High Fidelity Data Reduction for Big Data Security Dependency Analyses

Zhang Xu[†1]
zhang@nofutznetworks.com

Zhenyu Wu[‡]
adamwu@nec-labs.com

Zhichun Li[‡]
zhichun@nec-labs.com

Kangkook Jee[‡]
kjee@nec-labs.com

Junghwan Rhee[‡]
rhee@nec-labs.com

Xusheng Xiao[‡]
xsxiao@nec-labs.com

Fengyuan Xu[◇]
fengyuan.x@gmail.com

Haining Wang[∗]
hnw@udel.edu

Guofei Jiang[‡]
gfj@nec-labs.com

† NofutzNetworks Inc., Croton-on-hudson, NY, USA
‡ NEC Labs America, Inc., Princeton, NJ, USA
◇ Nanjing University, Nanjing, China
∗ University of Delaware, Newark, DE, USA

ABSTRACT

Intrusive multi-step attacks, such as Advanced Persistent Threat (APT) attacks, have plagued enterprises with significant financial losses and are the top reason for enterprises to increase their security budgets. Since these attacks are sophisticated and stealthy, they can remain undetected for years if individual steps are buried in background "noise." Thus, enterprises are seeking solutions to "connect the suspicious dots" across multiple activities. This requires ubiquitous system auditing for long periods of time, which in turn causes overwhelmingly large amount of system audit events. Given a limited system budget, how to efficiently handle ever-increasing system audit logs is a great challenge.

This paper proposes a new approach that exploits the dependency among system events to reduce the number of log entries while still supporting high-quality forensic analysis. In particular, we first propose an aggregation algorithm that preserves the dependency of events during data reduction to ensure the high quality of forensic analysis. Then we propose an aggressive reduction algorithm and exploit domain knowledge for further data reduction. To validate the efficacy of our proposed approach, we conduct a comprehensive evaluation on real-world auditing systems using log traces of more than one month. Our evaluation results demonstrate that our approach can significantly reduce the size of system logs and improve the efficiency of forensic analysis without losing accuracy.

[1]Work done during an internship in NEC Labs America, Inc.

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978378

1. INTRODUCTION

Today's enterprises are facing serious cyber-threat posed by intrusive multi-step attacks such as Advanced Persistent Threat (APT) attacks. It takes much time for attackers to gradually penetrate into an enterprise, to understand its infrastructure, to own the high-value targets, and to steal important information [1, 6, 2, 9, 10, 12] or to sabotage mission critical infrastructures [11]. Compared with conventional attacks, sophisticated multi-step attacks such as APT attacks usually inflict much more severe damage upon the enterprises' business. The recent DARPA Transparent Computing (TC) program [13] emphasizes that the challenges of multi-step attacks come from modern computing systems being large, complex and opaque, and the threats can remain undetected for years if individual steps are buried in background "noise." Thus, to counter these sophisticated attacks, enterprises are in great need of solutions to "connect the dots" across multiple activities that individually might not be suspicious enough. Even though these attacks can be powerful and stealthy, one typical constraint from the attacker side is that such an attack needs multiple steps to succeed, such as infrastructure reconnaissance and target discovery, as illustrated in the cyber kill chain [5]. Therefore, multiple attack footprints might be left as "dots."

In order to achieve the vision of connecting the dots, the first challenge is to collect and store the dots (attack provenance). Since the attackers have the potential to reach each host, we need to monitor and collect attack provenance from every single host. According to [14], in 2014 APT attacks penetrated enterprises and remained stealthy for 188 days on average and even years before launching harmful harvesting actions. This implies that in order to detect and understand the impact of such attacks, the enterprises need to keep a half year to one year worth of data. However, monitoring attack provenance on every host in an enterprise for more than a half year is burdensome and laborious. The system level audit data collected from Linux Audit system [8] or Window Event Tracing framework [7] alone can easily reach 0.5 to 1GB per host. In a real-world scenario, for a commercial

bank with 200,000 hosts, the required data is around 17PB (half year 0.5GB per host) to 70PB (one year 1GB per host).

Furthermore, to achieve fast dependency correlation analysis or to promptly connect the dots even interleaved with multiple normal activities, the data has to be efficiently stored and indexed. Therefore, meeting these storage and retrieval requirements for such a big-data system is a daunting task. Very little previous research focuses on how to mitigate the problem of overwhelmingly big-data of attack provenance derived from system-level audit events for dependency analysis. LogGC [31] observes that some system objects such as temporary files are isolated, have short life-spans, and have little impact on the dependency analysis; such objects and events can be garbage collected to save space. In our approach, we take a different perspective and primarily study the impact of different events. We found that some events have identical dependency impact scope, and therefore they can be safely aggregated. Based on this observation, we propose an algorithm called Causality Preserved Reduction (CPR) for event aggregation. Furthermore, we found that certain popular system behaviors lead to densely connected dependency graphs of objects and their related neighbors. Thus, we devise a causality-preserving one-hop graph reduction technique called Process-centric Causality Approximation Reduction (PCAR), which can further improve data reduction with very few false positives.

To validate the efficacy of our proposed approach, we conduct a comprehensive evaluation on real-world auditing systems using log traces of more than one month. Our evaluation results demonstrate that the CPR algorithm is general and does not lose any accuracy by design, which can improve the capacity of a big-data system by up to 3.4 times. In other words, the same system can store 3.4 times more data without affecting storage and data processing efficiency. With a trade off of introducing very few false positives (in our evaluation at the rate of 0.2%), the PCAR algorithm can enlarge the capacity of the system by up to 5.6 times. We also compared our approach with a naive time-window-based data aggregation. We show that the naive approach introduces many more false positives at the rate of 13.9%. Without considering manually tuning our approach for each application in an enterprise environment, we achieve a similar reduction ratio as LogGC, but our solution can be combined with LogGC to achieve more significant data reduction.

The major contributions of this paper can be summarized as follows:

- Through intensive data collection and analysis, we observe that some events have identical contributions to dependency analysis and thus can be aggregated. Based on this observation, we propose the CPR algorithm for data reduction.
- We further observe that some high-level system behaviors such as PCI scanning will result in dense one-hop dependency graphs, and such a subgraph as a whole is more useful than its internal structures. Thus, we propose the PCAR algorithm, which can achieve even higher data reduction rates with few false positives.
- We conduct a comprehensive evaluation over extensive datasets collected from a real enterprise environment for one month. The evaluation results demonstrate that our approach can improve the efficiency of forensic analysis up to 5.6 times without losing accuracy.
- Our work will facilitate the detection of multi-step attack behaviors that involve multiple steps of malicious behaviors (*i.e., dots)* whose footprints are buried inside data gathered from multiple hosts over a long period of time.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivates our work. Section 3 provides the definitions and design details of our work. Section 4 presents our real-world data driven evaluation. Section 5 discusses attack resilience and generality of our approach. Section 6 surveys related works, and finally, Section 7 draws the conclusion.

2. BACKGROUND AND MOTIVATION

In this section, we briefly introduce the basic concept of a system dependency graph and causality tracking. Then, we present our observation of data characteristics of low-level system event traces. Finally, we provide an intuition-based description of how to reduce event trace data while preserving its embedded causal dependencies.

2.1 System Dependency Analysis

A system event trace is a sequence of recorded interactions among multiple system objects (processes, files, network connections, etc.). It contains information, such as timing, type of operation, and information flow directions, which can be used to reconstruct causal dependencies between historical events. For ease of discussion, we use the terms *causality* and *dependency* interchangeably in this paper.

Intuitively, we present a system event as an edge between two nodes, each representing the initiator or the target of the interaction. For example, a process named "`/bin/bash`" reads a file named "`/etc/bashrc`", is represented as node A, B and edge e_{BA-1}, shown in Figure 1 denoting that the *read* operation from A to B occurred during the time interval of $[10, 20]$.

While each node and edge carries many pieces of information, such as process ID and file permissions, for simplicity, we only show the ones critical to causality analysis. This includes the type of event, presented as text under the edge name; the window of time the event took place, presented as a pair of numbers in square brackets, denoting the start and end timestamps, respectively; and the information flow direction, presented by the direction of the edge. Note that the dependency graph allows multiple edges for a node pair to distinguish events that happened at different time intervals. For instance, edges of e_{BA-1} and e_{BA-2} represent the same type of event that occurred in different time durations.

Illustrated by Figure 1, a trace of system events form networks of causality dependencies, (*i.e.,* the dependency graph). The dependency graph is essential to many forensic analysis applications, such as root cause diagnosis, intrusion recovery, attack impact analysis and forward tracking [28], which performs causality tracking on the dependency graph. Causality tracking is an recursive graph traversal procedure, which follows the causal relationship of edges either in the forward or backward direction. For example, in Figure 1, to examine the root cause of event e_{AD-1} ("`/bin/bash`" executes "`/bin/wget`"), we apply backtracking [27] on our Point-of-Interest (POI) edge, which recursively follows all edges that could have contributed to the occurrence of POI. Edges of e_{BA-1}, e_{CA-1}, e_{CA-2}, and e_{EC-1} have an earlier starting

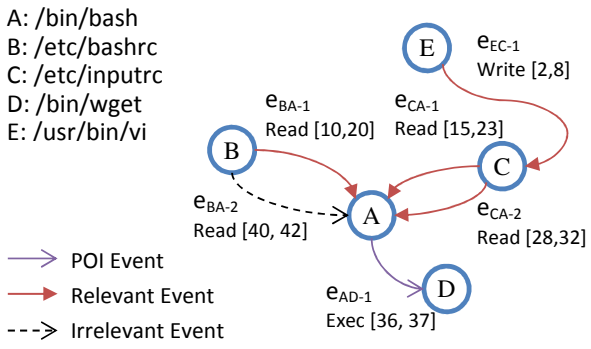

Figure 1: Dependency Graph and Backtracking

time than their predecessors and are therefore identified as relevant events.

2.2 Data Characteristics

The accuracy of system dependency analysis heavily depends on the granularity of event trace – the more finely grained the data, the more accurate the analysis results.

We built a ubiquitous monitoring system for enterprise security analyses, which collects low-level system events from Linux and Windows machines using kernel audit [8] and ETW kernel event tracing [7], respectively. We observe that an average desktop computer produces more than 1 million events per day, while a server could yield 10 to 100 times the volume. Each day, a rather small system of 100 computers generated more than 200 million events, which requires mid- to high-end server to process and produces databases over 200GB. A large enterprise can easily have more than 100,000 computers, and may require them to store the monitored data for several months or even years. Such a workload made it impractical to deploy an accurate dependency analysis to large enterprises. Reducing the data volume is key to solving the scalability problem.

While data reduction is a well-studied topic, the most commonly used techniques, spatial and temporal sampling, are not applicable to dependency analysis based on system event traces. Because sampling techniques do not have the inherit concept of causal relations, they are prone to introducing random causal errors. On the other hand, the recursive nature of causality tracking exponentially magnifies errors on the causal path – a single falsely introduced dependency when tracked forward or backward several hops could easily leads to hundreds of false positives. Therefore, any practical data reduction on system event traces must take great care to limit the introduction of causal errors.

2.3 Data Reduction Insights

Recognizing their critical importance, we first explore all possibilities to perform data reduction while perfectly preserving the causal dependencies. By studying causal relations of every event in our system traces, we discover that only a small fraction of events, which we call "key events", bear causal significances to other events. When performing forensic analysis on multi-step attacks, the key events reveal the footprint of an attacker (i.e., the sequences of activities an attacker triggers). Moreover, for each "key event" there exist a series of "shadowed events" whose causal relations to other events are negligible in the presence of the "key event", that is, the presence or absence of "shadowed events" does not alter any causality analysis. In multi-step attack forensic analysis, shadow events describe the same attacker activities that have already been revealed by key events. Therefore, we could significantly reduce the data volume while keeping the causal dependencies intact, by merging or summarizing information in "shadowed events" into "key events" while preserving causal relevant information in the latter.

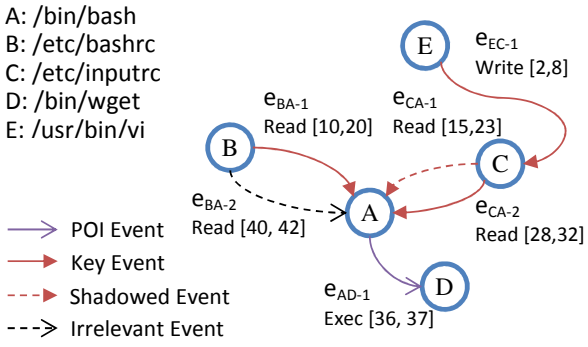

Figure 2: Unequal Dependencies in Backtracking

Figure 2 shows an example of unequal dependencies revisiting our backtracking example shown in Figure 1. In this figure, we backtrack POI event e_{AD-1} to e_{BA-1}, e_{CA-1}, e_{CA-2}, and e_{EC-1}. However, note that e_{CA-2} took place after e_{CA-1}, while no event involves either A or C happening in between, and both events are of the same type (*Read*). As a result, the existence of e_{CA-1} in this graph has no causal impact to the backward dependency analysis. In other words, the presence of the "key event" e_{CA-2} shadows the event e_{CA-1}. Therefore, e_{CA-1} can be removed by combining its information with e_{CA-2}, preserving the end time of the latter.

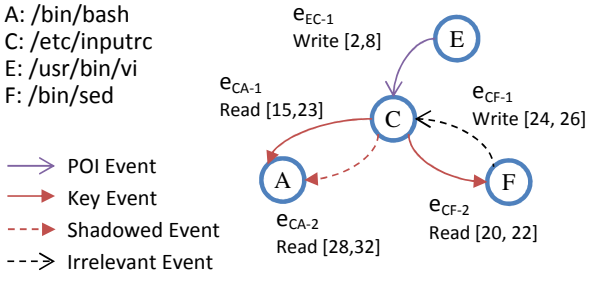

Figure 3: Unequal Dependencies in Forward-tracking

In Figure 2, we apply similar logic to identify unequal dependencies in forward-tracking analysis [28] where edge e_{EC-1} represents a POI event. For two edges that connects nodes C and A, event e_{CA-1} took place before e_{CA-2}, while no event involves either A or C happening in between; both events are of the same type and therefore "key event" e_{CA-1} shadows event e_{CA-2}.

When bi-directional tracking is concerned, the condition for event shadowing become more strict. Figure 4 combines backward and forward tracking into a single data dependency graph defining POI event as e_{HB-1} and e_{AD-1}, respectively. While "key event" e_{CA-2} shadows the event e_{CA-1} for both

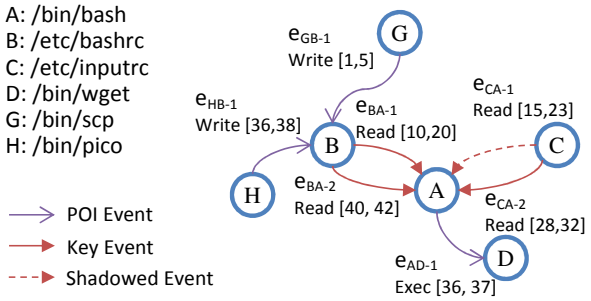

Figure 4: Dependency Reduction in Bi-directional Tracking

analyses, event e_{BA-1} and e_{BA-2} must be kept as independent "key events," since each event defines causal relation from different analyses.

2.3.1 Low-loss Dependency Approximation

With further study of our data, we discover that several applications (mostly system daemons) tend to produce intense bursts of events that appears semantically similar, but are not reducible by the perfect causality preserving reduction, due to interleaved dependencies. For example, process "pcscd" repeatedly interleaves read/write access to a set of files. We name this type of workload "`iBurst`."

We performed in-depth analysis of those applications and found that each `iBurst` is generated by an application performing a single high-level operation, such as checking for the existence of PCI devices, scanning files in a directory, etc. Those high-level operations are not complex, but they translate to highly repetitive low-level operations. With `iBurst`, data reduction is only possible with certain levels of loss in causality. We then analyse whether and to what extent causality loss is acceptable. From the causality analysis applications' perspective, tracking down to the high-level operations usually yields enough information to aid the understanding of the results. For example, "pcscd" checks for all PCI devices, or "dropbox" scans the directory. Although obtaining precise low-level operation dependencies does yield more information, the extra data usually do not add more value. Therefore, accuracy loss seems acceptable as long as we contain the impact of the errors so they do not affect events that do not belong to the burst.

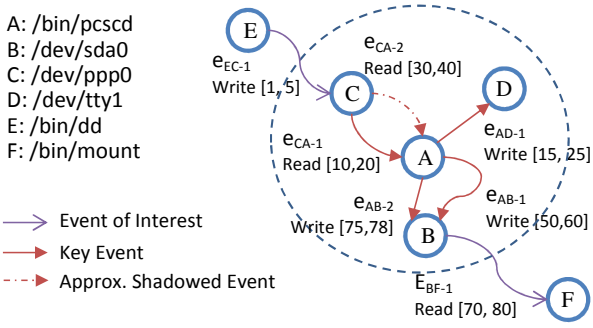

Figure 5: Dependency Approximation Reduction Example

We thus devise a method to detect an `iBurst` and apply a well-controlled dependency approximation reduction, which ensures that causality loss only impacts events within the `iBurst`. Shown in Figure 5, the "/bin/pcscd" process generates an `iBurst` involving many files, including "/dev/sda0," "/dev/ppp0," and "/dev/tty1," highlighted by the dashed circle. By ignoring the causal relationship among all events within the `iBurst`, event e_{CA-2} is considered approximately shadowed by e_{CA-1}, even though they are interleaved by e_{AD-1}. However, e_{AB-1} and e_{AB-2} must be kept as independent key events because they are interleaved by e_{BF-1}, which does not belong to the `iBurst`.

3. DESIGN

In this section, we first present a formal definition of important terms and concepts used in our design. Then we detail our algorithm used in data reduction. Our approach provides two schemes that serve to different reduction goals. While the primary scheme prioritizes perfect dependency preservation, the optional secondary scheme performs causality approximating reduction to gain better reduction rates at the expense of limited dependency accuracy loss. Finally, we describe an extension to incorporate domain knowledge-based data reduction.

3.1 Definitions

The formal definitions of three key concepts that are key to our data reduction design are given below.

For the rest of the paper, $tw(e)$ is used to represent the time window associated with an event, $ts(e)$ and $te(e)$ defines the start and end times of event e, respectively. We use the terms event and edge, entity and node interchangeably.

3.1.1 Causality Dependency

Two events have causality dependency with each other if an event has information flow that can 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.

3.1.2 Event Trackability

While each event has causality with other events in the system, we use a term called trackability to summarize the causality dependency information contained in an event. We define trackability as the forensic analysis results that can be derived from an event (i.e. the backtracking and forward-tracking results with the event as an POI event). A "key event" and its "shadowed events" have equal trackability.

DEFINITION 2: **Causality Shadowing (Partial Trackability Equivalence)**.

Given two event edges across the same pair of nodes $e_1 = (u_1, v_1)$ and $e_2 = (u_1, v_1)$, where $te(e_2) > te(e_1)$:

- e_2 shadows the backward causality of e_1, denoted as $e_2 \approx_B e_1$, if and only if there exists no event edge $e = (u_2, v_2)$ that satisfies all of:
  - $u_2 = u_1$ and $v_2 \neq v_1$,
  - $te(e) > te(e_1)$ and $ts(e) < te(e_2)$.

  We further explain this definition in Section 3.2.1.

Algorithm 1 Causality Preserved Aggregation

```
Require: E is event stream sorted by start time in ascending order
  function CPR_AGGREGATE(E)
    for each e in E do
      u←src(e)
      v←dst(e)
      Let S(u,v,R(e)) be a stack of events from u to v that are aggregable
      if S.Empty then
        S.push(e)
      else
        e_p←S.pop
        if FORWARD_CHECK(e_p,e,v) and
           BACKWARD_CHECK(e_p,e,u)
        then
          e_p←MERGE(e_p,e)
          S.push(e_p)
        else
          S.push(e)
        end if
      end if
    end for
  end function
  function MERGE((e_p, e_l))
    te(e_p)←te(e_l)
    Tune attributes of e_p
    DELETE(e_l)
    return e_p
  end function
```

Algorithm 2 Backward Trackability Check

```
  function BACKWARD_CHECK(e_p,e_l,u)
    for each e of u.INCOMING_EVENT do
      if tw(e) overlap with [te(e_p),te(e_l)] then
        return false
      end if
    end for
    return true
  end function
```

Algorithm 3 Forward Trackability Check

```
  function FORWARD_CHECK(e_p,e_l,v)
    for each e of v.OUTGOING_EVENT do
      if tw(e) overlap with [ts(e_p),ts(e_l)] then
        return false
      end if
    end for
    return true
  end function
```

- e_1 shadows the forward causality of e_2, denoted as $e_1 \approx_F e_2$, if and only if there exists no event edge $e = (u_2, v_2)$ that satisfies all of:
  - $u_2 \neq u_1$ and $v_2 = v_1$,
  - $te(e) > ts(e_1)$ and $ts(e) < ts(e_2)$,

  We further explain this definition in Section 3.2.2.

DEFINITION 3: **Full Trackability Equivalence**.

Given two event edges e_1 and e_2, they are fully equivalent in trackability, denoted as $e_1 \approx e_2$, if and only if $e_2 \approx_B e_1$ and $e_1 \approx_F e_2$.

In other words, for two events e_1 and e_2 that have both backward and forward trackability equivalence, with either e_1 or e_2 removed from the dependency graph, forward or backward causality tracking from any POI event (other than e_1 and e_2) would yield identical results (except e_1 and e_2). As a result, e_1 and e_2 are completely equivalent from the causality analyses' point-of-view.

3.1.3 Event Aggregability

Two events are *aggregable* only if they have the same type and share the same source and destination entities. For certain types of events such as file read/write, we also require that the two events share the same certain attributes, (*e.g.,* the file open descriptor). A set of aggregable events is a superset of a "key event" and its "shadowed events."

DEFINITION 4: **Event Aggregability**.

Given an event e and a set of trackbility equivalent events E, where $\forall e_i, e_i \approx e$:

- Define a set of attributes A to identify the aggregated event with acceptable level of granularity.
  For example, A can include subject and object attributes (i.e., process executable name, file path name), the event type (i.e., file access, IP channel access), and type-specific attributes, such as file access operation (i.e., read, write or execute).
- Event ε is aggregable with e, denoted as $\varepsilon \sim e$, if and only if $\varepsilon \in E$ and $a_i(\varepsilon) = a_i(e), \forall a_i \in A$.

For a given definition, without loss of generality, if $e_1 \sim e_2$ and $e_2 \sim e_3$, then $e_1 \sim e_3$.

3.2 Causality Preserved Reduction

Our primary reduction scheme aims at preserving causality dependency while performing the data reduction. Therefore, we name it Causality Preserved Reduction (CPR). The core idea is to aggregate all those events that are *aggregable* and share the same *trackability*.

Algorithm 1 shows the working procedure of CPR, in which a stream of events sorted by start time are taken as input. We maintain a stack for each type of event between a pair of entities. Every time we observe an event between the same pair of entities with same type, we check whether the events can be aggregated with the event in stack. This aggregation checking includes the examination of both forward and backward *trackability*.

Algorithm 2 describes the procedure to check the backward *trackability*. For two events e_1 and e_2 from entity u to entity v, they have the same backward *trackability* if all the time windows of incoming events of u do not overlap with the time window of [te(e_1),te(e_2)]. Algorithm 3 describes the procedure to check the forward-*trackability*. For two events e_1 and e_2 from entity u to entity v, they have the same forward *trackability* if none of outgoing events of v has an end time between the start times of e_1 and e_2.

If two events can be aggregated, we aggregate the event with a later start time (*i.e.,* the later event) to the event with an earlier start time (*i.e.,* the former event) by extending the end time of the former event to the end time of the later event and then discard the later event.

3.2.1 Backward trackability

According to our backward algorithm, for two aggregatable events e_1 and e_2 from u to v, they have the same backward trackability only if none of the incoming events of u has a time window overlapped with the time window between the end times of e_1 and e_2.

For any incoming event e of entity u, its timing can fall into one of the following three cases, illustrated in Figure 6:

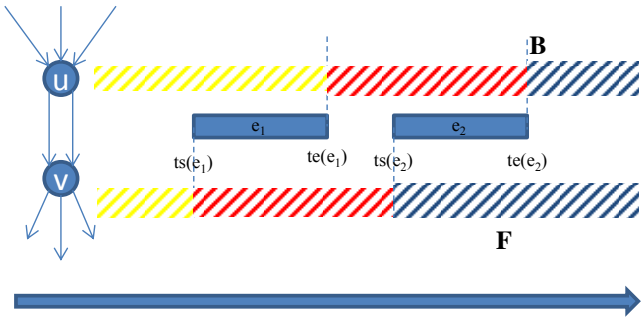

Figure 6: Determine whether to aggregate two events between u and v.

1. te(e)<te(e_1) (*i.e.,* the time window of e only overlaps with the yellow shadow of Area B).

2. te(e)>te(e_1) and ts(e)<te(e_2) (*i.e.,* the time window of e has an overlap with the red shadow of Area B).

3. ts(e)>te(e_2) (*i.e.,* the time window of e only overlaps with the blue shadow of Area B). By selecting e_1 as a POI event, the threshold will be set to te(e_1); by selecting e_2 as a POI event, the threshold will be set to te(e_2).

In the first case, e has dependency with both e_1 and e_2. According to the backtracking algorithm, the event will be included into the result set no matter whether we choose e_1 or e_2 as a POI event. Moreover, the threshold will be updated to the end time of the incoming event of u in both backtracking situations. The second case has another two sub-cases: *a)* ts(e)>te(e_1) and *b)* ts(e)<te(e_1). In *a)*, the event only has dependency with e_2, and thus choosing e_1 or e_2 as a POI event generates different backtracking results, which will break the trackability equivalence of e_1 and e_2. In *b)*, although the event has dependency with both e_1 and e_2, selecting e_1 as a POI event will make the populated threshold te(e_1) while selecting e_2 as a detection point will make the populated threshold $\min\{te(e_1), te(e)\}$. Such a situation will possibly make backtracking results different in the subsequent iterations, which also breaks the trackability equivalence of e_1 and e_2. In the third case, the event has no dependency with e_1 or e_2, implying that it will not be included in backtracking results anyway. As we can see, only case 2 can break the backward trackability equivalence. Thus, our algorithm will stop aggregating the two events if case 2 is reached.

3.2.2 Forward trackability

According to our forward algorithm, for two aggregatable events e_1 and e_2 from u to v, they have the same forward trackability only if none of the outgoing events of entity v has a time window overlapped with the time window between the start times of e_1 and e_2.

For any outgoing event e of entity v, its timing can fall into one of the following three cases, illustrated in Figure 6:

1. te(e)<ts(e_1) (i.e., the time window only overlaps with the yellow shadow of Area F).

2. te(e)>ts(e_1) and ts(e)<ts(e_2) (i.e., the time window overlaps with the red shadow of Area F).

3. ts(e)>ts(e_2) (i.e., the time window only overlaps with the blue shadow of Area F).

In the first case, e has no dependency with e_1 or e_2, so it will not be included in forward-tracking results. The second case also has two sub-cases: *a)* te(e)<ts(e_2) and *b)* te(e)>ts(e_2). In *a)*, the event only has dependency with e_1, breaking the trackability equivalence of e_1 and e_2. In *b)*, the threshold populated with e_1 as a POI event will be $\max\{ts(e_1), ts(e)\}$ while the threshold populated with e_2 as a POI event will be ts(e_2). Different populated thresholds will also break the trackability equivalence. In the third case, e has dependency with both e_1 and e_2 and the populated threshold is ts(e).

As we can see, only case 2 can break the forward trackability equivalence. Thus, our algorithm will stop aggregating the two events if case 2 is reached.

3.2.3 Tracking with an aggregated event

Given that e_1 and e_2 pass the backward and forward trackability equivalence checking of our algorithms, they will be aggregated and replaced with a new event e'. All the attributes of e_1 and e_2 will be copied to e' except that the time window of e' will become [ts(e_1), te(e_2)]. If e' is selected as an POI event, the backtracking threshold will be te(e_2) and the forward tracking threshold will be ts(e_1). Because all incoming events of entity u and all outgoing events of v satisfy the conditions set by our algorithms, the backtracking and forward-tracking results with e' as the POI event should be the same as those with e_1 or e_2 as the POI event.

3.2.4 Impact on tracking with other POI events

When we apply our data reduction, we attempt to aggregate all the events that share the same trackability. Before an attack is revealed, we cannot know what events will be selected as POI events; therefore, our approach should work equally on events no matter whether they will be selected as POI events.

To continue the example above, assume we start in another POI event to perform backtracking. While reaching the step we need to check the incoming events of entity v, we have a threshold th. Using data without reduction, we need to check events e_1 and e_2, but with the reduced data we need to check event e$'$. Since no outgoing events of v have the end time falling into the range of [ts(e_1), ts(e_2)], according to the threshold population algorithm, th should not fall into this range either. This leaves two cases for th: th <ts(e_1) and th >ts(e_2). In the first case, with the raw data both e_1 and e_2 will fail the dependency check and will not be included in the backtracking graph. With the reduced data, e$'$ will not pass the dependency check either, yielding the same results with the data without reduction: entity u and the events from u to v will not appear in the backtracking results. In the second case, with the raw data, both e_1 and e_2 will be included in the backtracking graph and the threshold will be populated.

Note that since all events that happen before e_1 will also happen before e_2, it means e_1 will be shadowed by e_2 while performing backtracking with other POI events in this case. Therefore we only need to populate the threshold with e_2, which would be $\min\{th, te(e_2)\}$. With the reduced data, e$'$ will be included with the populated threshold $\min\{th, te(e_2)\}$. Therefore, entity u will be included in the backtracking re-

Algorithm 4 Process-centric approximation reduction

Require: E is a stream of events involving hot process and its neighbours N(u)
Require: E is sorted by start time
function PCAR_AGGREGATE(E)
Let Q be a queue holding events in N(u)
Q←∅
for each e **do**
s←src(e)
t←dst(e)
if s∉N(u) **then**
N.IN$_{DEADLINE}$←te(e)
else if t∉N(u) **then**
N.OUT$_{DEADLINE}$←te(e)
else
CLEARSTATE(N(u),e)
end if
end for
return N(u)
end function

sults with both raw data and reduced data. The connectivity between u and v will also be kept with the data reduction.

The same procedure can be applied to forward tracking. Since no incoming events of u can have the start time in [te(e_1), te(e_2)], in the tracking process while reaching u, the threshold cannot fall into [te(e_1), te(e_2)]. Therefore, in forward-tracking of such a case, e_2 will be shadowed by e_1, and aggregating e_1 and e_2 will preserve the entities and connectivity in tracking results.

3.3 Process-centric Causality Approximation Reduction

Our secondary data reduction scheme, Process-centric Causality Approximation Reduction (PCAR), aims to reduce data from intensive bursts of events with interleaved dependencies, which are otherwise not reducible without dependency loss. PCAR constrains the causal dependencies compromised within the events within the burst, and achieves data reduction with a very limited impact to the dependency analyses.

We define a process that interacts with a large number of objects in a short time a *hot process*. Hot processes can be detected using a simple statistics calculation with a sliding time window If the number of events related to a process in a time window exceeds a certain threshold, the process is marked as a hot process. In our evaluation, we set the threshold to 20 events per 5 seconds. Once a hot process is detected, we collect all objects involved in the interactions, and form a neighbour set $N(u)$, where u is the hot process. We name the set *ego-net*. Illustrated in Algorithms 4, 5, and 6, instead of equally checking the trackability on all aggregation candidates based on the events, we only check the trackability with the information flow into and out of the neighbour set $N(u)$. The checking procedure is the same as CPR in checking the time window overlap. It can ensure that as long as the events inside the ego-net are not selected as a POI event, we can achieve high-quality tracking results.

It is noteworthy that PCAR algorithm does not reserve the trackability inside the ego-net. In a rare case that an event inside the ego-net is selected as a POI event, it has a chance that PCAR has aggregated the event, which results in an enlarged time window of the POI event. Due to the nature of forensic analysis, an enlarged time window can introduce false positives in tracking results.

Algorithm 5 Aggregate hot process events

function CLEARSTATE(N(u),e)
Let S(s,t,R(e)) be a stack of events aggregable with e in Q
if S.IS_EMPTY **then**
S.PUSH(e)
else
e_p←S.POP
if s==u **then**
flag←PCAR_CHECK(e_p,e,N.IN$_{DEADLINE}$,out)
else if t==u **then**
flag←PCAR_CHECK(e_p,e,N.OUT$_{DEADLINE}$,in)
else
flag←true
end if
if flag **then**
MERGE(e_p,e)
end if
end if
return
end function

Algorithm 6 Trackability check for hot processes

function PCAR_CHECK(e_p,e_l,t,flag)
if t>te(e_p) **then**
return false
else if flag==out **then**
return FORWARD_CHECK(e_p,e_l, e_p.dst)
else if flag==in **then**
return BACKWARD_CHECK(e_p,e_l, e_p.src)
return true
end if
end function

3.4 Domain Knowledge Reduction Extension

Besides exploiting causality, our work also utilizes domain knowledge to enhance our data reduction.

3.4.1 Special files

In a Linux system, many system entities are treated as files and interact with processes as file reads/writes. For instance, all device interfaces are regarded as files and can be accessed via `/dev` directory. Shared memory can also be accessed as files under `/shm` directory. All the files in a Linux system can be classified into two categories: plain files and special files. Plain files refer to the files that have data stored in the disk, and special files refer to the files that are only abstractions of system resources.

For a plain file, when a process reads from it or writes to it, an explicit information flow occurs, which will be further used in backtracking or forward-tracking. By contrast, the interactions between processes and special files may involve more complex behaviours and implicit information flows. For instance, if process A writes to a plain file and process B reads from the same file later, there is an explicit information flow from A to B. However, if process A writes to something under `/proc` and process B reads from the `/proc` file later, it is unlikely that there is an information flow from A to B. The reason is that the files under `/proc` are mappings of kernel information, and writing to them or reading from them involves complex kernel activities that cannot be treated as a simple read/write, resulting in no explicit information flow.

Based on such domain knowledge, we further integrate a special files filter into our approach to remove events related to those special files that will not introduce any explicit in-

formation flow. Note that for some special files, such as files under `/shm`, the processes interacting with them can still introduce information flows. Therefore, we will not filter them.

3.4.2 *Temporary files*

In a system, there are many files that only serve temporary purposes and will be deleted afterwards. Many processes need to store temporary information during execution, and hence generate files that only exist for a short period of time. Thus, these temporary files only interact with the process that creates them.

Similar to previous work [31], we define a temporary file as a file that is only touched by one process during its lifetime. Since a temporary file only has information exchange with one process, it does not introduce any explicit information flow in attack forensics either, and therefore we can remove all the events of temporary files from the data.

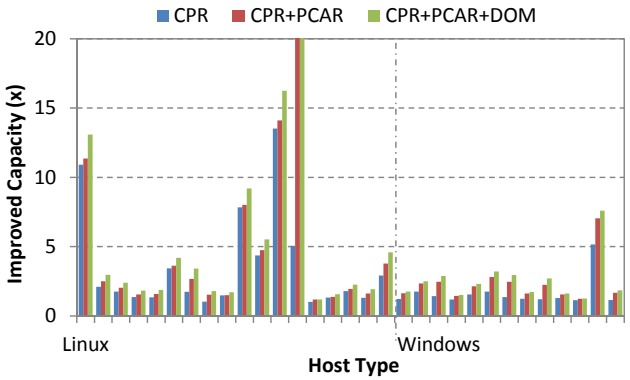

Figure 7: The effectiveness of our system on improving data processing and storage capacities

4. EVALUATION

We implement a prototype of our proposed approach, which consists of CPR, PCAR, and domain knowledge extension (which will be referred to as DOM). Then we conduct a series of experiments based on real-world data collected from a corporate R&D laboratory to demonstrate the effectiveness of our data reduction approach, in terms of data processing, storage capacity improvement and the support for forensic analysis.

We also perform a break-down analysis to show the effectiveness of our approach on different workload patterns, which therefore benefits enterprises with different workloads. Moreover, to fully evaluate our approach, we also implement a naïve event aggregation and conduct experiments for comparison. Finally, we measure the runtime overhead of our data reduction system.

4.1 Data collection

In the corporate research laboratory from which our data is collected, we have deployed an enterprise-wide audit system. There are monitoring agents deployed across servers, desktops and laptops to collect system events. We select one month of data for our study. The data logs we used are collected from more than 30 machines with various server models and operating systems. All the data collected is stored in a daily basis (i.e., the data of each day is stored in a separate database) and there will be a separate dependency graph generated from each individual day's data. There are more than 10 billion events captured during our data collection.

4.2 Data reduction

We first evaluate the overall effectiveness of our approach in data reduction, and then we present a break-down analysis to demonstrate how our solution works under different workloads. Finally, we compare our approach with a naïve aggregation in data reduction.

Our reduction system records how many events are aggregated and then reports the reduction statistics. A reduced data volume has multi-fold significance. Most intuitively, it results in saved storage capacity and reduced storage cost. Moreover, since our reduction system is a module in the data processing stream, the reduced data will save bandwidth and improve the data processing capacity for the following modules. For instance, if our system can reduce data by 80% and the following data processing module needs to perform event queries on our output, our reduction can increase the processing capacity of the query module by 5 times.

4.2.1 *Overall effectiveness*

For comparison reasons, the log data is processed in three phrases. In the first phrase, we only apply CPR. In the second phrase, we further apply PCAR. Finally, we apply domain knowledge reduction extension in the third phrase.

Figure 7 shows the data reduction results of our system (i.e., the improvements on the data processing and storage capacities after applying our data reduction techniques). In total we collected data from 31 hosts, 18 of which have Linux as operating systems and 13 of which have Windows as operating systems. For CPR, on average it can achieve the reduction ratio of 56% (i.e., it can reduce the data size by 56%) thus increasing the data processing and storage capacities by 2.27 times (2.63 in Linux and 1.61 in Windows, respectively). Next, if we apply CPR+PCAR, the overall reduction ratio will be raised to 70%, which achieves 3.33 times growth in the data processing and storage capacities (4 in Linux and 2.44 in Windows, respectively). Finally, after we apply our domain knowledge reduction extension, the reduction ratio can reach 77%, which increases the data processing and storage capacities by 4.35 times (5.26 in Linux and 2.56 in Windows, respectively).

It is evident that our system can effectively reduce data logs and improve data process and storage capacities in a significant fashion. Even with CPR alone, the data size can be reduced by more than half, increasing the data processing/storage capacities by 2 times; on certain hosts, our system can help to increase the data processing and storage capacities by more than 20 times. From Figure 7 we can see that at different hosts, the benefits gained by our system vary. This is because different hosts run different workloads, and the effectiveness of our system is affected by different workloads.

4.2.2 *Break-down analysis*

Different workloads introduce different system activities, resulting in different process behaviours and different amounts of event redundancy. Since the effectiveness of our system is sensitive to different workloads, we need to conduct a break-down analysis to scrutinize how our system works on different workloads.

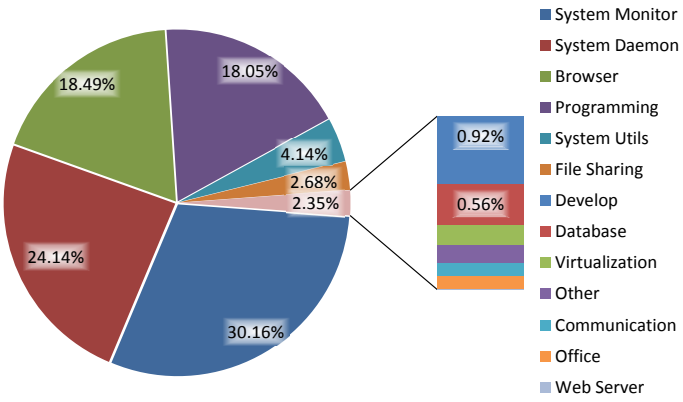

Figure 8: The workload distribution of our collected data

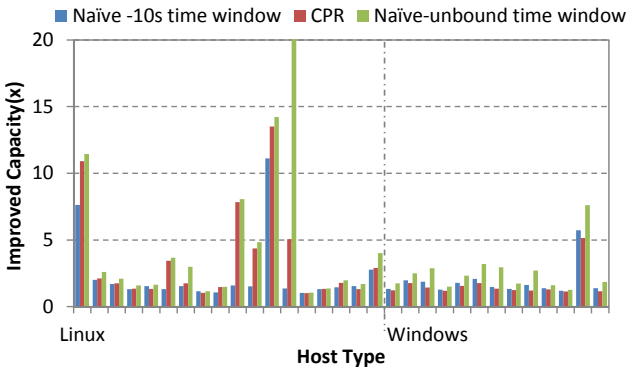

Figure 9: The comparison of naïve aggregation with a 10-seconds time window, CPR, and naïve aggregation with an unlimited time window

We categorize the common workloads in an enterprise environment to the following groups: system monitor, system utility, system daemon, file sharing, browser, office, communication, develop tools, programming, web server, database, and others. System monitoring includes the auditing and monitoring tools. System utility includes common utilities in Linux or Windows such as `mv, cp, ls`. System daemons are daemon processes running in the background, such as `pcscd, sshd` and security scanning in Windows. File sharing represents the applications used to share files such as `SVN, CVS`, and DropBox. Browser includes all types of browsers like Chrome, IE and FireFox. All the applications used for writing documents are categorized to office (e.g., vim, OpenOffice, PowerPoint, excel). Communication includes mailing tools such as GMail, voice tools like Skype and remote access tools like `SSH`. Develop tools are those dedicated to production development such as Matlab, R and Eclipse. Programming refers to all kinds of programming language compilers like `GCC, Java, python`. Web server includes tomcat, Apache and JBoss. Database includes these processes running as database services like MySQL, MangoDB, etc. Finally, the rest of the processes are categorized to others.

Figure 8 illustrates the workload pattern distribution in our collected data. Because this work is conducted in a research institute, the workload pattern is skewed to system monitoring, system utilities, system daemons, and programming. We can see that those processes contribute to the majority of the events we captured.

Table 1 lists the breakdown analysis under different workload patterns, showing the percentages of data reduction and the speed-ups on data processing and storage capacities. From the table we can see that CPR works well on most workloads. However, it works poorly on system daemons, and the reason has been explained previously: the daemon processes generally perform tasks that generate intensively interleaving events. That is also why PCAR works very well on this type of workload. Some system utilities exhibit similar behaviours, and thus applying PCAR can improve their reduction ratios significantly. Office workloads generate considerable temporary files, which is why our domain knowledge reduction extension works best on them. File sharing generates many interactions with temporary files (logs), and thus domain knowledge filtering can help to improve the reduction ratio significantly.

The workload on which our approach is least effective is communication applications. However, their workloads do not contribute much to the entire workload in an enterprise environment. Database, on the other hand, could be one of the majority workloads in certain circumstances, but our approach works less effectively on them. However, our system can still achieve an reduction ratio of more than 40% and increase the data processing and storage capacities by 1.67 times.

4.2.3 *Naïve aggregation*

The naïve approach is based on the heuristics that events appeared in a short period of time between two system entities that tend to share similar behaviours. Thus, in this naïve approach, we blindly aggregate events in a fixed time window, without considering any dependency. Such a naïve approach can be regarded as a state-of-art data reduction approach, which will provide us a baseline of reduction power to compare with.

We implement the naïve aggregation approach in two ways. First, we fix the time window to 10 seconds (i.e., we aggregate the events within a 10-second interval). Second, we set the time window to unlimited (i.e., we will aggregate all aggregatable events in the data). The second naïve aggregation should be an upper bound of data reduction power of any reduction approach that does not remove any entities and has the same event aggregability definition as our approach.

Figure 9 shows the comparisons among the naïve aggregation with a 10-seconds time window, CPR, and the naïve aggregation with an unlimited time window. On average, the naïve aggregation with a 10-seconds time window can improve the data processing/storage capacity by 1.85 times, while CPR can increase the data processing/storage capacity by 2.18. Thus, on average, CPR clearly outperforms the 10-second naïve aggregation; in many cases, its improvement is close to that of the upper bound (i.e., the naïve aggregation with an unlimited time window).

4.3 Support of forensic analysis

Our data reduction approach is designed to support forensic analysis. Therefore, we evaluate the quality of forensic analysis after applying our data reduction (i.e., whether the entities and connectivity can be well preserved in tracking results after we apply our data reduction techniques on the

Table 1: The effectiveness of our data reduction system under different workloads

	CPR				CPR+PCAR				CPR+PCAR+DOM			
	Linux		Windows		Linux		Windows		Linux		Windows	
	Reduc	Impro	Reduc	Impro	Reduc	Impro	Reduc	Impro	Reduc	Impro	Reduc	Impro
System Monitor	55.2%	2.22×	32.1%	1.47×	71.7%	3.53×	62.2%	2.65×	73.5%	3.77×	63.6%	2.75×
System Util	45.3%	1.83×	33.3%	1.50×	77.2%	4.39×	45.8%	1.85×	82.5%	5.71×	57.6%	2.36×
System Daemon	30.6%	1.44×	37.8%	1.61×	82.1%	5.59×	55.4%	2.24×	90.2%	10.20×	58.3%	2.40×
File Sharing	58.0%	2.38×	47.0%	1.89×	71.1%	3.46×	61.9%	2.62×	82.2%	5.62×	81.9%	5.52×
Browser	70.4%	3.38×	41.8%	1.72×	72.6%	3.65×	48.8%	1.95×	74.2%	3.88×	61.1%	2.57×
Office	66.8%	3.01×	32.2%	1.47×	71.2%	3.47×	44.4%	1.80×	82.5%	5.71×	70.7%	3.41×
Communication	22.2%	1.29×	18.2%	1.22×	31.8%	1.47×	30.0%	1.43×	33.5%	1.50×	31.5%	1.46×
Develop	56.5%	2.30×	54.5%	2.20×	75.2%	4.03×	77.4%	4.42×	78.1%	4.57×	78.1%	4.57×
Programming	68.4%	3.16×	38.4%	1.62×	71.5%	3.51×	55.4%	2.24×	82.2%	5.62×	60.1%	2.51×
Web Server	55.1%	2.23×	51.1%	2.04×	72.1%	3.58×	62.6%	2.67×	80.3%	5.08×	70.7%	3.41×
Database	21.2%	1.27×	31.1%	1.45×	33.2%	1.50×	42.5%	1.74×	34.8%	1.53×	47.6%	1.91×
Virtualization	65.2%	2.87×	71.9%	3.56×	66.7%	3.00×	72.8%	3.68×	80.2%	5.05×	73.0%	3.70×
Other	42.1%	1.73×	45.1%	1.82×	48.9%	1.96×	47.0%	1.89×	49.1%	1.96×	47.2%	1.89×

Table 2: Backtracking results before and after reduction

Test Case	Raw	CPR	CPR+PCAR
`distcc_nmap`	84	84	84
`distcc_passwd`	72	72	74
`Freesweep`	38	38	38
`PHPCgi_nmap`	56	56	56
`Gzip`	5	5	5
`netcat_bashrc`	432	432	441
`Passwd`	17	17	17
`Pbzip`	25	25	25
`Useradd`	7	7	7
`Wget_gcc`	19	19	19

auditing data). Since forward-tracking is the opposite of backtracking, in our evaluation we focus on backtracking.

We manually produce some test cases during data collection to generate traces for backtracking. Since we have control over the test environment of these test cases, we are able to figure out the ground truth backtracking results that review the attack/system activity sequences These test cases are listed in column 1 of Table 2. We first perform some multi-step attacks that exploit system vulnerabilities as test cases. The first two cases are the attacks on *Distcc* where the vulnerability (CVE-2004-2687 [3]) allows a remote attacker to execute an arbitrary command, such as `nmap` and `passwd`. The third case is the attack on Freesweep, where a user downloads a malicious package that will create a backdoor to the user's system. The fourth case is the attack on PHPCgi, where the vulnerability (CVE-2012-1823 [4]) allows a remote attacker to inject malicious commands in a query.

The rest of the test cases are a series of normal system operations, such as downloading a program with `wget` and using `gcc` to compile and execute it afterwards. These test cases can also be involved in various multi-step attacks.

For comparison reasons, we perform backtracking on three copies of the data: the raw data (i.e., the data without any reduction), the data only reduced by CPR, and the data reduced by CPR+PCAR.

Table 3: Backtracking results on random POI events

Reduction Method	False Positives	False Connectivities	Additional Entities
CPR	0	0%	0
CPR+PCAR	45	3.7%	1.4%
Naïve-10s	2778	17%	8.8%

Table 2 shows the connectivity change of backtracking on data before and after reduction. Since the entities are untouched anyway, we do not show the statistics of the entities here. The connectivity reflects the quality of the backtracking. Multiple aggregatable events only contribute to one connectivity. As we can see, for these test cases, CPR can perfectly preserve the connectivity and therefore maintains a high-quality tracking results. PCAR is a more aggressive approach, and in two cases it introduces false positive connectivity. A false positive connectivity is caused by the nature that PCAR will enlarge the time window of POI events. False positives will introduce noise nodes and connectivity in the resulted dependency graph from backtracking. The enlarged resulted graph will increase the difficulty of tracing the root cause of anomaly. However, as the false positive rate of our approach is very low, the impact on forensic analysis is negligible.

Since in all cases, both CPR and PCAR do not introduce any missing connectivity in the tracking results, we do not present the statistics here.

To further investigate the impact upon attack forensics and compare with the naïve aggregation, we randomly select 20,000 POI events from the data and apply backtracking. Table 3 illustrates the evaluation results for the naïve aggregation with a 10-seconds time window, CPR and CPR+PCAR. The false positive column indicates how many cases showed that, the backtracking results introduce additional connectivity.

The additional connectivity/entities indicate that for the false positive cases, what is the ratio of the extra connectivity and entities the false positive will introduce. Column 2 is calculated by dividing the number of extra connectivity introduced in all false positive cases with the number of

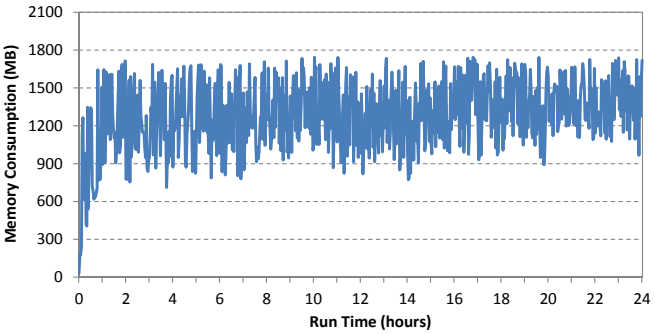

Figure 10: The runtime memory consumption of our system

ground truth connectivity. Ground truth connectivity is one that exists in the tracking results on data without reduction. Similar statistics are calculated for entities and listed in column 3.

From the results we can see, while CPR can preserve the high quality of tracking, PCAR will introduce very few false positives in rare cases (around 0.2%) and the impact is limited. By contrast, the naïve aggregation introduces false positives in more than 10% of the cases, which will introduce considerable noises into the tracking results and therefore significantly degrade the quality of forensic analysis.

4.4 Runtime Performance

While the main purpose of data reduction is to trade data processing and storage capacities with pre-processing overhead, it is important to keep the runtime overhead low. According to our algorithm design, both CPR and PCAR are linear in time complexity. We only need to scan all events once and update the states with hash tables. We also profile the runtime memory and CPU consumption of our approach. Since the data is stored/processed on a daily basis, our profiling lasts one day as well. Figure 10 illustrates the memory consumption of our complete system (CPR+PCAR+domain knowledge extension) over time. The data rate is around 4,000 events per second. As we can see from the figure, the memory consumption fluctuates. The reason is two-fold. On one hand, our system continues storing new events and updating new states; on the other hand, it continues to output the events that cannot be aggregated and drops events that are either aggregated or discarded. Overall, the memory consumption remains under 2GB, which is a small overhead for any commercial server. As for the CPU consumption, a single 2.5G Hz core can easily handle such a data rate. Thus, the CPU overhead is also minor.

5. DISCUSSION

5.1 Attack Resilience

Serving as the underlying support for forensic analysis, it is important that our technique is resilient to attacks, such as data evasions.

Both CPR and PCAR are resilient against evasion attacks. Since CPR guarantees that no causal dependency is lost after data reduction, an attacker simply cannot exploit CPR to distort the reality by any means. Although PCAR does result in dependency loss during data reduction, it is very difficult to exploit such a loss to cover any meaningful malicious activities. Because our system monitors events at fairly low-level, even simple activities consist of events chained in multiple dependency hops (e.g., a single ssh login consists of 4 hops of dependencies). PCAR guarantees that any dependency loss is confined within a single hop from a hot process. Therefore, although attackers could intentionally trigger PCAR by injecting multiple hot processes during the attack, each hot process is usually well-separated from others in term of low-level dependencies, and thus PCAR will unlikely lead to confusion in tracking the attack steps. In addition, it is also undesirable for an attacker to generate multiple hot processes, since doing so significantly enlarges its attack footprint, which defeats the intention of evading detections.

5.2 Generality

Our approach is designed to work with generic system level event traces, and reduces data volume by removing redundant dependencies within the data. We make little assumption on the specifics of data, such as system platforms, instrumentation techniques, and semantic levels, and therefore our approach is applicable to a large variety of workloads and data sources.

We demonstrated the capability of our technique using data collected from an enterprise security monitoring system which employs no custom instrumentation. The data covers Windows and Linux operating systems, and are sourced from off-the-shelf kernel auditing and event tracing. Moreover, our technique is complementary to other existing data reduction techniques. CPR and PCAR do not alter data characteristics with respect to platform, instrumentations and high-level semantics, and thus can function as an independent data reduction layer, applied either before or after other data reductions.

6. RELATED WORK

Very limited research directly addresses the issue of the reduction of big-data of attack provenance derived from system-level audit events. LogGC [31] is the closest work, and it has three major ideas. (*i*) Some system objects such as temporary files are isolated, have a short life-span, and have little impact on the dependency analysis, so that such objects and events can be garbage collected to save space. (*ii*) Application-specific adaptation by using the existing application logs. (*iii*) Application source modification to output finer-grained dependency information. Ideas (*ii*) and (*iii*) are application specific adaptation, need human in the loop to understand and change approach for the specific applications. The (*i*) and our approach are toward a general scheme. In our study with an enterprise of a few hundred hosts, we found a large number of diverse applications, so it is not easy to adopt an application-specific approach. For general approaches, we found that our approach offers comparable reduction as LogGC. Furthermore, the two approaches are orthogonal. Their approach focuses on object life-span, while we focus on event causality equivalence. The two approaches compliment one another and can be further combined.

ProTracer [34] proposes a new system that aims at reducing log volume for provenance tracing. While their approach is to build an audit system that is dedicated to provenance tracing, our approach can be applied on existing audit systems without any modification and our reduced data also retains the potential to be used by applications other than forensic analysis.

The other data reduction works do not consider preserving the dependency path and thus are not applicable to our problem. Some works on data aggregation aim to reduce communication cost and improving data privacy [17, 19, 20, 21, 26, 24, 38]. In contrast to these works, we focus on reducing system auditing data while supporting forensic analysis, particularly backtracking and forward-tracking. There are also other studies exploiting graph techniques for data reduction [18, 37]. However, none of them leverages dependency in data reduction.

Since the assumptions and usage scenarios in our approach are very different from these previous data reduction, it is hard to compare them side-by-side. Our work has its unique contribution in the novelty of the key techniques and is complementary to other reduction techniques.

Forensic analysis plays an important role in security. King and Chen [27] proposed a backtracking technique to perform intrusion analysis by automatically reconstructing a chain of events in a dependency graph. Following this approach, various forensic analysis techniques have been developed in various scenarios such as recovering an intrusion [23], worm break-in detection [25], forensic analysis in virtualized environments [29], binary level attack provenance [30], risk analysis in networks [35], and file system intrusion detection [36]. Ma et al. [33] proposed a Windows audit logging technique that can provide accurate traces for forensic analysis

Although there are different tracking techniques, the key concept of exploiting the dependency between system events for analysis remains the same. Dependency graphs have been widely used in system and security studies. Besides the usage of a dependency graph in forensic analysis [27, 23], researchers have leveraged dependency graphs to perform code generation on multiprocessor systems [15], identify user clicks in http requests [32], predict system failures [41], diagnose networking failures [40], assess attacks in enterprises [22], perform malware classification [39], and detect security failures in the cloud environment [16].

7. CONCLUSION

We presented a novel approach that exploits dependency among system events to reduce the size of data without compromising the accuracy of the forensic analysis. Core to our contribution, we proposed the concept of trackability, which determines causality relation among system events. By aggregating events under the same trackability, our approach could reduce a large potion of data while preserving events relevant to a forensic analysis. Incorporated with domain knowledges specific to system process behaviours, our prototype implemented data reduction for two types of forensic analyses: backtracking and forward-tracking.

Evaluated over datasets gathered from a real-world enterprise environment, our results show that our approach improves space capacity by 3.4 times with no or little accuracy compromise. Although the overall space and computational requirements for large-scale data analysis remain challenging, we hope our data reduction approach will bring forensic analysis in a big data context closer to become practical.

8. ACKNOWLEDGEMENT

We would like to thank our shepherd Tudor Dumitras and the anonymous reviewers for their insightful and detailed comments. Zhang Xu and Haining Wang were partially supported by ONR grant N00014-13-1-0088 and NSF grant CNS-1618117.

9. REFERENCES

[1] Anthem cyber attack. http://abcnews.go.com/Business/anthem-cyber-attack-things-happen-personal-information/story?id=28747729.

[2] Case study: The Home Depot data breach. https://www.sans.org/reading-room/whitepapers/casestudies/case-study-home-depot-data-breach-36367.

[3] CVE-2004-2687. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2687.

[4] CVE-2012-1823. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1823.

[5] Cyber kill chain. http://www.lockheedmartin.com/us/what-we-do/information-technology/cybersecurity/tradecraft/cyber-kill-chain.html.

[6] Ebay inc. to ask Ebay users to change passwords. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

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

[8] The Linux audit framework. https://www.suse.com/documentation/sled10/audit_sp1/data/book_sle_audit.html.

[9] OPM government data breach impacted 21.5 million. http://www.cnn.com/2015/07/09/politics/office-of-personnel-management-data-breach-20-million/.

[10] Sony reports 24.5 million more accounts hacked. http://www.darkreading.com/attacks-and-breaches/sony-reports-245-million-more-accounts-hacked/d/d-id/1097499?

[11] Stuxnet. https://en.wikipedia.org/wiki/Stuxnet.

[12] Target hit by credit-card breach. http://online.wsj.com/news/articles/SB10001424052702304773104579266743230242538.

[13] Transparent computing. http://www.darpa.mil/program/transparent-computing.

[14] Trustwave Global Security Report, 2015. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf.

[15] J. A. Ambrose, J. Peddersen, S. Parameswaran, A. Labios, and Y. Yachide. Sdg2kpn: System dependency graph to function-level kpn generation of legacy code for mpsocs. In *Proceedings of IEEE ASP-DAC'14*, pages 267–273.

[16] S. Bleikertz, C. Vogel, and T. Groß. Cloud radar: near real-time detection of security failures in dynamic virtualized infrastructures. In *Proceedings of ACM ACSAC'14*, pages 26–35.

[17] C. Castelluccia, E. Mykletun, and G. Tsudik. Efficient aggregation of encrypted data in wireless sensor networks. In *Proceedings of IEEE MobiQuitous'05*, pages 109–117.

[18] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proceedings of the ACM SIGMOD'08*, pages 993–1006.

[19] S. Cheng and J. Li. Sampling based (epsilon, delta)-approximate aggregation algorithm in sensor

networks. In *Proceedings of IEEE ICDCS'09*, pages 273–280.

[20] S. Cheng, J. Li, Q. Ren, and L. Yu. Bernoulli sampling based (ε, δ)-approximate aggregation in large-scale sensor networks. In *Proceedings of IEEE INFOCOM'10*, pages 1181–1189.

[21] G. Cormode and K. Yi. Tracking distributed aggregates over time-based sliding windows. In *Proceedings of SSDBM'12*, pages 416–430.

[22] N. Ghosh, I. Chokshi, M. Sarkar, S. K. Ghosh, A. K. Kaushik, and S. K. Das. Netsecuritas: An integrated attack graph-based security assessment tool for enterprise networks. In *Proceedings of ACM ICDCN'15*.

[23] A. Goel, K. Po, K. Farhadi, Z. Li, and E. De Lara. The taser intrusion recovery system. In *Proceedings of ACM SOSP'05*, pages 163–176.

[24] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top-k interesting subgraph discovery in information networks. In *Proceedings of IEEE ICDE'14*, pages 820–831.

[25] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang. Provenance-aware tracing ofworm break-in and contaminations: A process coloring approach. In *Proceedings of IEEE ICDCS'06*.

[26] J. Jose and S. Manoj Kumar. Energy efficient recoverable concealed data aggregation in wireless sensor networks. In *Proceedings of IEEE ICE-CCN'13*, pages 322–329.

[27] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of ACM SOSP'03*.

[28] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[29] S. Krishnan, K. Z. Snow, and F. Monrose. Trail of bytes: efficient support for forensic analysis. In *Proceedings of ACM CCS'10*, pages 50–60.

[30] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of NDSS'13*.

[31] K. H. Lee, X. Zhang, and D. Xu. Loggc: garbage collecting audit log. In *Proceedings of ACM CCS'13*, pages 1005–1016.

[32] J. Liu, C. Fang, and N. Ansari. Identifying user clicks based on dependency graph. In *Proceedings of IEEE WOCC'14*, pages 1–5.

[33] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of ACM ACSAC'15*.

[34] S. Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of NDSS'16*.

[35] M. Rezvani, A. Ignjatovic, E. Bertino, and S. Jha. Provenance-aware security risk analysis for hosts and network flows. In *Proceedings of IEEE NOMS'14*, pages 1–8.

[36] S. Sitaraman and S. Venkatesan. Forensic analysis of file system intrusions using improved backtracking. In *Proceedings of IEEE IWIA'05*, pages 154–163.

[37] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long. A hybrid approach for efficient provenance storage. In *Proceedings of ACM CIKM'12*, pages 1752–1756.

[38] X. Xu, R. Ansari, A. Khokhar, and A. V. Vasilakos. Hierarchical data aggregation using compressive sensing (hdacs) in wsns. *ACM Transactions on Sensor Networks*, 11(3):45, 2015.

[39] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of ACM SIGSAC'14*, pages 1105–1116.

[40] M. Zibaeenejad and J. Thistle. Dependency graph: an algorithm for analysis of generalized parameterized networks. In *Proceedings of IEEE ACC'15*, pages 696–702.

[41] T. Zimmermann and N. Nagappan. Predicting subsystem failures using dependency graph complexities. In *Proceedings of ISSRE'07*, pages 227–236.