NodeMerge: Template Based Efficient Data Reduction For Big-Data Causality Analysis

Yutao Tang[2], Ding Li[1], Zhichun Li[1], Mu Zhang[3], Kangkook Jee[1], Xusheng Xiao[4], Zhenyu Wu[1], Junghwan Rhee[1], Fengyuan Xu[5], Qun Li[2]
[1]NEC Laboratories America Inc [2]The College of William and Mary [3]Cornell University [4]Case Western Reserve University [5]National Key Lab for Novel Software Technology, Nanjing University
[1]{dingli,zhichun,kjee,adamwu,rhee}@nec-labs.com [2]{yytang,liqun}@cs.wm.edu [3]mz496@cornell.edu [4]xusheng.xiao@case.edu
[5]fengyuan.xu@nju.edu.cn

ABSTRACT

Today's enterprises are exposed to sophisticated attacks, such as Advanced Persistent Threats (APT) attacks, which usually consist of stealthy multiple steps. To counter these attacks, enterprises often rely on causality analysis on the system activity data collected from a ubiquitous system monitoring to discover the initial penetration point, and from there identify previously unknown attack steps. However, one major challenge for causality analysis is that the ubiquitous system monitoring generates a colossal amount of data and hosting such a huge amount of data is prohibitively expensive. Thus, there is a strong demand for techniques that *reduce the storage of data* for causality analysis and yet *preserve the quality of the causality analysis*.

To address this problem, in this paper, we propose NodeMerge, a template based data reduction system for online system event storage. Specifically, our approach can directly work on the stream of system dependency data and achieve data reduction on the read-only file events based on their access patterns. It can either reduce the storage cost or improve the performance of causality analysis under the same budget. Only with a reasonable amount of resource for online data reduction, it nearly completely preserves the accuracy for causality analysis. The reduced form of data can be used directly with little overhead.

To evaluate our approach, we conducted a set of comprehensive evaluations, which show that for different categories of workloads, our system can reduce the storage capacity of raw system dependency data by as high as 75.7 times, and the storage capacity of the state-of-the-art approach by as high as 32.6 times. Furthermore, the results also demonstrate that our approach keeps all the causality analysis information and has a reasonably small overhead in memory and hard disk.

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**;

KEYWORDS

Security, Data Reduction

ACM Reference Format:
Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, Qun Li. 2018. Node-Merge: Template Based Efficient Data Reduction For Big-Data Causality Analysis. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243763

1 INTRODUCTION

Modern enterprises are facing the challenge of Advanced Persistent Threat (APT) attacks. These sophisticated and specially designed threats are conducted in multiple steps, and can remain undetected for weeks, months and sometimes years, while stealthily and slowly gathering critical information from victims [2, 10, 12, 42]. For instance, the US retailer TARGET [42] leaked 400 million users' credit card information; one third of US citizens' SSNs has been exposed due to the Equifax data breach [12].

Unfortunately, it is extremely hard, if not impossible, to always capture intrusions at their early stages because they may not release signals that are strong enough to trigger alarms. Consequently, to be able to diagnose and safely recover from attacks, once a suspicious activity has been detected, causality analysis [3, 21, 23, 24, 30, 38] is very much desired to discover its initial penetration points as well as other attack footprints previously unknown.

To enable causality analysis in enterprises, it requires a comprehensive collection of operating system events from all corporate computer hosts. These events record the interactions between OS-level resources, such as file read/write, process start/end and network communication. Despite the effectiveness of causality analysis, such a ubiquitous data logging can generate a colossal amount of records, which introduces significant pressure to storage systems. For instance, 70PB of data will be produced from a typical commercial bank with 200,000 hosts every year [46]. Hosting this enormous data may require millions of US dollars and can impose a heavy burden on enterprise security budget [11].

In addition, when massive data have been accumulated, they may cost excessive time to search for necessary events and thus to reconstruct causalities. However, prior work [28] has shown that, although causality analysis is conducted in a post-mortem manner, it is in fact time critical. In order to quickly locate interesting causal dependencies, it is thus preferable to store historical events in well-indexed online databases than to save the data into backup storage (e.g., tape drives) in a compressed format such as 7zip. To further

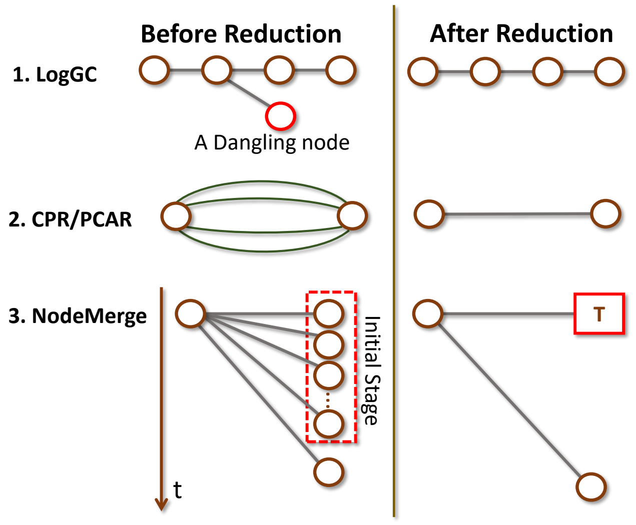

Figure 1: Compare our approach NodeMerge to existing reduction techniques such as LogGC [25] and Xu's work of CPR/PCAR [46].

increase query speed, it is also possible to employ SSD instead of HDD, since the superior random-access performance of SSD can greatly facilitate index-based search during causality graph construction. Nonetheless, the acceleration of causality analysis comes at a price, as a SSD is 5 times as expensive as a HDD of the same size [8] while a HDD is 4 times more costful than a tape drive [1]. Hence, reducing data storage of OS events may potentially lead to a more timely causality analysis because the same budget can then be spent on a smaller volume of faster storage device.

As a result, data reduction is of crucial importance for both reducing the cost and improving the speed of attack investigation. This, however, is a challenging task for two major reasons. First, an effective data reduction mechanism has to preserve the interdependencies among OS events, so as to support causality analysis. Hence, although simple, statistical summary of events can lead to major data reduction, it does not serve our needs because it may remove critical attack activities and thus break causalities. Second, the volume of OS events must be reduced before they reach storage systems. Otherwise, reducing data that has already been stored to databases causes excessive extra I/O overhead. Consequently, a fast online algorithm is highly desired to achieve our goal.

A few approaches have been proposed in this line of research in the security community. LogGC [25] removes temporary files from the collected data since these files have little impact on causality analysis. Xu et al.'s work [46] merges repeated low-level events between two OS objects, such as multiple read events between a process and a file, in order to extract a high-level abstraction of system activities. In contrast, our work, as shown in Figure 1, exploits a totally different data access pattern for data reduction, and thus is complementary to the previous work. In fact, in our evaluation, we demonstrate that our approach can achieve additional data reduction on top of Xu et al.'s technique.

Our data reduction is inspired by the key insight that frequent invocation of programs can produce a huge volume of redundant events because every process initialization performs constant and intensive actions to load libraries, access read-only resources and retrieve configurations. For instance, if a program has been executed for 1,000 times while each time 100 files have been accessed during initialization, eventually $1,000 \times 100 = 100,000$ events can be easily accumulated for merely this individual program. To our study, this execution pattern is fairly common in many different workloads, particularly in the environments of data analytics, big-data processing, and software development, testing and simulation. We also have an anecdotal observation that those machines bearing such workloads may create hundreds of times more events than others.

From the perspective of causality analysis, these libraries, resources and configuration files can be considered as a group without breaking original data dependencies. Based upon this observation, we propose a novel online, template-based data reduction approach, NodeMerge, that automatically learns the fixed sets of libraries and read-only resources for running programs as templates, and further uses these templates to compress system event data.

The key technical challenge of NodeMerge is how to discover the templates effectively and efficiently with a constrained system overhead. To address this challenge, we adopted and optimized the FP-Growth algorithm [16], which can discover frequent item patterns in system events. Our enhanced algorithm takes full advantage of the characteristics of system event data, so that it is orders of magnitude faster and thus more efficient than the original FP-Growth algorithm.

Our approach is designed as an online service that reads a stream of system events from the ETW [32] and Linux Auditing [37], learns the templates based on a subset of historic data, and reduces the future data after the learned period. NodeMerge has many novel and unique advantages. First, the data reduced by NodeMerge can be directly used in causality analysis without a costly inference process. Using the reduced data by NodeMerge introduces negligible runtime overhead to causality analysis. Second, NodeMerge uses constrained memory space to hold the intermediate data. Moreover, NodeMerge introduces nearly zero information loss in causality analysis. It guarantees that the dependencies between system events are preserved before they have been retired. Finally, being orthogonal to the existing data reduction approaches such as LogGC, NodeMerge can be integrated with other approaches for further data reduction.

To evaluate NodeMerge, we extensively evaluated 1.3 TB data collected from a real-world corporate environment with 75 hosts in an anonymous company, with multiple types of workloads. For certain workloads, we can reduce the data by 75.7 times. Even when the state-of-the-art data reduction approach [46] has already been applied, our technique can still further improve the storage capacity by 11 times on the host level and 32.6 times on the application level. Such a high data reduction ratio may potentially help an enterprise to save more than **half a million US dollars** for storage cost each year [36, 46]. To confirm NodeMerge's capability of keeping system dependency information, we performed causality analysis on the event data before and after reduction. Our test cases included five real-world attacks, five real-world applications, and 11,587 randomly selected events. Encouragingly, NodeMerge produced zero false positives in system dependency analyses. Lastly, we measured the computational cost of NodeMerge. Our approach on average uses 1.3 GB memory during the reduction for processing 1,337 GB data from 75 machines of 27 days. It took 60 minutes in each training cycle to learn the templates. This result confirms that NodeMerge could reduce the data with a reasonable amount of cost. Overall, our evaluations confirm that NodeMerge is effective and efficient in reducing the storage requirement of system event data.

We summarize the contributions of this paper as follows:

- We propose a novel system, NodeMerge, to improve the storage capability of security event storage systems by as high as 75.7 times.
- NodeMerge contains a novel FP-Growth based algorithm, which is substantially faster than the original FP-Growth algorithm, to discover the file access patterns of processes.
- We theoretically prove that, compared to the original FP-Growth algorithm, the loss of reduction capability of our algorithm is bounded.
- We performed an extensive evaluation with real-world enterprise data.

2 BACKGROUND AND MOTIVATION

In this section, we briefly discuss the background of causality analysis and its related data, and our observation of template based data compression that has inspired this work.

2.1 System Event Data and Causality Analysis

System event data record the causality relationships between system objects. In this paper, we discuss three types of events: process events, file events, and network events, as commonly used in security analysis. Process events record the actions of processes, such as a process start, fork, and stop. File events record how files are used, such as a file read, write, open, close, delete, and rename. Network events record the network related actions, such as a network open, read, write, and close. All these three types of events also record the source system object and the target system object of the event, the time stamps, file permissions, and other necessary system information for a dependency analysis. System event data can be collected by the system monitoring frameworks, such as Event Tracing for Windows (ETW) [32] and Linux Audit [37].

Causality analysis organizes system events as a dependency graph. It allows security experts to discover causal relationships between system events. By doing so, security experts may potentially reconstruct the attack scenarios for Advanced Persistent Threat (APT) attacks [15, 21]. The nodes of the dependency graph are system objects and the edges are system events, such as a file read and a process start. The directions of edges represent the directions of data flow or control flow between system objects. Figure 2 is an example of a dependency graph. This example illustrates how a bash script starts a python script to write logs to the disk. In Figure 2, the circles with solid boundaries are files and the ones with dashed boundaries are processes.

In a practical system dependency monitoring system, the amount of system event data can be colossal. Based on our experience of the system event data in an anonymous company, the volume of system event data can be as high as 2GB per one day per host. A typical

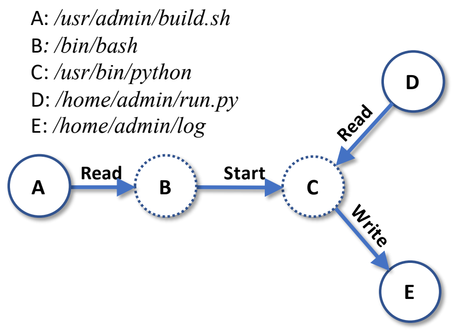

Figure 2: An example of dependency graph.

commercial bank can have more than 200,000 hosts [46], which implies that such places may need about 140PB storage to host the data for a year! This amount of data reflects the urgency of effective data compression algorithms. However, since causality analysis is frequently used by security experts, traditional data compression techniques, such as 7-zip [35], is not an efficient solution. Using the data compressed by techniques like 7-zip requires an explicit, non-trivial decompression process. Whenever people need to use the data, they need to decompress the data. It is unacceptable. Instead, a decompression-free data reduction technique is required.

2.2 Template Based Data Reduction

By studying the system event data collected in an anonymous company, we observed a common behavior that processes often repetitively access the same set of files when they start. This pattern is not particular on our environment rather a program characteristic. This is because when a process starts, it often needs to load several libraries and resources for initialization. We have several observations on this behavior for data compression: (1) these libraries and resources are the same across different process instances of the same executable; (2) libraries and resources do not contain useful system dependency information since they are read-only. Thus, the file events that are associated with the accesses of these libraries and resources can be merged into one special event without losing the accuracy in system dependency analysis. For example, Figure 3 shows a dependency graph of running a Perl script $/home/admin/run.pl$. The circles with dashed lines represent the process. Before the Perl interpreter reads the script, it will first load eight libraries, such as $/etc/ld.so.cache$, $/lib/x86\_64-linux-gnu/libpthread.so.0$, $/usr/lib/libperl.so.5.18$ and so on. Loading these libraries is necessary to execute any Perl scripts. Thus, for every Perl process, we can observe that the same eight files are loaded. Furthermore, these eight files are all read-only, which means that they are always the sources of any paths in the system dependency graph. Hence, in the dependency graph, these eight files can be merged as one single node without losing the dependency information outside these eight files.

Based on the observation above, our conclusion is that: *representing a set of read-only files that always be accessed together in the system event data as a single special file in a system dependency graph does not change the result of causality analysis.*

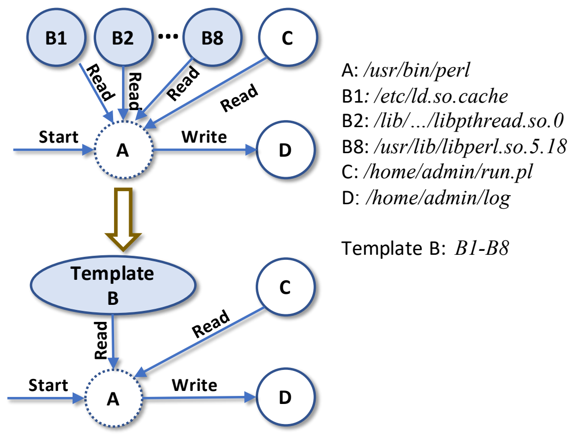

Figure 3: Dependency graph of running a Perl script

Based on this conclusion, we propose a new approach of template based data reduction. Specifically, for each executable, our method first discovers a set of templates of read-only files that are always accessed together. After that, it merges the read-only files in the templates as one item without breaking the system dependencies.

In practice, it is very challenging to do the template based data reduction due to following reasons. The first challenge comes from the randomness of system behaviors. Although an executable will access a fixed set of files when it starts, the order of these files can be varied or may contain noises. For example, python accesses $/etc/ld.so.cache$, $/lib/x86\_64-linux-gnu/libpthread.so.0$, and $/usr/lib/libperl.so.5.18$ when it starts, the order of them are often different in different instances. Furthermore, besides accessing the three files, python may also access some random files in different instances. Under such a noisy condition, it is challenging to discover templates for data reduction.

The second challenge comes from the large volume of data. Storing all the system event data in the database and performing the reduction offline require too much space overhead to host the intermediate data. To minimize the space requirement for intermediate data, we need an online compression approach that can directly compress an incoming stream of system event data before storing them in the database.

3 THREAT MODEL

We consider the same threat model as related works,e.g., [25, 45, 46]: while the adversaries can compromise user-level programs, the system audit data is still provided and protected by the kernel. We assume that adversaries have full knowledge about the NodeMerge reduction algorithm and can gain control of a certain portion of hosts from the enterprise. Adversaries can run malwares, plant backdoors, or run other reconnaissance tools (e.g, nmap [29]). Events generated by those programs are faithfully recorded and reported. We also assume that the backend remained in the trusted domain and cannot be compromised by an adversary. Although keeping the integrity of event data during system auditing and safely storing the data are important, those problems are outside the scope of this work. We can employ existing techniques [17, 18, 20] to address these problems.

4 DESIGN DECISIONS

In our approach, we made four major design decisions. First, we choose to design NodeMerge as an online reduction system. An offline system has less requirement on the reduction speed which allows a more complex algorithm to achieve higher reduction ratio. However, an offline system also needs to cache the original data, which could be very large and expensive to store. In our design, we choose to limit the storage cost to cache the original data. Thus, we make our system an online system that directly reduces the data from the stream.

Second, we choose to have a decompression-free data schema to store the data. That is, there is not an explicit decompression process when the NodeMerge data is used in causality analysis. Our data can be decompressed on-the-fly without slowing down the causality analysis. We make this choice because the system event data are frequently accessed by causality analysis in the daily routine of security experts in an enterprise. An expensive decompression may significantly affect the speed of causality analysis. A decompression-free schema can avoid such an impact.

Third, we choose to merge read-only files as our main reduction method. We make this decision to keep the system dependencies in the causality analysis. Read-only files are "dead ends" in causality analysis. Merging multiple read-only files does not change the dependencies between other events. By having a novel read-only file detection method, our system guarantees that the dependencies between system events are maintained before they have been retired.

Fourth, we deploy NodeMerge on a centralized server. In general, there are two options for the deployment of our system: (1) distributed deployment on each host in an enterprise; (2) deployment on a centralized server. A distributed deployment may have less network cost but will also increase the workload of the hosts. It may decrease the user experience of the hosts. Further, deploying NodeMerge on a centralized server may achieve more data reduction by learning the data reduction templates with the global information across the whole network. According to [46], collecting the events of one host to the centralized server may cost 100Kbit/s network bandwidth. Such a cost is trivial for the internal network of an enterprise. Thus, in our design, we choose to keep the user experience of each host and achieve higher data reduction by spending slightly more cost of the network.

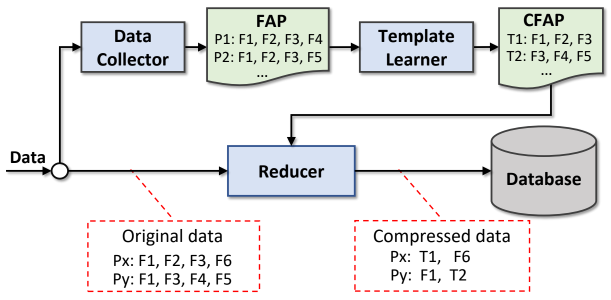

Figure 4: The architecture of NodeMerge

Process Name	Files	Ranked
P1	F1, F2, F3, F4, F5, F7	F2, F4, F7, F3, F5, F1
P2	F2, F4, F7	F2, F4, F7
P3	F6, F7	F7, F6
P4	F2, F3, F4	F2, F3, F4
P5	F2, F3, F4	F2, F3, F4

Table 1: Collected file access data for each process

5 ARCHITECTURE

Figure 4 presents the architecture of NodeMerge. In our design, NodeMerge runs on the cloud in an enterprise. The input of NodeMerge is the stream of system events gathered by ETW or Linux Audit on each host. In the enterprise, each host reports its event from ETW or Linux Audit to a centralized stream. NodeMerge runs on a server which monitors the centralized stream, reduces the data, and then store them in the database.

NodeMerge comprises three components collaboratively realizing the four design decisions. The first component is the Data Collector, which organizes and caches the most recent streamed data into a cache – File Access Pattern (FAP), on the local disk. Such a cache design allows the online reduction method. The Data Collector also checks the items in FAP and only keeps the read-only files. The size of the FAP is configurable. The second component is the Template Learner, which periodically checks the FAP and generates the Compressible File Access Pattern (CFAP) for data reduction. The third component is the Reducer, which uses the CFAP to reduce the incoming streamed data. The generated data is already in the decompression-free schema. It then stores the decompression-free schema into the database.

Our design of NodeMerge has two advantages. First, its resource usage is limited. The disk cost for hosting intermediate results is the size of FAP, which only hosts a subset of the total data and has a fixed size. The memory usage of NodeMerge is mainly for hosting CFAPs, which is also limited as we will show in our evaluation. Second, NodeMerge is robust to bursts of system log data events. In our design, the data cached in the FAP is only used for CFAP learning. If there is a burst of system log data events that causes the FAP to drop events, it will only lose a few CFAP instead of losing the events in the database.

6 DATA COLLECTOR

Data Collector has two main functionalities. First, it caches and organizes the file access information as a FAP, which will be then used to learn the file access templates in the online reduction method. Second, it checks the items in the FAP to only retain the read-only files. This allows the reduced data to keep the event dependencies in the causality analysis.

6.1 Generating FAP

In our approach, a FAP is defined as follows:

- A FAP is a table on the local disk with two columns.
- The first column, Process ID, contains the IDs of processes
- The second column, Files, contains the lists of files that are accessed by the process of the same row in the initial stage. In our approach, we define the first k seconds after a process starts as the initial stage of the process. We will show how we choose the value of k in Section 9.

An example of FAP is shown in the columns of Process Name and Files of Table 1. In this example, the Files in the row of P1 contains all the files that are accessed by P1 in its initial stage. Note that in our example, we only show the file names in the FAP for the conciseness of the illustration. However, in our design, other attributes of the files, such as permissions and owners, are also stored.

The process of building the FAP is as follows. The Data Collector monitors the streamed data. If the Data Collector finds an event that represents a process start, it creates a new row with the corresponding process ID and inserts it to the FAP. At the same time, the Data Collector also records the start timestamp of the process. If the FAP is full, the Data Collector will remove the previous rows in the FAP with a First-In-First-Out (FIFO) scheme. When the Data Collector finds a file event that is in the initial stage of a process, the Data Collector obtains the corresponding row of the FAP and inserts the file event to the Files column of the row.

6.2 Identifying Read-only Files

Data Collector needs to only keep the read-only files in the FAP so that the reduction does not break the event dependencies in the causality analysis. Thus, it is important for the Data Collector to detect read-only files.

The Data Collector classifies a file as a read-only file if there isn't a file write event to the file since a predefined time point t_g. Any files that are older than t_g is out of the concerns of the security experts. To detect the read-only files, the Data Collector monitors the file write events in the streamed data. For each file, the Data Collector keeps its last write event time, which is the timestamp of the Data Collector seeing the latest file write event to the file on the stream, to a local database. When the Data Collector sees a file write event, it will update the last write event time of the corresponding file accordingly. While checking the items in the FAP, the Data Collector compares the latest file write event time of each item to the predefined timestamp t_g. If the latest file write event time is older than t_g, the Data Collector classifies the file as a read-only file. In our design, t_g is the earliest time point that the system log data is available or concerned. In practice, system log data will be retired after a certain period of time, e.g. three months. NodeMerge does not keep the dependencies in the retired data since they will not be available anyway.

One challenge for our read-only file reduction method is the initialization of the local database. There are two types of initialization that NodeMerge needs to handle. First, when NodeMerge is newly installed, it needs to give each file an initial last file write event time. However, NodeMerge does not know the actual last file write event at the installation time since it hasn't started monitoring the event stream. To initialize the last file write event time for each file in the enterprise environment, NodeMerge scans the file objects in the enterprise environment and uses the last modification time in the metadata of these files as their initial last file write event time. By doing this, we assume that the system is not compromised before the installation of NodeMerge. Second, when a trusted application is newly installed, NodeMerge needs to initialize the last write event time of the installed files. Otherwise, this application will not be compressed since its last write event time is newer than t_g.

For this case, we allow the security manager of an enterprise to manually set a bogus last write event time, which is older than t_g, for a newly installed trusted application. By doing so, NodeMerge will ignore the creation of the application and detect a file of the application as read-only if there are not future modifications to it. For applications installed by normal users, their last write event time will be their creation time. The files of these applications will not be considered as read-only files until the security manager updates t_g to a time that is newer than the last write event time of the files.

7 TEMPLATE LEARNER

In a high level, Template Learner detects the frequent patterns about how files are accessed by different processes and summarizes the frequently used file combinations as the templates. Specifically, Template Learner periodically checks the FAP and generates the templates for each application, which are stored as CFAPs.

A key challenge is that it needs to be efficient enough for an online reduction system. To learn the templates efficiently, we propose an FP-Growth [16] based approach. It has three steps. First, the Template Learner builds an FP-Tree, which is a more concise representation of the FAP. Second, the Template Learner generates the CFAPs for data reduction. After the Template Learner has learned the CFAPs, it will also clean the current FAP and release the space. Third, the Template Learner converts all the CFAPs as a Finite State Automaton (FSA) so that the Reducer can use it efficiently during data compression.

7.1 Building FP-Tree

As the first step of our approach, Template Learner takes the FAP as an input and generates a FP-Tree. Before building a FP-Tree, the Template Learner makes a ranking based on the usage frequency of files for the FAP. The usage frequency of a file is defined as how many times does the file appear in the FAP. The files in each row of the FAP are ranked in a descending order based on their usage frequency. For example, the Ranked column of Table 1 is the result of the ranking. In Table 1, F2 is used four times (by P1, P2, P4, and P5) while F1 is used only once (by P1). Thus, in the first row of the Ranked column, F1 is ranked after F2. An FP-Tree is defined as follows [16]:

- It has a root node.
- Each node except the root node represents an item in the *Files* column of the FAP. A node contains three fields, the file id, a counter, and a link to the node that has the same file id.
- For two nodes $A \longrightarrow B$, B is the child of A if there exists a row in the FAP that contains both A and B in the *Files* column, and A is ranked exactly one position before B after the usage frequency based ranking.

The algorithm of building FP-Tree is as following [16]. First, this approach creates the root node of the FP-Tree. Then, it scans each row of the FAP, which is already sorted, to build other nodes of the FP-Tree. For each row, R, in the FAP, the Template Learner calls the $INSERT(R, Root)$ in Algorithm 1. The method $INSERT$ accepts two parameters, the first one, F, is the list of files in a row of the FAP. The second parameter, N, is a node of the FP-Tree. The method of $INSERT$ recursively inserts files in F into the tree after the node, N. It first checks if the first file of F has the same ID as the node N (line 2). If so (line 3 and 4), $INSERT$ increases the counter of N and inserts the following files of F to the tree recursively. Otherwise (line 6 and 7), it creates a new node of the FP-Tree for the first file of F and appends it as a child of N. Then, $INSERT$ recursively inserts the other files of F after the new node.

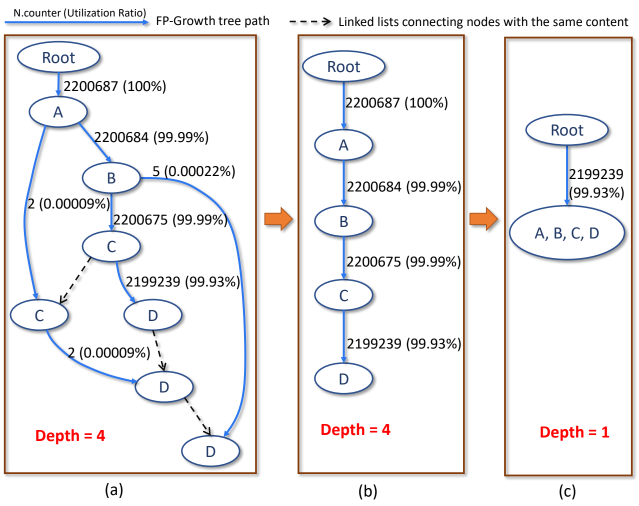

Figure 5: An example of FP-Growth tree optimization. (a) Original FP-Growth Tree. (b) The FP-Growth tree after infrequently accessed paths are removed. (c) The FP-Growth tree after merging WDPs

Algorithm 1 FP-Tree Insertion

```
Input: F: a list of files, N: a node in the FP-Tree
1: procedure INSERT($F$, $N$)
2:     if $F[0].file\_id == N.file\_id$ then
3:         $N.counter++$
4:         $INSERT(F[1:], N)$
5:     else
6:         Add $F[0]$ as a child of $N$
7:         $INSERT(F[1:], F[0])$
8:     end if
9: end procedure
```

7.2 Discovery of Compressible File Access Pattern

In this step, the Template Learner analyzes the FP-Tree and generates the CFAPs, which will then be used to generate the templates for reduction. In our approach, we define a CFAP as a set of files that appears together more than k times in the FAP. For example, in Table 1, if we set k as 2, $\{F2, F4, F7\}$ and $\{F2, F3, F4\}$ are two distinct CFAPs.

The CFAPs can be generated by applying the standard FP-Growth algorithm [16] over the FP-Tree. However, this approach has two limitations. Firstly, the standard approach usually generates an FP-Growth tree with a depth larger than 100 levels. As the algorithm's complexity is exponential to the depth of the tree, such a deep tree would hinder the original FP-Growth algorithm from finishing in a reasonable amount of time. Secondly, the standard approach is deemed to generate conflicting CFAPs (whose intersection is not empty) which would confuse Reducer. To address these two limitations, we altered the standard FP-Growth algorithm by prioritizing CFAPs selection and making optimizations to the FP-Tree.

7.2.1 Prioritized Compressible File Access Pattern Selection: The purpose of prioritized CFAP selection is to address the conflicting CFAP items. The main idea is when there are conflicting CFAPs to choose the one with the largest data reduction capability. It has two steps. First, our approach runs the standard FP-Growth algorithm to discover all combinations of each path in the FP-Tree. Finally, our approach selects the CFAP segment candidates that can approximately maximize the data reduction ratio.

To do the prioritized CFAP selection, for each item in the CFAP generated by the original FP-Growth algorithm, our approach gives it a data reduction score, which is defined by Equation (1). On the high level, Equation (1) measures the potential data reduction of a CFAP segment candidate t. Before our reduction process, the unmerged data that matches t takes $t.freq * t.size$ units of space. After the reduction, it takes $t.size$ units of space to save the template and $t.freq * 1 = t.freq$ units of space to save the merged data. Thus, the total benefit of our data reduction will be the same as Equation (1).

$$score(t) = t.freq * t.size - t.size - t.freq \tag{1}$$

With the definition of the reduction scores, our approach uses a greedy algorithm to choose the CFAP candidates. We choose the greedy algorithm since it is computationally inexpensive and can provide a good enough data reduction ratio in practice. Our algorithm iterates over the CFAPs generated by the FG-Growth algorithm and selects the one with the highest data reduction score that does not have overlap with other selected candidates. This process is repeated until no candidates can be selected.

7.2.2 FP-Tree Optimization. To address the first limitation of the original FP-growth, which is not efficient enough, we propose two FP-Tree optimization methods to accelerate the learning process. First, our algorithm removes infrequently accessed paths. Second, our approach merges the single paths in the FP-Tree. Note that these optimizations do affect the accuracy of the reduced data. They only slightly lose certain reduction ratios. Figure 5 is a real study case that explains how our optimizations work.

Removing Infrequently Accessed Paths: Templates learned from infrequently accessed paths do not have much reduction capability. Thus removing them can speed up the process of template discovery while maintaining the reduction rate.

To trim barely utilized paths in FP-Tree, our approach trims a node and all its children from the FP-Tree if it has the utilization ratio less than the threshold. The utilization ratio for a node N of an FP-Tree is defined as:

$$\frac{N.counter}{Root.counter} \tag{2}$$

On the high level, for a node, N, in the FP-Tree, Equation (2) checks the ratio of its counter to the total number of patterns in the FP-Tree, which is the counter of *Root*. According to Algorithm 1, the counter of N represents the frequency of N being access together with its parent by the same process. If the utilization ratio of N is low, it means that N is rarely accessed together with its parent. Thus, the CFAP candidates that contain N and its parent will not be very effective in data reduction.

Weakly Dominated Path Merging: Merging Weakly Dominated Paths (WDPs) can reduce the length of the FP-Tree so that it can accelerate the speed of FP-Growth. This complexity of FP-Growth is exponential to the depth of the FG-Tree. Reducing the depth of the FP-Tree can significantly speed up the algorithm.

On the high level, a WDP is a single path of the FP-Tree that the counters of all the nodes on the path have roughly the same values. To define WDPs, we first define the weakly domination relationship. For following part of this section, we define s and e as two nodes of an FP-Tree, FP. We also define $P(s, e)$ as a path in FP that starts with s and ends with e. We say s weakly dominates e in FP if and only if $P(s, e)$ is a single path and Equation (3) holds.

$$\forall p1 \forall p2 \in P(s, e) \rightarrow (1 - \sigma) < \frac{p1.counter}{p2.counter} < (1 + \sigma) \tag{3}$$

Where σ is a redefined error threshold.

With the definition of weakly domination, we define the WDPs. We note $WDP(s, e)$ as the WDP that starts from s and ends at e. Then $WDP(s, e)$ meets following two conditions: first, s weakly dominates e; second, the path from s to e is the longest path that holds the weakly domination relationship.

Our method of merging WDPs guarantees that the CFAPs selected from the FP-Tree after WDP merging are close enough to the CFAPs selected without WDP merging. Strictly speaking, assume P is a WDP in the FP-Tree, T_{pre} is the set of CFAPs selected on P by the CFAP choosing algorithm in Section 7.2.1 before merging the WDP and T_{post} is the set of templates selected on P after merging the WDP, we have:

$$score(T_{post}) > (1 - \sigma - \varepsilon) score(T_{pre}) \tag{4}$$

where $score(T) = \sum_{t \in T} score(t)$, σ is as the same as in Equation (3), and $\varepsilon \rightarrow 0$ when the max value of frequency of the CFAPs is very high. To prove Equation (4), we have following two lemmas:

LEMMA 7.1. *$score(T_{post}) = score(t_{all})$, where t_{all} is the template candidate that contains all the nodes in P.*

PROOF. Since all nodes in P are merged as one node, the selective CFAP discovering has to select them all after WDP merging. □

LEMMA 7.2. *$\frac{score(t_{all})}{score(T_{pre})} > 1 - \sigma - \varepsilon$. When $f_{max} \rightarrow +\infty$, $\varepsilon \rightarrow 0$, where f_{max} is the largest frequency of all the CFAP candidates in T_{pre}.*

PROOF. For $score(T_{pre})$, since in Section 7.2.1, our algorithm selects CFAP segment candidates that have no overlap on a single path, we have

$$\sum_{t' \in T_{pre}} score(t') < f_{max} T_{pre}.size - T_{pre}.size - f_{max}.$$

According to Equation (3), we have $\frac{f_{min}}{f_{max}} > 1 - \sigma$, where f_{min} is the minimum frequency of all CFAP candidates in T_{pre}. Then, we have:

$$\begin{aligned}\frac{score(t_{max})}{score(T_{pre})} &> \frac{f_{min}T_{pre}.size - f_{min} - T_{pre}.size}{f_{max}T_{pre}.size - f_{max} - T_{pre}.size}\\ &> \frac{(1-\sigma)f_{max}T_{pre}.size - (1-\sigma)f_{max} - T_{pre}.size}{f_{max}T_{pre}.size - f_{max} - T_{pre}.size}\\ &= 1 - \sigma - \sigma\frac{T_{pre}.size}{f_{max}T_{pre}.size - f_{max} - T_{pre}.size}\end{aligned} \tag{5}$$

When $f_{max} \to +\infty$, $\sigma\frac{T_{pre}.size}{f_{max}T_{pre}.size - f_{max} - T_{pre}.size} \to 0$ □

Based on Lemma 7.1 and Lemma 7.2, it is clear that Equation (4) holds.

8 REDUCER

The function of the Reducer is to merge incoming file events based on the CFAPs learned by the Template Learner. The CFAPs are organized as an FSA. Then, the Reducer matches the file events in the initial stage of a process to the FSA for data reduction.

To reduce the streamed data, the Reducer maintains a small FAP in the memory as a buffer. While monitoring the streamed data, the Reducer stores the processes and the files accessed by them in the initial stage. At the same time, the Reducer monitors the age of each process, which is the time elapsed since the start time of the process. When the Reducer sees a process has an age older than a threshold, it sorts the files accessed by the process in its initial stage on the file IDs and matches the sorted file sequence to the FSA. The threshold is a long enough time span that can cover the whole events of a process in its initial stage. It should be longer than the length of the defined initial stage since the events may have delays. If the sequence is accepted by the FSA, it will be merged as one reduced event.

8.1 Building Finite State Automaton

To build the FSA, the Reducer sorts the items in each CFAP on the file ID of each item. Then, the Reducer treats all the sorted CFAPs as strings of file IDs and merges them as an FSA. Finally, in the FSA, when each state finds a mismatch in the incoming file, instead of transiting to the failed state, it will stay in the current state. Thus, the FSA won't fail when there are a few noises in the stream.

The formal process of building the FSA is shown in Algorithm 2. Before processing the CFAPs, Algorithm 2 builds an empty FSA, F, and adds an initial state, $Init$, to it. At line 4, Algorithm 2 ranks the files of each CFAP in a list r. At line 5 to line 9, Algorithm 2 iterates over the ranked list r and creates a state for each of the file in r. For the file $r[i]$, its associated state in F is $S_{r[i]}$. For each $S_{r[i]}$, it jumps to the state of the next file, $S_{r[i+1]}$, if it sees the next file in the ranked list r (Line 7). Otherwise, the FSA stays in current state $S_{r[i]}$ (Line 8). After the loop, Algorithm 2 add the transition between the initial state and $r[0]$. Finally, the Algorithm 2 reduces F to remove duplicated states in F. This could be done with a standard process [33].

Algorithm 2 Building FSA

```
Require: A list of CFAP, $List$
Ensure: An FSA,$F$ for data reduction
 1: Build an empty FSA, $F$
 2: Add an initial state,$Init$, for $F$
 3: for all $cfap \in List$ do
 4:     Rank items in $cfap$ based on their file IDs in a list $r$
 5:     for all $i \in [0, r.length)$ do
 6:         create a state $S_{r[i]}$ in $F$
 7:         Add transition $(S_{r[i]}, r[i]) \to S_{r[i+1]}$ to $F$
 8:         Add transition $(S_{r[i]}, !r[i]) \to S_{r[i]}$ to $F$
 9:     end for
10:     Add transition $(Init, r[0]) \to S_{r[1]}$ to $F$
11:     Reduce $F$
12: end for
```

8.2 Retiring Invalid CFAPs

Our read-only detection method may have the "out of date template" problem. In our method, it is possible that while parsing the FAP and generating the CFAPs, a file is read-only. However, after a certain period of time, the file is modified and not read-only any more. In this case, the CFAPs that are related to the file are not valid after the modification to the file. This problem does not affect the reduction before the modification to the file. However, continuously using these CFAPs may break dependencies of the modified file.

To address this problem, the Reducer maintains a key-value store that maps the read-only files to their corresponding CFAPs. Once the Reducer finds an write event to a read only file, it removes the modified file from the key-value store and deletes all the CFAP related to the file.

8.3 Decompression Free Data Schema

Our reduced data support fast on-the-fly decompression in the causality analysis. The reduced data can be directly used in the causality analysis without slowing down the analysis speed. To do so, NodeMerge organizes the reduced data into two relational databases. The first is the event database, which stores the unmerged system events and the merged events for read-only files. The unmerged events contains the source, the destination, and other attributes. The special events have the same format as the unmerged events. The only difference is that instead of representing one system event, a special event represents a set of events that have been merged by a CFAP. Each special event has a unique ID for its used CFAP. The second database is the template database, which contains all the CFAPs and their corresponding events to read-only files. Each CFAP has a unique ID, which is used in the special events in the event database.

The two databases are opaque to the causality analysis. NodeMerge provides an API to hide the query details from causality analyses. While using the reduced data, causality analysis can directly use the provided interface to query the events and their dependencies. If the query is to a normal unmerged event, NodeMerge directly returns it to the causality analysis. If the query is to a special event, NodeMerge first retrieves the merged read-only file

Event type	Event count (proportion)
Process events	219, 425, 286 (5.26%)
File events	3, 760, 786, 832 (90.18%)
Network events	189, 649, 659 (4.55%)

Table 2: The distribution by system event types.

Hosts	Raw Data	Improve
Data Analytics Server	33.7X	11.2X
Big Data Processing Server	23.5X	9.0X
Simulation And Test Bed	17.5X	6.7X
Developer's Host 1	15.1X	6.3X
Developer's Host 2	11.6X	4.9X
Other Hosts	4.2X	2.3X

Table 3: Host level reduction.

	Raw Data	Improve	#File	#Process
Big Data Processing	21.5X	9.17X	155	853113
Data Analytics Software	75.7X	32.6X	927	630347
System Monitor	2.6X	1.3X	9	168148
Develop	13.5X	6.8X	45	425519
File Sharing	5.2X	1.9X	20	79857
System Util	1.9X	1.3X	18	329372
System Daemon	2.5X	1.8X	9	2593742
Database	3.2X	1.7X	57	815357
Communication	1.8X	1.3X	14	315478
Other	1.5X	1.1X	11	16974

Table 4: Workload Based Reduction.

events in the corresponding CFAP and returns them to the causality analysis.

9 EVALUATION

In our evaluation, we focus on answering following research questions:

- RQ 1: How effective is our approach in data reduction?
- RQ 2: What is the impact of our approach on the accuracy of system dependency analysis.
- RQ 3: What is the time and memory cost to learn the CFAPs?
- RQ 4: How fast is our approach compared to the the original FP-Growth algorithm?

9.1 Experiment Protocol

We implemented the agents to collect system event data using the Linux Audit framework [4] and report to the back-end database. We implemented the Data Collector, the Template Learner, and the Reducer using Python programming language. We implemented the FAP with Postgres Database. The Template Learner updates the CFAP every 24 hours. To experiment our data reduction approach, we deployed NodeMerge on a server with 16 core (Intel Xeon CPU E52640 v3 @ 2.60GHz), 102.4 GB memory. We also deployed our data collection agents to 75 hosts from NEC Labs America at Princeton. The deployment includes all hosts for the lab's daily activities which include R&D and administrative such as legal, accounting and so forth. Collected events are stored in a Postgres database.

Our data-set is collected from 01/06/2016 to 02/02/2016. In total, we have 4,169,861,777 events, and the size amounts to 1,337 GB.

9.2 Effectiveness in Data Reduction

To evaluate the effectiveness of our approach in data reduction, we measure the improvement of storage capacity by using NodeMerge. We define the improvement of storage capacity as $\frac{Size_{pre}}{Size_{post}}$, where $Size_{pre}$ is the size of hard disk to hold the system event data before applying NodeMerge and $Size_{post}$ is the size of hard disk after using NodeMerge. We measured two distinct types of improvement of storage capacity. First, we measured how much storage capacity was improved over the raw system event data. Second, we measured the improvement of storage capacity over the reduced data of the baseline approach [46]. We also measured the improvement of storage capacity on two different levels, one the host level and one the application level.

The improvement of storage capacity of on the host level is shown in Table 3, in which we summarize the improvement of storage capacity over 75 hosts. In Table 3, we can see that NodeMerge is particularly effective on the data analytics and data processing hosts. On average, NodeMerge improves the storage capacity on the raw data for 28.3 times and for 10.1 times on the data reduced by the baseline method for these two hosts. We studied the reason about this. We found that on these hosts, people tend to frequently and repeatedly use a similar set of programs, such as python, bash, perl, R, and matlab, to do machine learning and data processing tasks. In each execution, these program will load a fixed set of libraries and configuration files. Thus, there are strong patterns for NodeMerge to learn.

Similar behavior also exists in our test bed and some of the developer's desktops. In the test bed, experiments are constantly conducted. In the developer's desktops, developers constantly start the development tools, email clients, and git during their development. For the these hosts, NodeMerge can also substantially improve the storage capacity by 4.9 to 6.7 times over the data from the baseline method. For other hosts, the average improvement is 4.2 times on the raw data and 2.3 time on the data reduced by the baseline method. This means our approach can further save 56% of the price of hosting system event data even after the data has already been reduced by Xu et al.'s approach [46]. For a typical enterprise with 200,000 nodes, this could mean a saving of more than **half a million US dollars** per each year [36, 46].

On the application level, we categorized the all applications in the 75 hosts into 10 categories based on their workload. The result is in Table 4. Our categorization methodology is based on the approach in [46]. Specifically, "Big Data Processing" represents the tools to process data, such as bash and perl. "Data Analytics Software" contains the machine learning tools, such as python, R, and Matlab. "System Monitor" includes the auditing and monitoring tools. "Develop" includes the development tools such as Eclipse, Visual Studio, and compilers like GCC. "File Sharing" contains the applications used to share files such as SVN, CVS, and DropBox. "System Util" has the common utilities tools in Windows, like Disk Utilities. "System Daemon" represents daemon processes running in the background. "Database" includes these processes running as database services like MySQL, MangoDB, etc. "Other" includes all other types of applications.

As reported in Table 4, NodeMerge performs best on the machine learning, data processing, and development tools. Especially for the machine learning software, NodeMerge can improve the storage capacity for 75.5 times on the raw data and 32.6 times of the data reduced by the baseline method. To further explain the reason of reduction, we also report the average number of accessed files in the initial stage (*#File*) and the average numbers of instances (*#Process*) of each application in Table 4. In our experiment, two factors determine the reduction rate of NodeMerge: how many files are accessed in the initial stage and how many times do applications repeat. For the applications that access a lot of files in their initial stage and are repeated frequently, NodeMerge has a higher reduction rate.

This explains why it achieves the best result on our machine learning servers and some of the developers' host. The similarity between these applications is that they are frequently started and closed. Once people open an application for multiple times, NodeMerge can capture the initial stage of each execution and learn a pattern to merge them. The more initial stages, the more reduction.

In conclusion, NodeMerge is effective in reducing data, particularly for machine learning and development hosts. These hosts could be important parts of modern enterprises. By reducing data, an enterprise can potentially save a substantial amount of operation cost.

9.2.1 Impact of the Length of Initial Stage. To have a complete evaluation, we also tested the effectiveness of our approach under different configurations of the length of the initial stage. Remember that the initial stage is defined as the first k seconds of a process. Having a longer time of initial stage means increasing the length of CFAPs and thus can achieve more data reduction. On the other hand, a longer time of the initial stage also means that the Template Learner needs to process more data and thus can increase the overhead for data reduction.

In our experiment, we measured the improvement of storage capability of the raw data for NodeMerge with the initial stage as one to five seconds respectively. The average host level improvement of storage capacity for one, two, three, four, and five seconds were 3.8, 4.7, 4.74, 4,81, 4.88 respectively. The data reduction improved most significantly by extending the initial stage from one second to two seconds. Further extending the length did not improve the data reduction much. This result indicates that having a two-seconds long initial stage can already achieve near optimal data reduction ratio. Having longer initial stage will not improve the effectiveness of data reduction but increase the overhead.

9.3 Support of Causality Analysis

To evaluate how well can our approach keep the information in system dependency analysis, we simulated ten realistic attack cases during our experiment and performed the system dependency analysis both for the raw data and the reduced data. The attack data is contained in our data used in Section 9.2. Then, we compared the precision of the results of system dependency analysis on both data set. Since our data reduction was designed to support security causality analysis, we evaluated the precision of causality tracking results on reduced data. Without loss of generality, we focused on backtracking [21] for this experiment, as forward-tracking is the opposite of backtracking.

We conducted two experiments. In the first experiment, we tested our approach on real world cases. The first five cases were real-world attacks. The Email Phishing attack delivers the Excel attachment that drops and execute `Trojan` malware to a victim machine, and triggered Excel to launch the malware instance, which in turn created a backdoor to download other malware payloads. In the Info-Stealing incident, an attacker dumped databases from SqlServer and leaked the stolen data to a remote cite. The Shellshock case exploited the notorious Bash vulnerability and tricked an Apache server to fork a Bash terminal. To enable an attack based upon Netcat, an adversarial downloaded a Netcat utility, used it to open a backdoor, and further downloaded malware through this stealthy channel. Our Ransomware attack spawned multiple processes to simultaneously encrypt specific targeted files on the system. The other five test cases were collected from Xu et al's test suite [46]. These were normal system operations including file reduction using Gzip and Pbzip, account management using Useradd and Passwd, and downloading and compilation of source code via Wget and Gcc.

With the ten test cases, we collected both of the raw system events and corresponding reduced data and then ran backtracking on both data to produce causal graphs and compare their results. Due to the control over the test environment, we had full knowledge of the ground truth for these cases.

In this experiment, we compared the connectivity (i.e., edge) change of backtracking graphs before and after data reduction. The backtracking results show that, for all test cases, our data reduction can preserve original entities and connectivity and therefore does not affect the quality of tracking results.

In the second experiment, we randomly select 11,587 Point of Interest (POI) events from the data and apply backtracking to further investigate the impact of data reduction upon attack forensics. Our result demonstrates that backtracking results generated on reduced data were completely unaffected compared to those produced from raw data. This was fundamentally due to the fact that our templates captured and reduced the read-only files that were intensively loaded at program initialization but have little backward dependencies in recent past.

In all of our experiments, we also compared the execution time of the causality analyses on the reduced data. We found that there was **no statistical significant difference** between the execution time on the reduced data and on the raw data.

9.4 Learning Cost

We measured the time and memory cost spent by the Template Learner to learn the CFAPs under different lengths of the initial stage. The length of the initial stage varied from one to five seconds. In our experiment, the learning time for one to five seconds were 45, 60, 72, 77, and 80 minutes respectively. The memory cost for one to five seconds were 0.5, 1.3, 3, 5.5, and 14.5 GB.

As shown in our experiment, our approach took less than an hour to update the CFAPs. Since our approach updates the CFAP every 24 hours, this length of learning time could not cause any problems. The memory cost of our approach increases significantly when the length of initial stage increases. Increasing one second of initial stage often means about 2-3 times of increment of memory cost. Based on the result in Section 9.2.1, it indicates that prolonging the length of initial stage beyond two seconds is not efficient.

Note that the NodeMerge runs on a centralized server. The learning cost is spent only on the server instead of each host. Thus, we believe the computational cost of our system is acceptable.

9.5 Acceleration of FP-Growth

In this section, we evaluated how much learning time could be saved by our selective CFAP and FP-Tree pruning algorithms. To do this, we altered the Template Learner in our approach. In this experiment, the Template Learner used the original FP-Growth algorithm to find CFAP candidates. To avoid confusion in the Reducer, we also applied the CFAP selection algorithm in Section 7.2.1 to remove CFAP candidates that contained others or were contained by others. Then we followed the same protocol in Section 9.4 to measure the learning time of the original FP-Growth algorithm.

In our experiment, we found that the original FP-Growth could not finish learning in 24 hours under all different configurations. We then terminated our experiment since it is too costly to keep running the original FP-Growth algorithm. It was clear that the original FP-Growth algorithm was not capable of learning CFAPs in a reasonable amount of time. This result proves that our modification to the FP-Growth algorithm is necessary.

10 DISCUSSION

In this section, we discuss two aspects of our approach regarding preserving attack information after reduction and generality of this approach.

10.1 Preservation of System Dependency Information

Our approach is designed to keep the information after reduction in causality analysis. We achieve this goal by only merging the read-only files. The reduction in our method does not keep the temporal orders of the read only files. However, these orders do not affect the causality analyses because the read only files are always the source of paths in a causality graph, merging them together does not introduce any confusion in causality analysis. The capability of our approach to keep system dependency information is evaluated in details in Section 9.3.

10.2 Generality

Our approach is designed to be general for all types of data for causality analysis, or system event data [15, 21, 32, 37]. We make little assumption about the format of data, the types of hosts, the types of applications, or the infrastructure of enterprise systems. Hence, our approach can be applied to various kinds of systems and data.

We also show that our system can achieve significant storage capacity improvement with 1.3 GB memory. This configuration is modest in enterprise level servers. Thus, we believe our system can be widely applied to various enterprise environments.

10.3 Properties of the Algorithm

Our problem can be reduced to frequent itemset mining problems; thus, our problem is NP-Hard, and achieving optimal reduction is difficult. FP-Growth is one of the state-of-the-art algorithms for frequent itemset mining. As we proved in Section 7.2.2 the difference in pattern selection between our approach and FP-Growth is bounded by σ. In Section 9 we show that our approach is substantially faster than FP-growth.

Since our approach is based on the patterns of frequent itemset mining, the capability of reduction relies on the workload of hosts. We evaluated the quantified differences between the reduction rate on different types of hosts in Section 9.2.

10.4 Possible Attacks

NodeMerge is generally robust to attacks. When the attacker installs malware in a host of an enterprise, the malware will be considered as a non-read-only file. Thus, NodeMerge will not merge the events of the malware and it will not break the dependencies of the malware. Although the attacker can inject malicious code to an existing read only file, such an injection can cause a file write event to the read-only file and nullify the read-only property of the file. For any future events of the nullified read-only file, NodeMerge directly records the original events to keep the dependencies without any reduction.

There is one corner case for attackers to compromise NodeMerge. The attacker first uploads a malware to a host of an enterprise. Then the attacker waits for a long time without any malicious behaviors until the security manager of the enterprise decide to update the threshold of read only file detection, t_g, to a time that is newer than the creation time of the malware. After that, the attacker starts attacks through the installed malware. In this case, NodeMerge may potentially take the malware as a read only file and potentially break the dependencies of the malware.

The root cause of this threat is not NodeMerge but the fact that the security manager of an enterprise may want to retire the very old system events. In practice, it is not possible for an enterprise to store and maintain the system events for the whole enterprise forever due to the cost of storage and the high volume of daily generated system events. Very old system events that are not concerned by the security team should be retired for new data. Thus, even without NodeMerge, the attacker can always upload the malware, wait for a long enough time until the system events of the malware are retired, and then start the actual attack to bypass the possible security detection techniques in the target enterprise. To address this problem, the enterprise only need to keep the events for a longer and set t_g as a very old timestamp that can guarantee the integrity of the events before t_g.

10.5 Limitations

NodeMerge only compresses read-only files, and we chose to omit the write operations in our system. Such a design choice may reduce the capability of data reduction, e.g. NodeMerge may not be effective for hosts that contain a lot of write operations. In this case, a finer-grained storage reduction algorithm is needed, which may substantially increase the computational cost of reduction. Achieving the balance between the computational cost and the reduction capability is challenging and beyond the scope of this paper. We leave the techniques to compress write operations to future work.

NodeMerge is more effective on specific hosts than others using different sets of applications in different ways. As discussed in Section 9.2, NodeMerge shows higher reduction rate on the hosts for data analysis or big data processing tasks, where they frequently repeat the execution of the same applications. NodeMerge is not effective on applications that do not load many files or have file access patterns in their initial stages. By the enlarging of the analysis scope from processes' initial stage to the whole life-cycle of processes, our approach can achieve more reduction with higher computational cost. We leave this to our future work.

11 RELATED WORK

Only a few previous approaches focus on reducing the size of system event data for system dependency analysis. Xu et al. proposed an approach [46] to reduce system event data. The basic idea is to aggregate the time windows of events related to the same system objects (i.e., edges with the same source node and the same target node in the dependency graph), since these edges carry the same information about system activities except different time windows. Despite the effectiveness, this approach misses a lot of opportunities for data reduction. The time window aggregation based data reduction technique only handles the events with the same source and destination. It does not merge the events that are different in source or destination but have correlations. Similar to this approach, our approach can preserve the information in system dependency analysis. However, our approach can merge the file events that cannot be handled by Xu's approach. In our evaluation, our approach can improve the storage capacity of his approach for nearly two times.

Another closely related work is LogGC [25]. The idea of this work is to remove the logs for system objects that have little impact on system dependency analysis. The difference between our work and LogGC is that LogGC focuses on the lifespan of files while our work focuses on the correlation relationship between files. These two pieces of work focused on two different directions of system event reduction and can be coupled together to achieve further data reduction.

Prior efforts have also been made to prune out system events that are labeled either by operating systems (i.e., CABdedupe [41]) or user-specified rules [6]. In contrast, our approach is fundamentally different because we aim to automatically discover repeated templates in order to compress redundant data. ProTracer [31] reduced the size of logs by switching between tainting and logging. Bates et al. [5] extended the audit log framework to extract finer-grained information and thus to achieve more precise causality tracking. Unlike these approaches, which rely on additional kernel instrumentation, our approach works directly with kernel audit logs and thus is easy to deploy.

Causality analysis is an important technique in security. King et al. proposed an approach to backtrack system events [21] . Goel et al. also proposed a similar backtracking system for system recovery [15]. Sitaraman et al. improved the backtracking technique with more detailed system information logs [40]. Many other techniques were also proposed to apply the backtracking technique in different security related tasks, such as break-in detection, recovery, and risk analysis [3, 13–15, 19, 21, 23, 24, 30, 38]. Other approaches also use system dependency analysis to generate code [3], detect failures [7, 51, 52], identify user input [27], and classify malware [50]. Other techniques also rely on causality analysis to detect malware [48, 49]. However, none of these techniques focuses on reducing the data for system dependency analysis.

Han et al. first proposed the FP-Growth algorithm in the data mining community to mine frequent patterns [16]. Li et al. proposed a method to parallelize the FP-Growth algorithm [26]. Wang et al. proposed the top-down FP-Growth algorithm to optimize the speed of the original FP-Growth method [43]. Although these methods are effective in a general purpose of mining frequent patterns, they cannot be directly used for the big system log data as discussed in Section 9.5. Unlike these approaches, our work focuses on finding CFAPs from the system log data. Our approach made novel optimizations to ensure the performance of CFAPs learning by taking advantages of the features of CFAPs. We are not aware of other existing machine learning techniques that can do similar work, which is improving the storage capability of security event storage systems while maintaining the dependencies between system events.

There are several well known data compression algorithms, such as gzip [34], bzip2 [39], and other techniques [22, 47]. There are also other approaches to compress similar parts in graphs to save storage [9, 44]. The main problem for these techniques is that they require an explicit, non-trivial decompression process. However, if people need to frequently use the compressed data in the causality analysis, it is not realistic to decompress the data every time. On the contrary, the data compressed by our approach can be decompressed on the fly with a negligible runtime overhead.

12 CONCLUSION

In this paper, we propose a novel online template based approach to reduce system event data. Our approach learns fixed patterns of read-only file accesses and merges them into one special item in the reduced data. By doing so, our approach effectively reduces the volume of system event data while keeping the system dependency information in system event data. We conducted a thorough evaluation of our approach which improves the storage capacity for at most 11.2 times over the data reduced by the baseline approach. During the reduction, our approach on average took 1.3 GB memory. This overhead was reasonable for most modern enterprise level servers. The reduced data also preserved the accuracy of causality analysis in realistic attack cases.

13 ACKNOWLEDGEMENT

We thank the hard work of reviewers and the shepherd. This work was done by Yutao Tang as the intern at NEC Labs America. Yutao Tang and Ding Li equally contributed to this paper. Ding Li and Mu Zhang are corresponding authors. This work was partially supported by the NSFC-61872180, Microsoft Research Asia, Jiangsu "Shuangchuang" Talents Program, CCF-NSFOCUS "Kunpeng" Research Fund, and Alipay Research Fund.

REFERENCES

[1] 2018. The Cost of Tape vs. Disk Data Backup. http://www.backupworks.com/costofdiskvstape.aspx.

[2] abcNEWS. 2015. Anthem Cyber Attack. Retrieved August 2017 from http://abcnews.go.com/Business/anthem-cyber-attack-things-happen-personal-information/story?id=28747729

[3] J. A. Ambrose, J. Peddersen, S. Parameswaran, A. Labios, and Y. Yachide. 2014. SDG2KPN: System Dependency Graph to function-level KPN generation of legacy code for MPSoCs. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 267–273. https://doi.org/10.1109/ASPDAC.2014.6742901

[4] The Linux audit framework. 2016. https://wiki.archlinux.org/index.php/Audit_framework.

[5] Adam Bates, Wajih Ul Hassan, Kevin Butler, Alin Dobra, Bradley Reaves, Patrick Cable, Thomas Moyer, and Nabil Schear. 2017. Transparent Web Service Auditing via Network Provenance Functions. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*.

[6] Adam Bates, Dave (Jing) Tian, Grant Hernandez, Thomas Moyer, Kevin R. B. Butler, and Trent Jaeger. 2017. Taming the Costs of Trustworthy Provenance Through Policy Reduction. *ACM Trans. Internet Technol.* 17, 4 (Sept. 2017).

[7] Sören Bleikertz, Carsten Vogel, and Thomas Groß. 2014. Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 26–35.

[8] Tom Brant and Joel Santo Domingo. 2018. SSD vs. HDD: What's the Difference? https://www.pcmag.com/article2/0,2817,2404258,00.asp.

[9] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. 2008. Efficient Provenance Storage. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 993–1006. https://doi.org/10.1145/1376616.1376715

[10] DARKReading. 2011. Sony reports 24.5 million more accounts hacked. Retrieved August 2017 from http://www.darkreading.com/attacks-and-breaches/sony-reports-245-million-more-accounts-hacked/d/d-id/1097499

[11] Barbara Filkins. 2016. IT Security Spending Trends. https://www.sans.org/reading-room/whitepapers/analyst/security-spending-trends-36697.

[12] Forbes. 2017. Equifax Data Breach Impacts 143 Million Americans. https://www.forbes.com/sites/leemathews/2017/09/07/equifax-data-breach-impacts-143-million-americans/a7bd9db356f8.

[13] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. 2018. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 639–656. https://www.usenix.org/conference/usenixsecurity18/presentation/gao-peng

[14] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R. Kulkarni, and Prateek Mittal. 2018. AIQL: Enabling Efficient Attack Investigation from System Monitoring Data. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 113–126. https://www.usenix.org/conference/atc18/presentation/gao

[15] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. 2005. The Taser Intrusion Recovery System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 163–176. https://doi.org/10.1145/1095810.1095826

[16] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns Without Candidate Generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/342009.335372

[17] Ragib Hasan, Radu Sion, and Marianne Winslett. 2009. Preventing history forgery with secure provenance. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 12.

[18] Ragib Hasan, Radu Sion, and Marianne Winslett. 2009. Sprov 2.0: A highly-configurable platform-independent library for secure provenance. In *ACM Conference on Computer and Communications Security (CCS)*.

[19] Xuxian Jiang, A. Walters, Dongyan Xu, E. H. Spafford, F. Buchholz, and Yi-Min Wang. 2006. Provenance-Aware Tracing ofWorm Break-in and Contaminations: A Process Coloring Approach. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. 38–38. https://doi.org/10.1109/ICDCS.2006.69

[20] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs With SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. ACM, New York, NY, USA, 19–30. https://doi.org/10.1145/3052973.3053034

[21] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/945445.945467

[22] SR Kodituwakku and US Amarasinghe. 2010. Comparison of lossless data compression algorithms for text data. *Indian journal of computer science and engineering* 1, 4 (2010), 416–425.

[23] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. 2010. Trail of Bytes: Efficient Support for Forensic Analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, New York, NY, USA, 50–60. https://doi.org/10.1145/1866307.1866314

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

[25] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer; communications security (CCS '13)*. ACM, New York, NY, USA, 1005–1016. https://doi.org/10.1145/2508859.2516731

[26] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. 2008. Pfp: Parallel Fp-growth for Query Recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems (RecSys '08)*. ACM, New York, NY, USA, 107–114. https://doi.org/10.1145/1454008.1454027

[27] J. Liu, C. Fang, and N. Ansari. 2014. Identifying user clicks based on dependency graph. In *2014 23rd Wireless and Optical Communication Conference (WOCC)*. 1–5. https://doi.org/10.1109/WOCC.2014.6839915

[28] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *Proceedings of NDSS Symposium 2018*.

[29] Gordon Fyodor Lyon. 2009. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure.

[30] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. 2015. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2818000.2818039

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

[32] Microsoft. 2017. ETW events in the common language runtime. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.

[33] State Minimization/Reduction. [n. d.]. http://www2.elo.utfsm.cl/ lsb/elo211/aplicaciones/katz/chapter9/chapter09.doc2.html.

[34] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng. 2010. FPGA implementation of GZIP compression and decompression for IDC services. In *2010 International Conference on Field-Programmable Technology*. 265–268. https://doi.org/10.1109/FPT.2010.5681489

[35] Igor Pavlov. 2014. 7-zip. (2014).

[36] Amazon S3 Price. [n. d.]. https://aws.amazon.com/s3/pricing/.

[37] Redhat. 2017. The Linux audit framework. https://github.com/linux-audit/.

[38] M. Rezvani, A. Ignjatovic, E. Bertino, and S. Jha. 2014. Provenance-aware security risk analysis for hosts and network flows. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. 1–8. https://doi.org/10.1109/NOMS.2014.6838250

[39] Julian Seward. 1998. bzip2.

[40] S. Sitaraman and S. Venkatesan. 2005. Forensic analysis of file system intrusions using improved backtracking. In *Third IEEE International Workshop on Information Assurance (IWIA'05)*. 154–163. https://doi.org/10.1109/IWIA.2005.9

[41] Y. Tan, H. Jiang, D. Feng, L. Tian, and Z. Yan. 2011. CABdedupe: A Causality-Based Deduplication Performance Booster for Cloud Backup Services. In *2011 IEEE International Parallel Distributed Processing Symposium*.

[42] Techcrunch. 2017. Target Says Credit Card Data Breach Cost It 162M Dollars In 2013-14. https://techcrunch.com/2015/02/25/target-says-credit-card-data-breach-cost-it-162m-in-2013-14/.

[43] Ke Wang, Liu Tang, Jiawei Han, and Junqiang Liu. 2002. *Top Down FP-Growth for Association Rule Mining*. Springer Berlin Heidelberg, Berlin, Heidelberg, 334–340. https://doi.org/10.1007/3-540-47887-6_34

[44] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell D.E. Long. 2012. A Hybrid Approach for Efficient Provenance Storage. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM '12)*. ACM, New York, NY, USA, 1752–1756. https://doi.org/10.1145/2396761.2398511

[45] Xi Xu, Rashid Ansari, Ashfaq Khokhar, and Athanasios V. Vasilakos. 2015. Hierarchical Data Aggregation Using Compressive Sensing (HDACS) in WSNs. *ACM Trans. Sen. Netw.* 11, 3, Article 45 (Feb. 2015), 25 pages. https://doi.org/10.1145/2700264

[46] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 504–516. https://doi.org/10.1145/2976749.2978378

[47] En-Hui Yang and John C Kieffer. 2000. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. I. Without context models. *IEEE Transactions on Information Theory* 46, 3 (2000), 755–777.

[48] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. 2014. Detection of Stealthy Malware Activities with Traffic Causality and Scalable Triggering Relation Discovery. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '14)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2590296.2590309

[49] Hao Zhang, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Zhibin Zhang. 2016. Causality Reasoning About Network Events for Detecting Stealthy Malware Activities. *Comput. Secur.* 58, C (May 2016), 180–198. https://doi.org/10.1016/j.cose.2016.01.002

[50] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1105–1116. https://doi.org/10.1145/2660267.2660359

[51] M. H. Zibaeenejad and J. G. Thistle. 2015. Dependency graph: An algorithm for analysis of generalized parameterized networks. In *2015 American Control Conference (ACC)*. 696–702. https://doi.org/10.1109/ACC.2015.7170816

[52] T. Zimmermann and N. Nagappan. 2007. Predicting Subsystem Failures using Dependency Graph Complexities. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. 227–236. https://doi.org/10.1109/ISSRE.2007.19