for each of the subsequent edges included into the dependency graph, and use this new attribute as part of the computation for the weights of the edges.

Amount of Transfered Data. The second improvement compared with the traditional dependency analysis method is that we observe that the amount of transferred data is another important factor that impacts the reputation propagation.Not all the updates are rewriting application executables to malicious executables; some updates to an application are merely for property updates. For example, in Windows, the search indexer program frequently updates executables' metadata for indexing purposes, and each update writes only a few bytes to the executables. Therefore, we propose to extract *the amount of data transferred* for each read/write operation and annotates the corresponding edges in the dependency graph with this information. The

Algorithm 1: GraphSplit

```
   Input: A dependency Graph
 1 function: GraphSplit($(G)$)
 2 $Queue \leftarrow$ FindPair($G$);
 3 while $Queue$ is not Empty do
 4     $V \leftarrow Queue.poll()$;
 5     $S \leftarrow V.source$;
 6     $T \leftarrow V.target$;
 7     if Set contains $S$ or Set contains $T$ then
 8         continue
 9     end
10     $list1 \leftarrow$ list of outgoing edges of $S$ whose object is
         not $T$;
11     $list2 \leftarrow$ V.list;
12     UpdateGraph ($G$,$list1$,$list2$);
13     add $S$ to Set;
14     $Queue \leftarrow$ FindPair ($G$);
15 end
16 RecoverTimeLogic($Set$);
```

larger amount of data transferred, the more important the edge is. So for the weight calculation, we need to consider the relative time and amount of data together.

Algorithm 2: FindPair

```
   Input: A dependency Graph(G)
   Result: A queue containing the vertex pair need to
           be splited
 1 function: FindPair$(G)$
 2 $V \leftarrow G.vertexSet$;
 3 for $v \in V$ do
 4     $E \leftarrow$ incoming edges of $v$;
 5     if $\exists e \in E$ sharing the same source then
 6         $vertexPair.source \leftarrow e.source$;
 7         $vertextPair.object \leftarrow v$;
 8         $vertexPair.object.list$ add $e$;
 9         $Queue$ add $vertexPair$;
10     end
11 end
12 return $Queue$
```

4.2 Inference of Application Reputation via PageRank (Milestone 2)

PageRank [16] considers the World Wide Web as a set of linked nodes and ranks them based on their importance. The insight of PageRank is that a node linked by important nodes should be more important than the ones linked by uninfluenced nodes. That is, a web page's "reputation" is impacted by all the other web pages pointing to it. PageRank uses a transition matrix to represent the weights of edges from node j to node i. However,although Causality Preserve Reduction can merge many edges, there still may have multiple edges existing the same pair of nodes, because we don't want that the dependency relationship is lost during Causality Preserve Reduction. Therefore, we borrow the idea of static single assignment from(SSA) [14], splitting source node into multiple nodes, where each of the split node has only one outgoing edge pointing to target node. We need duplicate all the edges originally pointing to source to

Algorithm 3: UpdateGraph

```
   Input: A dependency Graph and.
   list1 is a list of outgoning edges of the source except
   edges whose object is V.
   list2 is a list containing edges between S and V
 1 function: UndateGraph$(G, list1, list2)$
 2 $list \leftarrow$ empty list of vertex;
 3 for $e \in list2$ do
 4     $u \leftarrow split(S)$;
 5     $list$ add $u$;
 6     $G$ add $u$;
 7     $G$ add a new edge from $u$ to $V$;
 8 end
 9 for $u \in list$ do
10     for $e \in list1$ do
11         $G$ add a new edge from $u$ to the object of $e$;
12     end
13 end
14 for $u \in list$ do
15     for $e \in$ in the incoming edges of $S$ do
16         $G$ add a new edge from the source of $e$ to $u$;
17     end
18 end
19 remove $S$, the incoming and outgoing edges of $S$ from
     the $G$;
```

each of the split node from node j.By replacing source node with the split nodes, we can then use PageRank to compute system entities' reputation.

Graph Split. Algorithm 1 shows the algorithm for GraphSplit. The basic idea for GraphSplit is that for every vertex(V) of the dependency graph, we need to check its incoming edges. If several incoming edges share the same source, we declare a $vertexPair$ contains object V, source S and a list of edges that connect these two vertexes. We maintain a queue for all $vertexPairs$. We also maintain a set of vertex has already been split. If we find the current $vertexPair.S$ or $vertexPair.V$ has been split($Line7$), then we just skip this pair. If both $vertexPair.S$ and $vertexPair.V$ are not be split, the *GraphSplit* will call *UpdateGraph* to split *vertexPair.S* at line 12. Then line 14 *FindPair* will update this queue. If the queue is empty, it means no more vertexes need to be split.

Find Pair. The basic idea of Algorithm *FindPair* is to count the edges between any connected vertexes. If any pair vertexes is connected by more than one edge, we will declare a *vertexPair* and push it into the queue.

Update Graph. This function generates vertexes that are as the same number as edges existing between the $vertexPair.S$ and $vertexPair.V$(from line 3 to 8) and then adds the copy of other outgoing and incoming edges of $vertexPair.S$ to these new vertexes(from line 9 to 18). After this, we can removes the $vertexPair.S$ and all the edges that connect with *ver-texPair.S* from the input dependency graph.

Recover Time Logic. This function is used to rebuild the time logic of the dependency graph. The Function *UpdateGraph* only deals with the graph geometric structure. According to the backward dependency definition [10, 19, 11], we need to make sure the exit time of any incoming edge of node generated by *UpdateGraph* is not bigger than its largest starting time of outgoing edges. If any incoming

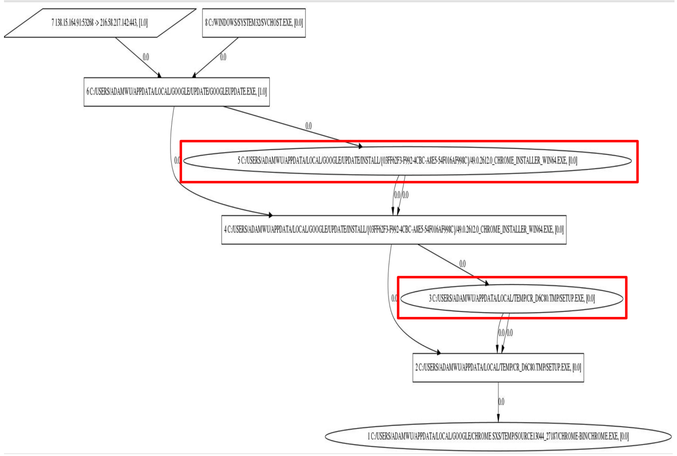

7 138.15.164.91:53268 -> 216.58.217.142:443, [1.0]
8 C:/WINDOWS/SYSTEM32/SVCHOST.EXE, [0.0]
6 C:/USERS/ADAMWU/APPDATA/LOCAL/GOOGLE/UPDATE/GOOGLEUPDATE.EXE, [1.0]
5 C:/USERS/ADAMWU/APPDATA/LOCAL/GOOGLE/UPDATE/INSTALL/{03FF62F3-F992-4CBC-A8E5-54F016AF998C}/49.0.2612.0_CHROME_INSTALLER_WIN64.EXE, [0.0]
4 C:/USERS/ADAMWU/APPDATA/LOCAL/GOOGLE/UPDATE/INSTALL/{03FF62F3-F992-4CBC-A8E5-54F016AF998C}/49.0.2612.0_CHROME_INSTALLER_WIN64.EXE, [0.0]
3 C:/USERS/ADAMWU/APPDATA/LOCAL/TEMP/CR_D6C80.TMP/SETUP.EXE, [0.0]
2 C:/USERS/ADAMWU/APPDATA/LOCAL/TEMP/CR_D6C80.TMP/SETUP.EXE, [0.0]
1 C:/USERS/ADAMWU/APPDATA/LOCAL/GOOGLE/CHROME SXS/TEMP/SOURCE13044_27187/CHROME-BIN/CHROME.EXE, [0.0]
0.0

Figure 6: Test sample

edge breaks this rule, we need to remove it. The line 4 in Algorithm 4 is to check whether this node is generated by *UpdateGraph*. The line from 6 to 8 is to rebuild backward dependency.

5. PRELIMINARY RESULT

In this part, will show our current project progress. The first table show our method's performance on the system monitoring data. The first column show the test case is that user use Linux command: *apt get* to install some applications. This operation is quite common in different environment. The second column is the log file size. We use the vertex and edge number to evaluate the size of the dependency graph. It is very obvious that our method reduce the number of vertex and edge in the dependency graph. It reduce the burden for further calculation. We also already test out GraphSplit on a small sample from the real-world scenario.Figure 6 and Figure 7 is the sample and result of GraphSplit. The node in the red square has multiple edges pointing to the same object, so this one need to be split. The node in the blue square is nodes generated by Graphsplit algorithm. In the next step, we will gather the aduiting data in a longer time and test its performance and overheat.

Algorithm 4: RecoverTimeLogic

```
   Input: A set of splited vertex
 1 function: RecoverTimeLogic($Set$)
 2 $V \leftarrow G.vertexSet()$;
 3 for $v \in V$ do
 4     if $v$ is the splitting node belongs to $Set$ then
 5         $endtime \leftarrow$ the biggest endtime of outgoing
             edges of $v$;
 6         for $e \in$ incoming edges of $v$ do
 7             if $e.start > endtime$ then
 8                 remove this edge;
 9             end
10         end
11     end
12 end
```

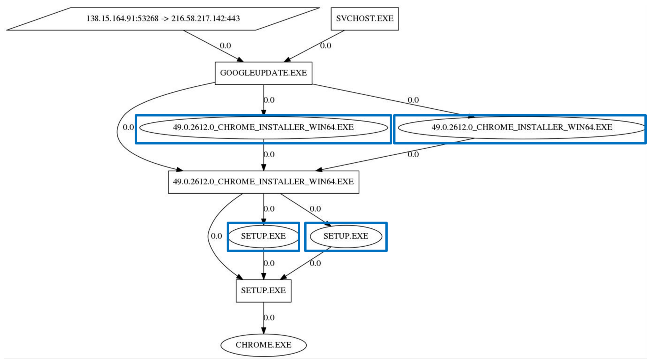

138.15.164.91:53268 -> 216.58.217.142:443
SVCHOST.EXE
GOOGLEUPDATE.EXE
49.0.2612.0_CHROME_INSTALLER_WIN64.EXE
49.0.2612.0_CHROME_INSTALLER_WIN64.EXE
49.0.2612.0_CHROME_INSTALLER_WIN64.EXE
SETUP.EXE
SETUP.EXE
SETUP.EXE
CHROME.EXE
0.0

Figure 7: GraphSplit Result

6. REFERENCES

[1] Cyber Kill Chain. http://www.lockheedmartin.com/us/what-we-do/information-technology/cybersecurity/tradecraft/cyber-kill-chain.html.
[2] Trustwave Global Security Report 2015. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf.
[3] Yahoo discloses hack of 1 billion accounts, 2016. https://techcrunch.com/2016/12/14/yahoo-discloses-hack-of-1-billion-accounts/.
[4] cnn. OPM government data breach impacted 21.5 million, 2015. http://www.cnn.com/2015/07/09/politics/office-of-personnel-management-data-breach-20-million.
[5] Ebay. Ebay Inc. to ask Ebay users to change passwords, 2014. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.
[6] Fireeye. Anatomy of advanced persistent threats. 2017.
[7] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *SOSP*, 2005.
[8] E. M. Hutchins, M. J. Cloppert, and R. M. Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1:80, 2011.
[9] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *ICDCS*, 2006.
[10] S. T. King and P. M. Chen. Backtracking intrusions. In *SOSP*, 2003.
[11] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.
[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
[13] Microsoft. ETW events in the common language runtime, 2017. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.
[14] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 2004.
[15] NPR. Home Depot Confirms Data Breach At U.S.,

Canadian Stores, 2014.
http://www.npr.org/2014/09/09/347007380/home-depot-confirms-data-breach-at-u-s-canadian-stores.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[17] Redhat. The linux audit framework, 2017. https://github.com/linux-audit/.

[18] C. T. U. research. Lifecycle of an Advanced Persistent Threat , 2012. http://www.redteamusa.com/PDF/Lifecycle

[19] S. Sitaraman and S. Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *IWIA*, 2005.

[20] Symantec. Advanced Persistent Threats: How They Work, 2017. https://www.symantec.com/theme.jsp?themeid=apt-infographic-1.

[21] J. B. Thummala. Defending advanced persistent threats - be better prepared to face the worst, 2016. https://www.infosecurity-magazine.com/opinions/defending-advanced-persistent/.

[22] N. Y. Times. Target data breach incident, 2014. http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html?_r=1.

[23] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang. High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 504–516. ACM, 2016.

Table 3: Statistical Result

sample	log file size	vertex(original)	edge(original)	vertex(backtracking)	edge(backtracking)	vertex(CPR)	edge(CPR)
apt-get instll unrar	17.5MB	5092	28502	2148	3911	2148	2346
apt-get instll postgresql	53.0MB	12174	82684	2667	11564	2667	3178
apt-get install zookeeper	19.7MB	4264	24368	2516	6982	2516	3020
apt-get install mongoDB	65.9MB	4205	45510	2712	11131	2712	2949
apt-get install wireshark	66.2MB	5838	64411	3511	34136	3511	4488