APTrace: A Responsive System for Agile Enterprise Level Causality Analysis

Jiaping Gui*‖, Ding Li*‖, Zhengzhang Chen*, Junghwan Rhee*, Xusheng Xiao†, Mu Zhang‡,
Kangkook Jee§, Zhichun Li¶ and Haifeng Chen*
*NEC Labs America Inc.
‖Corresponding authors
†Case Western Reserve University
‡University of Utah
§Unversity of Texas at Dallas
¶Stellar Cyber

***Abstract*—While backtracking analysis has been successful in assisting the investigation of complex security attacks, it faces a critical dependency explosion problem. To address this problem, security analysts currently need to tune backtracking analysis manually with different case-specific heuristics. However, existing systems fail to fulfill two important system requirements to achieve effective backtracking analysis. First, there need flexible abstractions to express various types of heuristics. Second, the system needs to be responsive in providing updates so that the progress of backtracking analysis can be frequently inspected, which typically involves multiple rounds of manual tuning. In this paper, we propose a novel system, *APTrace*, to meet both of the above requirements. As we demonstrate in the evaluation, security analysts can effectively express heuristics to reduce more than 99.5% of irrelevant events in the backtracking analysis of real-world attack cases. To improve the responsiveness of backtracking analysis, we present a novel execution-window partitioning algorithm that significantly reduces the waiting time between two consecutive updates (especially, 57 times reduction for the top 1% waiting time).**

***Index Terms*—Backtracking analysis, domain language, expressiveness, responsiveness**

I. Introduction

Modern enterprises are facing complex Advanced Persistent Threat (APT) attacks. It has been widely reported that APT attacks cause severe financial loss to today's enterprises [1], [2], [3], [4], [5]. Once attackers have successfully penetrated the network of an enterprise, they may first perform a series of complex but normal behaviors, and then hide for a long period of time before launching the actual attack. Traditional security solutions, such as firewall or anti-virus software, that only detect malicious behaviors are insufficient to defend against these APT attacks. The reason is that these solutions fail to locate the starting point (root cause) of attacks in the enterprise network.

To counter these attacks, backtracking analysis [6] is introduced to recover their attack scenarios. Specifically, backtracking analysis searches the system activity log, which is generated by security auditing frameworks such as ETW [7] or Linux Audit Framework [8], and produces a dependency graph that connects system events. This has been proven to be successful in helping security analysts to analyze complex security attacks that involve multiple software vulnerabilities in enterprise environments [6], [9], [10], [11], [12], [13], [14], [15].

Despite the effectiveness of backtracking analysis, it suffers from a critical problem that limits its usability and application, the dependency explosion problem [11]. A backtracking analysis may explore many irrelevant events, which can cause backtracking analysis to take hours to generate a dependency graph that is too large to interpret effectively. To apply backtracking analysis, security analysts still need to manually tune or "debug" the analysis with case-specific heuristics. However, existing systems fail to meet two important system requirements that facilitate the tuning process.

The first system requirement is to have flexible and unified abstractions for different types of heuristics. In existing systems, all the heuristics are tightly coupled with the implementation [6], [9], [12], [16]. As a result, security analysts cannot easily apply heuristics that do not belong to the specific system implementation to the tuning of backtracking analysis. In the literature, there has been no treatment on exposing such a common interface that enables security analysts to leverage various heuristics. The second system requirement is to enable backtracking analysis to provide updates responsively so that the frequent inspection of progress is feasible. This requirement is critical to the application of backtracking analysis since certain heuristics such as excluding searching for specific processes or files must be applied in time to prevent waste of resources in searching irrelevant events. In this paper, we define *responsive backtracking analysis* as displaying the already explored part of the dependency graph, while the system is still searching for more system logs to update the graph. Unfortunately, existing backtracking analysis systems do not meet this responsive requirement, as the backtracking analysis results are not returned until the execution is completed. If there is a massive amount of log data, the security analysts could wait for a very long time before he gets the results of backtracking analysis.

Fulfilling two requirements above involves several significant scientific challenges. First, we need to find out the appropriate abstractions to express different heuristics. To

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00151

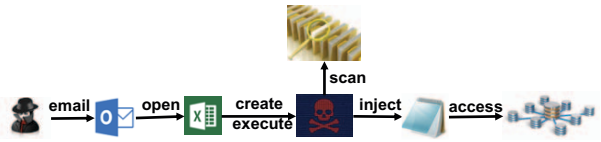

Fig. 1: The example attack

do it, the abstractions need to be flexible to support all common heuristics, yet they still need to be concise and have a reasonable learning curve. Second, we need to ensure the responsiveness of backtracking analysis. Since the dependencies are not evenly distributed, responsive backtracking analysis can be blocked for a long time without any updates or returned very quickly. The security analysts expect a steady update interval so that they may better plan their tasks and provide heuristics in time. But how to guide backtracking analysis to maintain a steady update interval is an open problem.

In this paper, we propose Attack Provenance Tracer (*APTrace*), a backtracking analysis system that meets the above two requirements. *APTrace* first provides Backtracking Descriptive Language (BDL), a domain specific language to express the common heuristics that appear in the literature. To meet the second requirement, *APTrace* uses a novel execution-window partitioning algorithm, which takes advantage of the temporal locality of system events, to reduce the waiting time between two consecutive updates, i.e., maintaining a steady update interval. To our knowledge, we are the first to provide the system with flexible abstractions for common heuristics and responsive execution of backtracking analysis.

The contributions of this paper can be summarized as follows:

- We propose a domain specific language that enables the incorporation of various heuristics and domain knowledge to backtracking analysis.
- We design a new execution-window partitioning algorithm to accelerate backtracking analysis.
- We have implemented *APTrace* and deployed it into a real-world enterprise computer environment. We perform extensive evaluation using a real-world security dataset that is orders of magnitude larger than the ones used in previous work [11], [16]. Our experimental results are promising, demonstrating that *APTrace* can significantly improve the responsiveness of backtracking analysis.

The remainder of this paper is organized as follows. In Section II, we discuss a real-world APT attack scenario and the challenges when we apply backtracking analysis to get its root cause. In Section III, we introduce the system, *APTrace*. In Section IV, we present the evaluation results. Finally, we discuss related work in Section V, and conclude this paper in Section VI.

II. Motivation

In this section, we use a real-world APT attack scenario to discuss the usage of backtracking analysis and the problems it faces in practice. Before delving into the attack example, we first formulate the terminologies below to be used in the rest of this paper.

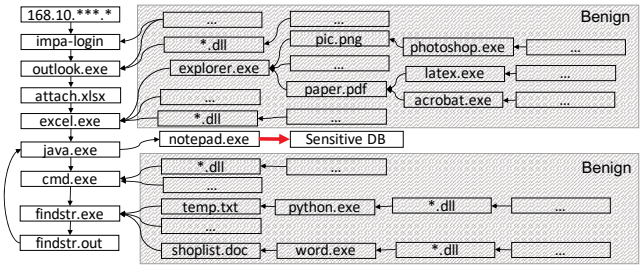

Fig. 2: Dependency graph of the motivating attack

- System object: a file, process instance, and network socket in operating systems.
- System event: an interaction between two system objects, such as a process forking another process, a process reading or writing a file, and a process talking to a network channel. A system event contains four attributes: the subject, which is the process instance that initiates the interaction; the object, which is the system object that the subject interacts with; the direction of the data flow (from the subject to the object or vice versa); and the timestamp of interaction.
- Backward event dependency: an event, B, backward depends on another event, A, if they meet two conditions. First, A happens before B. Second, the destination of A's data flow is the source of B's data flow.
- Tracking graph: a directed graph that connects all system events with backward dependencies among them. The nodes are system objects, the edges system events, and the directions of edges the directions of data flow.
- Backtracking analysis: the analysis that tracks the backward dependencies among system events. It takes an event as the starting point and generates a tracking graph.

Our example is a phishing email attack that uses the Windows Privilege Escalation vulnerability [17]. The attack process is shown in Figure 1. It has several steps. First, the attacker sends an email with a malicious Excel file to the victim. When the victim receives and opens the Excel file, a malware named `java.exe` is created and executed. `java.exe` first uses `cmd.exe` to run `findstr.exe` in the home directory to search important credentials on the victim's machine. In this process, `java.exe` may hibernate itself multiple times to avoid scanning too many files in a short period of time, which otherwise may trigger the alert from system anomaly detectors. Such a scanning can take days or even months. Once `java.exe` gets the credential, it injects the malicious code to the memory of `notepad.exe` [18], [19], takes the advantage of Windows Privilege Escalation vulnerability to get the administrator privilege through `notepad.exe`. Lastly, `java.exe` leverages `notepad.exe` to connect the internal database and dumps the sensitive data.

Traditional defence methods, such as virus scanners or system anomaly detectors, are insufficient to detect this type of attack, since these methods only detect the malicious behavior, such as dumping the sensitive data or executing a malware, but fail to uncover its root cause. Hence as long as the security

loophole exists, which is the phishing email, future attacks will keep coming in.

Backtracking analysis is thus proposed to address above APT attacks. The input of backtracking analysis is a system anomaly alert, the output is a dependency graph. By viewing the dependency graph, security analysts can discover the penetration point that leads to the anomaly alert. For instance, the dependency graph corresponding to the example attack in Figure 1 is shown in Figure 2. The part of Figure 2 that is not covered by the grey areas shows the critical steps of the attack. Each node represents a system object, such as a process, a file, or a network connection. An edge is the causality relationship, or dependency, between two system objects. The red bold arrow in the graph is the edge that represents the anomaly alert. By viewing this dependency graph, security analysts can discover that the anomaly comes from a phishing email.

The process of building the dependency graph (Figure 2) is as follows: when backtracking analysis receives the anomaly alert as input (red bold arrow), it searches the system log history and finds the event that `notepad.exe` is executed by `java.exe`. Then, backtracking analysis adds `java.exe` to the dependency graph and continues searching for the causalities of `java.exe`. This process repeats until security analysts find the penetration point, which may expose the exploited security loopholes, or there is no new nodes that could be explored and added.

a) The Dependency Explosion Problem: The key challenge for backtracking analysis is the dependency explosion problem [11]. That is, backtracking analysis can explore a lot of relevant yet benign system events and add these events to the dependency graph. Analyzing these events does not expose the penetration point but only consumes a lot of time and generates a dependency graph that is too big to view and investigate. For example, in Figure 2, the nodes in grey areas, although relevant, do not lead to the penetration point. Having these benign nodes are problematic. In our case, it takes more than *four* hours to generate the dependency graph that contains about 30.75K nodes! It is very difficult for people to effectively recover the security loopholes with such a big graph.

b) Current Approach In Practice: There are a few automated techniques to reduce the benign events [11], [13], [15]. However, they cannot fully address the dependency explosion problem. In practice, heuristic based techniques are often applied [16], [6], [9], [14]. In these approaches, people apply different heuristics based on their domain knowledge to prune the benign events. For example, security analysts may exclude from backtracking analysis `*.dll` files and the Windows File Explorer (`explorer.exe`) since they often introduce a lot of benign events. The problem of above techniques is that the heuristics are case specific and cannot be automatically applied. Otherwise, attackers can craft sophisticated attacks to bypass backtracking analysis. For example, the attack can inject into the memory of `explorer.exe` to execute malicious code. Simply pruning `explorer.exe` in backtracking analysis regardless of the specific cases can lead to severe security issues.

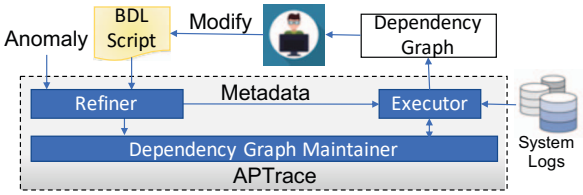

Fig. 3: Overview of *APTrace*

c) Realistic Workflow of backtracking analysis: In practice, it is a repeated process of trying and confirming different heuristics when security analysts use backtracking analysis. That is, they may first analyze a small dependency graph (e.g. limit the execution time of backtracking analysis for a short time), then interpret the small dependency graph and estimate the heuristics that may be effective based on their expertise, and lastly apply the heuristics to start a new iteration of backtracking analysis. For example, in our attack case above, when security analysts terminate backtracking analysis in a few minutes, they can get a small dependency graph that contains a few `*.dll` files. By investigating this graph, security analysts realize that these `*.dll` files are library related and may be benign. In this case, they first confirm there is no suspicious modifications to the `*.dll` files, then they apply the heuristics to exclude the `*.dll` files. Such a process may involve several iterations until security analysts find the penetration point.

d) Problems of Current Workflow: Backtracking analysis in current systems has to finish the execution before being completed (so-called *execute-to-complete*). To try and verify different heuristics, security analysts have to terminate the execution early or set a time limit. However, setting an appropriate time limit is non-empirical, since the distribution of system dependencies is imbalanced. For a certain node in the dependency graph, its dependents are often concentrated in certain ranges of time in the history. While backtracking analysis is searching for the dependents of the node among the whole history log, it cannot query the dependents and return them with a constant and predictable speed. Due to this reason, when security analysts stop running backtracking analysis, they often get either a too-small dependency graph that contains very little information to discover effective heuristics, or a too-large dependency graph that already contains many benign events.

A more appropriate way is to make the execution responsive. That is, when there is an update to the dependency graph, the system immediately reports it to security analysts. In this process, security analysts keep monitoring the progress of backtracking analysis. When there is enough information to estimate an effective heuristic, they can stop the execution immediately. By doing so, security analysts can avoid having a too-small or too-large dependency graph.

III. Design

To achieve agile causality analysis of attacks on enterprise security big data, *APTrace* leverages an interactive process that incorporates the heuristics from security analysts in an

efficient and effective manner. In particular, *APTrace* meets two requirements: (1) it allows security analysts to express different heuristics in a unified framework through backtracking descriptive language. (2) it provides responsive backtracking analysis through the execution-window partitioning techniques.

The workflow of *APTrace* is shown in Figure 3. It has three inputs. The first input is the security anomaly alert that security analysts want to explore. The second input is the system log history that records system activities, each of which is represented by a dependency relationship. The third one is the Backtracking Descriptive Language (BDL) script that describes the heuristics from security analysts.

In the first step, the *Refiner* accepts the BDL and the system anomaly. It processes the BDL and generates the metadata, which is the executable instructions and parameters to run backtracking analysis. This metadata is then sent to *Executor*.

When the *Executor* gets the metadata from the *Refiner*, it executes the backtracking analysis progressively. It searches the system logs with the execution-window partitioning algorithm and progressively updates the dependency graph. During this time, it relies on the *Dependency Graph Maintainer* to realize the certain types of heuristics. We will discuss the usage of *Dependency Graph Maintainer* in more detail in Section III-B2.

Finally, during the execution, security analysts can monitor the dependency graph incrementally. When they come up with a new heuristic based on the current dependency graph, they can pause the execution of *APTrace*, modify the BDL script, and then resume the execution. Once *APTrace* gets the new BDL script, the *Refiner* will check if the new version of BDL script is compatible to the old version and try to reuse the result of the previous execution. Lastly, the *Executor* will run backtracking analysis with the new heuristics included in the BDL script.

A. Backtracking Descriptive Language

BDL is a domain specific language that allows users to specify conditions in the backtracking analysis. We design BDL to allow users to specify the constraints of backtracking analysis in a concise way. BDL supports users to specify the time range, host range, the starting point of backtracking, the termination conditions of backtracking, and the path to explore. These are the major heuristics that are frequently used by security analysts. During the execution of a BDL script, users will get a progressively updated backtracking graph as the output. An example of BDL script is shown in Program 1. This example tracks the path of two malicious applications that steal a sensitive file and send it to the network.

A BDL script contains three main parts. The first part, the *general constraint* (Lines 1-2 in Program 1). In Program 1, Lines 1-2 indicate that the tracking analysis only tracks the system events in the "desktop1" and "desktop2" between the dates "04/02/2019" and "05/01/2019". In BDL, the general constraints are optional. If they are not specified, BDL will search all the hosts in the default time range.

```
1 from "04/02/2019" to "05/01/2019"
2 in "desktop1", "destop2"
3 backward file f[path = "C://Sensitive/important.doc"
      and event_time = "04/16/2019:06:15:14" and
     type = "write" ]
4 -> proc p[exename = "malware1" or exename = "
     malware2" and event_id = 12] // added in v2
5 -> ip i[dstip = "168.120.11.118"]
6 where time < 10mins and hop < 25
7 and proc.exename != "explorer" // added in v3
8 output = "./result.dot"
```

Program 1: Example of TDL script

We have the time range and host range as the general constraints because, in practice, the system logs have properties with temporal and spatial locality. When the security team receive an alert, in many cases, they will first study the related hosts in a recent time range. To do it, we provide the general constraints that support a concise representation of the time and host range.

The second part, *tracking declaration* (Lines 3-7 in Program 1), specifies what events should be analyzed and when the tracking analysis should be terminated. The tracking declaration has two parts. The tracking statement, which starts with "backward", specifies the critical points, such as the starting point, the end point, and some critical intermediate points. The tracking statement supports the intermediate points since, in many cases, the security team need to specify the backtracking to focus on the paths that meet certain patterns. During the execution, *APTrace* will automatically explore the paths that go through the intermediate points before other paths. Once the backtracking is done, *APTrace* removes the paths that do not meet the constraints of the intermediate points from the final result. The *where* statement, which starts with "where", specifies the more general constraints on events, such as excluding the events that meet certain conditions or limit the time of backtracking. This statement allows users to specify the general constraints in a more concise way.

Lines 3 of Program 1 indicates that the backtracking analysis starts at the event that writes to the file `C://Sensitive-/important.doc` at "04/16/2019:06:15:14". This event is defined as the starting point. Line 5 means the backtracking analysis ends at the networking communication with the IP address "168.120.11.118", which is defined as the end point. The user can also use "*" as the end point if he has no constraints for it. Line 4 means that paths from the starting point to the end point should go through the process with the name as "malware1" or "malware2" and have the ID as 12.

Lines 6-7 of Program 1 indicate that both of the tracking analyses should exclude the processes with the executable name as "explorer". It also indicates that the whole process should not be terminated until the execution exceeds 10 minutes or the diameter of tracking graph is greater than or equal to 25.

The third part of BDL is *output specification*, which is shown in Line 8 of Program 1. It specifies that the generated tracking graph should be stored to the path `./result.dot`.

Program 1 also reflects the interactive process in Figure 3. It contains three versions. The first version, v1, does not have

Line 4 or Line 7 in Program 1. This version cannot find any interesting result within the time limit. Through the interaction with *APTrace*, users are able to learn more information. As such, they create the second version, v2, which has Line 4 to accelerate the tracking analysis. Similarly, in the third version, v3, users add Line 7 to remove the Windows File Explorer from the tracking analysis. With the third version, users can find the suspicious IP within the time limit.

1) Tracking Declaration: The tracking statement can be declared as follows:

```
backward (type var[condition_list] ) (-> type var[
    condition_list] )+
```

In the tracking statement, the keyword "backward" indicates the start of the tracking statement. It is followed by a list of nodes. A node is a filter of events, and can be declared as "type var[condition_list]". "type" declares the type of a system object. It has three values: "proc" for processes events, "file" for file events, and "ip" for network connection events. "var" is a user-defined variable name. "condition_list" is a list of constraints that filter the system events. The constraints are connected by logical operations. A constraint in the "condition_list" is a binary operation statement in the form of "field op value", where "field" is an attribute name of the variable. There are two types of options for the "field". The first one is the shared options. These options are "subject_name", "subject_pid", "action_type", "event_id", "event_time". They can be used in all "file" , "proc" and "ip". The second type is the object specific options. For "file", the possible options for "field" are "filename", "host", "path", "last_modification_time", "last_access_time", and "creation_time". For "proc", the possible options are "host", "exename", "pid", and "starttime". For "ip", the possible options are "src_ip", "dst_ip", and "start_time". "op" is a binary operation whose possible choices are "<", "<=", ">", ">=" "=", and "! =". "value" after "op" could be a string, a numeric value, or a time string. If "value" is a string, then "=" and "!=" will be interpreted as a regular-expression match and not match, respectively.

Assume that there are k nodes, the list of which has the format as $n_1 \rightarrow n_2 ... \rightarrow n_k$. In this list, n_1 is the starting point that is required, while n_k is the end point. The user can use a "*" as the end point to specify that there is no specific constraints about the end point. n_2 to n_{k-1} are the intermediate points. They are optional. The list means that the backtracking should find the paths from the starting point to the end point, which go through n_2, $n_3 ... n_{k-1}$ sequentially.

The *where* statement, which is optional, defines the constraints that are not associated with any specific system objects. These constraints will be used to filter system objects during the tracking analysis. For any system object that does not meet the constraints in the *where* statement, it will be deleted from the tracking analysis without further exploration. A *where* statement can be declared as follows:

```
where (type.field | hop | time) op value
```

In the *where* statement, users can specify a list of constraints in the form of "type.field op value". The constraints are also connected by logic operations. "type", "field", and "op" have the same value set as in the tracking statement. Besides "type.field", the *where* statement also accepts two special fields: "time" and "hop". They are used to terminate the tracking analysis and can only be used with the "<=" operation. "time" is used to limit the time of the tracking analysis. "hop" is used to limit the maximum length of paths in the tracking analysis. When the tracking analysis finds a path that has the length longer than the threshold specified by "hop", it stops exploring the path and switches to other shorter paths if any.

B. APTrace Component Design

In this section, we describe our design for various components of *APTrace* and how they collectively fulfill the system requirements and necessary functionalities of BDL.

Algorithm 1: Responsive Tracking

Input: e_0: starting point event
Output: G: dependency graph

```
1 Initialize priQueue ← genExeWindow(e_0) and G ← e_0;
2 while priQueue is not empty do
3     curr ← priQueue.poll();
4     G ← addInComingEdges(curr);
5     for Event e in curr.getEdges() do
6         priQueue ← genExeWindow(e);
7     end
8 end
9 return G;
```

1) Executor: The *Executor* is the core component of *APTrace*. It accepts the metadata from the *Refiner* and performs responsive backtracking analysis. The key challenge for *Executor* is to ensure the responsiveness. Dependencies of a system event are not evenly distributed in the log history. The naïve approach that directly searches the whole log history can cause the backtracking analysis to be blocked for a long time (hours or even days) without any updates. To address this challenge, the *Executor* uses an execution-window partitioning algorithm for responsive backtracking analysis.

Execution-Window Partitioning: There are two insights that motivate the execution-window partitioning algorithm. First, a system event may depend on many other events (dependents). Retrieving all the dependents together in one update can take a long time. Instead, retrieving the dependents in many smaller batches can reduce the waiting time between two updates. Second, the system events have temporal locality. A system event tends to depend on another event that is temporally close. This means that the *Executor* is more likely to find the direct and indirect dependents of an event in the time period that is temporally close to the current event.

Based on the above observation, we design the execution-window partitioning Algorithm as shown in Algorithm 1. This is a prioritized graph searching process on system events. In this process, instead of pushing all the dependents of current event to the queue, our approach adds the execution windows that contain the dependents of the current event (Line 1 and

Line 6). Formally, an execution window is defined as a 3-tuple $\langle begin, finish, e\rangle$, where $begin$ is the starting time point, $finish$ the end time point, and e the event that needs to be explored. The events will be retrieved from the database in the unit of an execution window.

In the "while" loop of graph searching (Lines 2-6), our approach first pulls an execution window from the queue, find all the events in the current execution window that have dependencies to the generated part of the dependency graph, and add these events to the final dependency graph. The events are used as edges in the dependency graph (Line 4). Then, in the "for" loop between line 5 and line 7, our approach enumerates all the events that occur in the current execution window from the database (Line 5), obtains their execution windows (Line 6), and adds these execution windows to the queue for future exploration (Line 6).

In our approach, the function $genExeWindow$ is used to get the execution windows of an event. It accepts an event, e, as input and returns all the execution windows of the event. The procedure of $genExeWindow$ is as follows: first, $genExeWindow$ gets the timestamp te of the input event. Then, $genExeWindow$ generates a monolithic execution window as $\langle ts, te, e\rangle$, where ts is a pre-defined global starting time. Third, $genExeWindow$ cuts the monolithic execution window into k pieces, where k is a user configurable parameter. The cutting starts from te to ts. The length of each execution window is a geometric sequence with common ratio 2. Assume the length of the first execution window is $\sigma = \frac{te-ts}{2^k-1}$. The length of the second execution window is 2σ, the length of the third window is 4σ, and so on. We have a different length for each execution window because the windows in the front are temporally closer to the current events. So, the density of events in these windows are potentially higher. Having a smaller size can reduce the total number of dependencies and reduce the waiting time between updates.

In Algorithm 1, the generated execution windows are added to a priority queue, $priQueue$, at Line 1 and Line 6. This priority queue prioritizes the execution windows based on their end time. In particular, the execution window that is temporally closer to the time of the starting point will be prioritized in the queue and placed before the ones with a farther end time. We do this for the reason as we described in the beginning. This could improve the responsiveness of backtracking.

2) The Dependency Graph Maintainer: The main purpose of having the *Dependency Graph Maintainer* is to realize the prioritization on search directions. Ideally, we would like backtracking analysis to first generate the part of the dependency graph that meets the given patterns in the tracking statement. However, it is impossible for *APTrace* to know which part meets the given patterns until it has fully explored the whole dependency graph.

To achieve the prioritization, *APTrace* uses a state propagation algorithm. Assume that the list of nodes declared in the tracking statement is $n_1 \rightarrow n_2 ... \rightarrow n_k$. The *Dependency Graph Maintainer* assigns the node n_i a state s_i. The starting point has the state s_1. During backtracking analysis, if the *Dependency Graph Maintainer* finds a node, $curr$, with the state s_i, has a successor, $succ$, that meets the constraints of $n_i \rightarrow n_{i+1}$ (where $i + 1 < k$), the *Dependency Graph Maintainer* assigns $succ$ the state s_{i+1}.

These states will be used by the *Executor* during backtracking analysis. For two nodes n_i and n_j, assume their states are s_i and $s_j, j < i$ respectively. The *Executor* always explores n_i before n_j.

3) Refiner: The purpose of having the *Refiner* is to avoid rerunning the whole backtracking analysis when users have changed the search conditions during the responsive execution.

To do this, the *Refiner* leverages the following solution. Assume that, at a certain time, the users pause the execution of *APTrace*, update the current BDL, C, to a newer version, C', and then resume the execution. When the *Refiner* gets the updated script of C', it first checks if the starting point specified by C' is the same as the starting point of C. If not, it means that the user wants to conduct a new backtracking analysis from a different starting point. To that end, the *Refiner* abandons the current analysis, clears the dependency graph in the *Dependency Graph Maintainer*, and triggers the *Executor* to start a new backtracking analysis. If the starting point is not changed, the *Refiner* first checks if the intermediate points are changed. If so, the *Refiner* triggers the *Dependency Graph Maintainer* to recalculate the states for each node. To do this, the *Dependency Graph Maintainer* traverses the current explored dependency graph from the starting point and performs the state propagation in Section III-B2. Note that, at this time, the tracking graph is already cached in the memory, it is much faster to do the state propagation than the first time, which retrieves the data from database. After all states are calculated, the *Refiner* triggers the *Executor* to update nodes in the searching queue and resume backtracking analysis.

IV. Evaluation

The primary research question with *APTrace* is its usefulness in helping security analysts to perform responsive backtracking analysis efficiently and effectively. This high-level research question is evaluated by four different metrics: the necessity of having responsive backtracking analysis, the expressivity of BDL, the responsiveness of *APTrace*, and the efficiency of *APTrace*.

A. Experiment Setup

We implemented *APTrace* in 20K lines of Java code, and deployed it as part of the security solution at NEC Labs America. We collected system events with Windows ETW [20] and Linux Audit messages [21]. The collected system events were stored in a PostgreSQL database.

We monitored and recorded the system activities for 256 hosts in the company. Each day, the security system collects over 538 million system events, which represents 145GB data in the database. In total, we monitored the enterprise system for a three-month period of time and collected over 13 TB data. We deployed *APTrace* on a server with 16 cores (Intel Xeon CPU E52640 v3 @ 2.60GHz) and 64 GB memory.

B. Necessity of Responsive Backtracking Analysis

In this section, we evaluate if it is necessary to have responsive backtracking analysis. We answer this research question from two aspects. The first aspect is to answer: *how often does the dependency explosion problem exist in realistic scenarios of backtracking analysis?* If the dependency explosion problem does not happen frequently, people do not need to try different heuristics and thus there is no need to have a responsive backtracking analysis. The second aspect is: *how difficult is it for people to set an appropriate time limit in the execute-to-complete (all results returned after execution) system?* If people can have a good estimation about when they should stop the execution of backtracking analysis, they may not need to run it responsively. To evaluate this research question, we implemented the baseline backtracking analysis [6] and used it for the evaluation of *APTrace*'s performance. Note that this baseline technique was also adopted and evaluated for different purposes in related work, such as [9], [10], [11], [12].

1) Severity of Dependency Explosion: To answer this research question, we measured how often the baseline backtracking analysis can take a long time and generate a huge dependency graph due to the dependency explosion problem. To do it, we randomly selected 200 events from the database, assumed they were system anomalies that served as the starting point, and performed backtracking analysis on them. For each execution, we limited the running time for two hours to avoid overly long executions.

In our experiment, about 50% of executions lasted for more than 20 minutes. 36% of executions had reached the two-hour time limit. Note that these executions should have taken longer to finish the running. In terms of the size of dependency graphs, more than 36% of executions generated over 1,000 events in their dependency graphs, 26% more than 2,500 events, and 17% even more than 5,000 events. The largest dependency graph had as many as 35,288 events!

Above experimental results show that the dependency explosion problem is not rare in real world. In fact, there is a very high chance that security analysts need to use heuristics to optimize backtracking analysis. If security analysts are using the traditional execute-to-complete engine, they may set a time limit to have a quick response. However, setting an appropriate time limit is very challenging and infeasible, which we explain below.

2) Difficulty of Setting Right Time Limit: The second aspect questioning the necessity of responsive backtracking analysis is whether people can set an appropriate time limit for the current execute-to-complete system. To answer this research question, we evaluated if we could summarize a general piece of knowledge about the appropriate time duration when executing a backtracking analysis so that it did not generate a too-small or too-large dependency graph.

To do the evaluation, we ran the backtracking analysis for all 200 events randomly selected previously with 30 different configurations of execution time. In each configuration the time limit was set to be k minutes, where k was an incremental number from 1 to 30. Then we checked if there was a k to ensure that the generated dependency graph was neither too small nor too large.

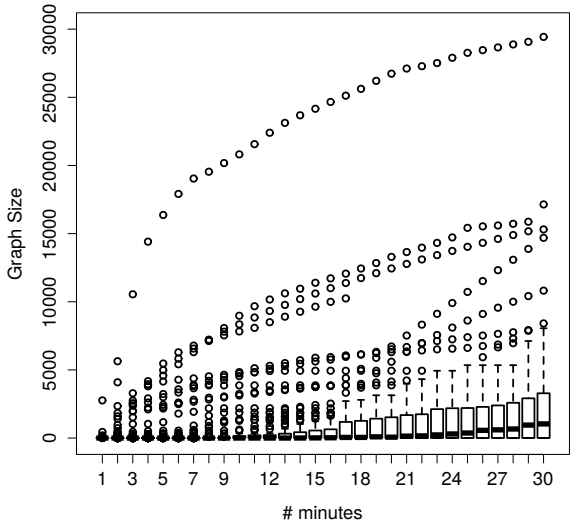

Fig. 4: The size of dependency graph under different thresholds during backtracking analysis

The result is shown in the box plot of Figure 4. In this graph, the y-axis is the size of dependency graph, and the x-axis is the threshold k that represents the number of minutes before backtracking analysis was terminated. Each box denotes the size distribution of dependency graph among all 200 events whose backtracking analysis had the time limit given by the x-axis. From Figure 4, we can see that there are many outliers in each box. The values of points in one box also suffers a dramatic variance. In particular, on average the largest point is 15,079 times larger than the smallest point for each box. The top ten percent of points are 2,857 times larger than the bottom ten percent of points.

Hence, our experiment shows that there is no general knowledge about what the appropriate time limit is for backtracking analysis. Given a certain time limit, it has a substantial chance that security analysts will either get a dependency graph with less than ten nodes or a graph with thousands of nodes. In either way, security analysts are not able to verify their heuristics effectively.

C. Expressivity of BDL

To evaluate the expressivity of BDL, we first express the common heuristics used in related work [6], [9], [12], [16]. Then, we evaluate how people can use BDL to investigate realistic APT attacks through *APTrace*.

1) Expressing Heuristics: In the literature, there are two main types of heuristics: excluding system events, and prioritizing a part of dependency graph. Most of the heuristics in the literature fall into these two categories. The main differences between heuristics in different systems are what type of events should be excluded or prioritized. The basic heuristics for excluding or prioritizing events have been shown

in Section III-A. In this section, we mainly focus on the complex and advanced heuristics.

Quantity Based Heuristics: In these heuristics, people want to exclude or prioritize the events by comparing quantitative features with their neighbors. One typical example is prioritizing data uploads [12]. In this case, security analysts want to prioritize during backtracking analysis the processes that upload sensitive files to the network. The simple approach is to check if a process reads a sensitive file and then writes to the network. However, such a simple approach may include a lot of false positives. For example, the Adobe Reader may also open a sensitive file and then send some log information to the server. The simple approach cannot distinguish such log collection from data uploading. A more sophisticated approach is to check if the amount of data sent to the network is at least as big as the amount of data read from the sensitive file. To achieve such an advanced event pattern, the security analysts can use the code in Program 2, where the last condition checks the amount. Note that we introduce the keyword "prioritize" in the program for the purpose of explanation. In practice, the new heuristic is added directly to the BDL script during responsive backtracking analysis, as described in Section III-A.

```
1 prioritize [type = file and src.path = "
      sensitivefile"] <- [type = network and dst.ip =
       "unkownIP" and amount >= size]
```

Program 2: BDL to connect multiple anomalies

Excluding Read-Only Files and Write-Through Processes In some cases, people may also want to exclude "read-only" files and "write-through" processes [6]. A "read-only" file is the file that is not written during the period of time being analyzed. A process is called a "write-through" process if it is only connected to another process. In many cases, a "write-through" process is a helper process. In particular, it takes the input from its parent process, does the processing, and returns the result to the parent. These two special types of heuristics correspond to two attributes "isReadOnly" and "isWriteThrough" of the destination node of a process. They can be used as part of the condition in BDL, as shown in Program 3, for each event.

```
1 where proc.dst.isReadonly = true or proc.dst.
      isWriteThrough = true
```

Program 3: Read-only files and write-through processes

D. Usefulness of BDL In Real Attack Cases

To evaluate the usefulness of BDL, we recruited two teams in the company, a red team and a blue team. The red team consisted of six experienced security experts who prepared the environment, set up the monitoring system, and created attacks from an adversarial point of view. The blue team consisted of two newly recruited employees who performed backtracking analysis to identify the root cause of these attacks using *APTrace*. Before the analysis, the blue team were not aware of the attack information, such as how or when the attack was conducted except the general security knowledge. In the end, the red team created five attack cases. These cases were motivated by the real-world attacks in the literature. For each of the attacks, the blue team used BDL to express the heuristics independently and then applied these heuristics to *APTrace*. During this process, they also recorded the analysis information such as the heuristics applied and the time spent. For the user configurable parameter k in the execution-window partitioning algorithm, they used the empirical value eight. The evaluation results are reported in Table I, where we measured the size of dependency graph without heuristics (**No Opt**), the size of dependency graph with heuristics (**Opt**), the number of heuristics applied (**# Heuristics**), and the total execution time (in average) of *APTrace* to generate the dependency graph with heuristics (**Time**). Note that we did not report the execution time that it took for *APTrace* to generate the full dependency graph without heuristics because, for every case, it took more than four hours to generate the dependency graph without any heuristics and the execution was terminated before it was completed.

From Table I, we can see that BDL is effective to express the heuristics of real-world attack cases. With BDL, security analysts in the blue team are able to express the heuristics that reduce the size of dependency graph and the execution time of analysis by more than 99.5% and 96.5%, respectively. Below we present the details of the first two attack cases and discuss how the blue team achieved responsive backtracking analysis using *APTrace* in the experiments.

A1: Phishing Email This is the motivating example mentioned in Section II. The alert raised in an anomaly detector was the communication between `java.exe` and an external IP address.

The attack scenario reconstruction proceeded as follows. In the beginning, the security team only received an alert regarding the communication between `java.exe` and the external IP address. At this time, they had no knowledge of the root cause of the alert. In fact, the security team could only run a basic backtracking analysis from the given alert without any guidance. To do so, they ran the sample BDL script as shown in Program 4. The tracking process searched the data dependencies of the starting point within one month, which was given by the "from to" statement at line 1. It also directed *APTrace* to store the dependency graph in the `result.dot` file.

```
1 from "03/26/2019" to "04/26/2019"
2 backward ip alert[dst_ip= "an external IP",
      subject_name = "java.exe" and event_time = "
      04/26/2019:16:31:16" and action_type = "write"]
       -> *
3 output = "./result.dot"
```

Program 4: BDL for the basic backtracking of A1

Once the code of Program 4 was provided as input, *APTrace* started to update the dependency graph progressively. After viewing two events in less than three minutes, the security team noticed that the tracking graph involved `excel.exe`,

Attack	Description	No Opt	Opt	# Heuristics	Total analysis time
Phishing Email [17]	The motivating example	30.75K	140	2	10m
Malicious Excel Macro [22]	A malicious Excel makes Sqlserver run the command line abnormally	5.34K	45	3	10m
Shell Shock [23]	Use the Shell Shock vulnerability of Apache to execute a bash, then steal sensitive data, upload the data through Apache	32.25K	154	2	5m
Cheating Student [11]	The student steals the credential of the admin laptop, then uploads a backdoor program to the server, and then changes his score	43.64K	152	3	9m
wget-unzip-gcc [24]	A ZIP file containing malicious source code is downloaded, unzipped, compiled and executed. The malware then steals the sensitive data	121.26K	75	2	10m

TABLE I: The summary of five attack cases

which might load a lot of `dll` files. At this time, the security team paused backtracking analysis, quickly searched for other alerts from the backend anomaly detector in the recent three months, and found that there were no suspicious modifications to the `dll` files or new malicious `dll` files dropped by the attacker. Hence, the security analysts concluded that the attack was not from the injected code in the `dll` files and they should focus on other data dependencies. Thus, the security team updated the BDL script from Program 4 to Program 5 by adding a new heuristic, which excluded all `dll` files.

```
1 from "03/26/2019" to "04/26/2019"
2 backward ip alert[dst_ip= "an external IP",
      subject_name = "java.exe" and event_time = "
      04/26/2019:16:31:16" and action_type = "write"]
       -> *
3 where file.path != "*.dll"
4 output = "./result.dot"
```

Program 5: BDL for A1 with `*.dll` excluded

After the modification, the security team resumed the execution of *APTrace* and continued monitoring the output. After viewing eight more events in two minutes, the security team noticed that the dependency graph had reached the `findstr.exe` through `findstr.out`. After looking at the first 100 events following `findstr.exe`, the security team realized that `findstr.exe` might scan a lot of files. Thus, it might take a very long time to fully explore the dependency graph after `findstr.exe`. The security team also realized that `findstr.exe` was more likely to be used by `java.exe` rather than the root cause of it. So, the security team paused the analysis again and updated the script from Program 5 to Program 6 to exclude `findstr.exe` from the dependency graph.

```
1 from "03/26/2019" to "04/26/2019"
2 backward ip alert[dst_ip= "an external IP",
      subject_name = "java.exe" and event_time = "
      04/26/2019:16:31:16" and action_type = "write"]
       -> *
3 where file.path != "*.dll" and proc.exename != "
      findstr.exe"
4 output = "./result.dot"
```

Program 6: BDL for A1 with `findstr.exe` excluded

Finally, after resuming the execution again for about four minutes, the security analysts found the `outlook.exe` and the sockets connected to it by checking about 30 more events. At this moment, the security team found that `java.exe` was created by `excel.exe`, which was spawned by `outlook.exe`. Thus, they confirmed that the root cause of `java.exe` was a phishing email. Until this point, the security team had spent about ten minutes in backtracking analysis and checked about 140 events in total.

A2: Malicious Excel Macro In this attack scenario, the attacker took advantage of the Excel macro validation vulnerability [22]. The main dependency graph of this attack is shown in Figure 5. The attacker would like to dump a backdoor to an internal host (Host 2). However, Host 2 could not be accessed from the external network. Thus, the attacker hacked another host (Host 1) in the same company as Host 2 by leading the user of Host 1 to download an Excel file (`data.xls`) with a malicious macro through the browser. When the user of Host 1 opened the Excel file, the macro was executed and another malware named `java.exe` was created and executed. Since the anomaly detector lacked the capability to verify signatures and the malware `java.exe` had a normal name, it bypassed the anomaly detector in the company. Then the `java.exe` connected to the SQL server of Host 2 and executed a batch script through the shell interface of SQL server. Finally the batch script dropped to Host 2 the back door named `qfvkl.exe` and executed it.

In this attack, the anomaly detector raised an alert when the SQL server started the `cmd.exe` since it was an abnormal activity for SQL server. To reconstruct this attack scenario, the security team started backtracking analysis from the alert raised by the anomaly detector. In the beginning, the security team used the sample BDL script in Program 7 for backtracking analysis. After viewing five events in two minutes, the security team saw the `dll` files and decided to exclude the `dll` files. Thus, the security team paused the analysis and updated the BDL script from Program 7 to Program 8 by adding a new heuristic.

```
1 from "03/03/2019" to "04/03/2019"
2 backward proc p[exename = "cmd" and event_time = "
      04/03/2019:11:34:45" and action_type = "start"
      and subject_name = "sqlserver.exe"] -> *
3 output = "./result.dot"
```

Program 7: BDL for A2 starting from the alert

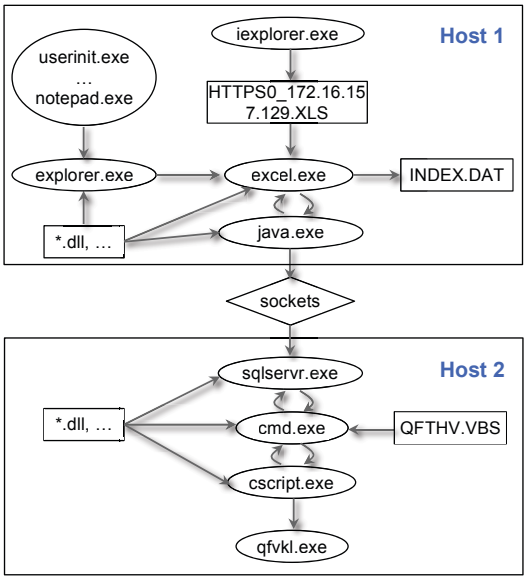

Fig. 5: Dependency graph of case 2 - Malicious Excel Macro

```
from "03/03/2019" to "04/03/2019"
backward proc p[exename = "cmd" and event_time = "
      04/03/2019:11:34:45" and action_type = "start"
      and subject_name = "sqlserver.exe"] -> *
where file.path != "*.dll"
output = "./result.dot"
```

Program 8: BDL for A2 with `*.dll` excluded

The security team resumed backtracking analysis, and after viewing one more event in less than one minute, they found that the `java.exe` on Host 1 was connected to the `sqlservr.exe` through the network. However, by looking at the parent folder of `java.exe`, which was the Document folder, the security team suspected that such a `java.exe` was malware. Then the security team updated the BDL script to make backtracking analysis focus on the socket connection to the `java.exe` first and deprioritized other connections. The code is shown in Program 9.

```
from "03/03/2019" to "04/03/2019"
backward proc p[exename = "cmd" and event_time = "
      04/03/2019:11:34:45" and action_type = "start"
      and subject_name = "sqlserver.exe"] -> ip i[
      dst_ip = "host2" and src_ip = "host1" and
      subject_name = "java.exe"] -> *
where file.path != "*.dll"
output = "./result.dot"
```

Program 9: BDL for A2 with socket connections included

After running Program 9 for four minutes, backtracking analysis finished the exploration of ten events. Then it reached the node `explorer.exe`, which is the Windows File Explorer. In Windows, when people open a file, the File Explorer will not only open the target file but also open other files in the same folder to retrieve their meta information. Thus, in many cases, the Windows File Explorer may introduce millions of unrelated data dependencies. At this time, the security team checked the first 20 successors of `explorer.exe`

	Average	STD	90%	95%	99%
Baseline	7	210	58	613	1,149
APTrace	2	20	4	9	19

TABLE II: Waiting time between updates (unit: second)

and estimated that there were no suspicious events related to `explorer.exe`. Thus, the security team decided to remove `explorer.exe` from backtracking analysis and updated the script to Program 10. At this moment, this change was temporary. If the security team failed to identify the root cause of the alert after removing `explorer.exe`, they could redo the analysis quickly and explore the dependencies of `explorer.exe` again.

```
from "03/03/2019" to "04/03/2019"
backward file p[exename = "cmd" and event_time = "
      04/03/2019:11:34:45" and type = "start" and
      subject_name = "sqlserver.exe"] -> *
where file.path != "*.dll" and file.path != "
      explorer.exe"
output = "./result.dot"
```

Program 10: BDL for A2 with explorer.exe excluded

Finally, after Program 10 had explored ten more events in three minutes, backtracking analysis reached the browser, `iexplorer.exe`, which confirmed that the attack was originated from downloading an Excel file from the Internet. In the reconstruction process of this attack, the security team checked in total 45 events in less than ten minutes. Since the security team had successfully found the root cause, they did not search the dependencies of `explorer.exe` again.

E. Responsiveness of APTrace

The third research question is how responsive is *APTrace*. In other words, how long do people need to wait between two updates? This is important because it is annoying and inefficient for people to wait for a long time to see nothing. To answer this question, we measured the waiting time between updates for the same 200 backtracking analysis cases as randomly selected in Section IV-B1. For each update to the dependency graph, we recorded its time stamp. Then, we calculated the delta, which is named as the "update time" of time stamps between any two consecutive updates.

The result is shown in the row "*APTrace*" in Table II, where we report the average, the standard deviation, the 90 percentile, the 95 percentile, and the 99 percentile. The unit is in seconds.

Our experiment shows that *APTrace* can update the dependency graph in a fast speed. On average, it only takes two seconds for *APTrace* to generate a new update to security analysts. In 99% of cases, the update time is only 19 seconds. This means that only in very rare cases, security analysts need to wait for more than 20 seconds to get a new update. In this case, we conclude that *APTrace* has enough responsiveness to support effective tunings to backtracking analyses.

Effectiveness of Execution-Window Partitioning Algorithm One aspect of the research question regarding the responsiveness of *APTrace* is how effective is the execution-window partitioning algorithm. To answer this question, we

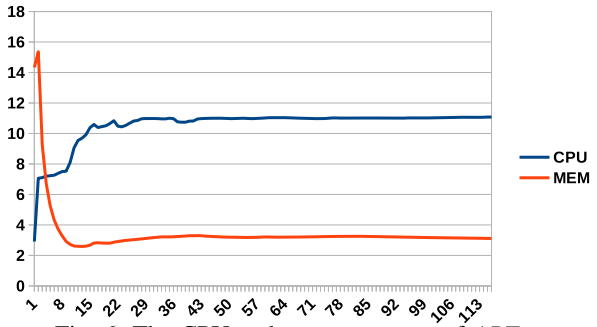

Fig. 6: The CPU and memory usage of *APTrace*

measured the update time of the baseline method, which ran without applying the execution-window partitioning algorithm, and compared it with *APTrace*.

As shown in the row "Baseline" in Table II, the baseline method without the execution-window partitioning algorithm has the average update time as seven seconds. This is a moderate value. However, the baseline method has a considerably higher chance to block the execution of backtracking analysis. This is reflected on the 90, 95, and 99 percentiles. In particular, for the baseline method, in 10% of times, an update can take about one minute. In 5% of times, it can take more than 10 minutes to update the dependency graph. For some extreme cases, this can be as long as near 20 minutes. Although this chance looks low, in practice, one backtracking analysis may need to update the dependency graph for hundreds or even thousands of times. 1% of chance means that nearly in every backtracking analysis, there will be at least one update being blocked for more than 20 minutes. In comparison, *APTrace* with the execution-window partitioning algorithm rarely takes more than 20 seconds to generate an update. Particularly, *APTrace* reduces the 90 percentiles by 15 times, the 95 percentiles 68 times, and the 99 percentiles 57 times.

F. Efficiency of APTrace

The fourth aspect of the usefulness of *APTrace* is its efficiency. To answer this research question, we quantify its runtime overhead when performing the responsive backtracking analysis. We measured the average CPU and memory usage of the 200 backtracking analysis cases randomly selected in Section IV-B1. In our experiment, we measured the CPU usage in Solaris mode.

The result is shown in Figure 6, where the y-axis is the average CPU and memory usage in percentage, and the x-axis is the execution time in minutes. In our experiment, *APTrace* has the highest memory usage (i.e., 15%) in the early stage of backtracking analysis. This is because it needs to initialize the database, compile the BDL, and load the heuristics. After the beginning stage, the memory usage drops to about 3%. The CPU usage of *APTrace* raises from 3% to 11% during the execution.

This runtime overhead is moderate for an enterprise-level server. Thus, we conclude that *APTrace* can be deployed as part of security solutions in a real-world enterprise.

V. Related Work

Backtracking analysis is proposed by King and colleagues [6]. It is then improved and applied in many other studies [9], [10], [11], [12], [14], [16]. For example, PrioTracker [25] and NoDoze [26] try to use anomaly based techniques to prune common behaviors. SLEUTH [16] and HOLMES[27] use rule based systems to solve the dependency explosion problem. Our approach addresses a very different problem and is complementary to these techniques. In many cases, people may still need to manually "debug" backtracking analysis to discover very stealthy attacks or verify the result of aforementioned techniques. Under such a situation, our approach provides a unified framework to express different heuristics and enable the responsive backtracking analysis.

There are a few techniques that focus on addressing the dependency explosion problem automatically. ProTracer [11], Beep [13], and LogGC [28] instrument the executables to obtain finer grained system logs to reduce benign events. Ma and colleagues propose a static binary analysis technique to achieve the same goal [15]. In a commercial enterprise, it may not be realistic to assume that people can instrument or have an extensive capability to apply static analysis to all executables. In practice, security analysts still need to try different heuristics manually.

Traditional prevention based security solutions, such as malware detectors [29], [30] and system anomaly detectors [31], [32], have been studied for a long time. These techniques have two limitations. First they can be bypassed by sophisticated obfuscation techniques [33] or have a high false positive rate [32]. Second, these techniques can only detect the malicious behaviors in a system. And they do not detect the intrusion point. As discussed in Section II, only detecting the malicious behaviors is insufficient to defend against APT attacks.

Human knowledge is also introduced to correlate system anomalies from different detectors [34] and guide system anomaly searching [35]. These approaches provide models to allow security analysts to describe high-level information about anomalies. Similar ideas are introduced [36], [37], [38], [39] to facilitate the investigation of software bugs. However, all of them do not focus on backtracking analysis and their frameworks do not support expressing the heuristics for analysis.

The basic idea of our responsive backtracking analysis is motivated by the approximate query processing (AQP) systems [40], [41], [42]. These systems may potentially be used as the underlying infrastructure of *APTrace* to provide responsive backtracking analysis, but they do not directly solve the problem of imbalanced data distribution. As a result, they fail to achieve the responsiveness without leveraging the execution-window partitioning algorithm or similar mechanisms. Gui and colleagues [43] proposed *ProbeQ*, a system to progressively process system-behavioral queries. *ProbeQ* is focused on the progressive processing of a single query instead of backtracking analysis that involves multiple queries.

VI. Conclusions

In this paper, we propose a novel system, *APTrace*, to help security analysts to tune backtracking analysis. This work is motivated by the fact that existing systems fail to fulfill two important system requirements: having a unified framework to express heuristics and supporting responsive backtracking analysis. Our system meets these two requirements in that it provides a domain specific language, BDL, to express different common heuristics and uses an execution-window partitioning algorithm to support the responsive backtracking analysis. Our evaluation results demonstrate *APTrace* can be used to help security analysts to tune backtracking analysis efficiently and effectively in real-world attack cases. In particular, *APTrace* reduces the top 1% longest waiting time between two consecutive updates by 57 times compared to the baseline solution.

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] "Sony Reports 24.5 Million More Accounts Hacked." https://www.darkreading.com/attacks-and-breaches/sony-reports-245-million-more-accounts-hacked/d/d-id/1097499.

[4] "Ebay inc. to ask Ebay users to change passwords." https://www.ebayinc.com/stories/news/ebay-inc-ask-ebay-users-change-passwords/.

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

[6] S. T. King and P. M. Chen, "Backtracking intrusions," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 223–236, 2003.

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

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

[9] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The Taser Intrusion Recovery System," in *SOSP*. New York, NY, USA: ACM, 2005, pp. 163–176.

[10] S. Krishnan, K. Z. Snow, and F. Monrose, "Trail of Bytes: Efficient Support for Forensic Analysis," in *CCS*. New York, NY, USA: ACM, 2010, pp. 50–60.

[11] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting," in *NDSS*, 2015.

[12] T. Wuchner, M. Ochoa, and A. Pretschner, "Malware Detection with Quantitative Data Flow Graphs," in *ASIA CCS*. New York, NY, USA: ACM, 2014, pp. 271–282.

[13] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition." in *NDSS*, 2013.

[14] S. King, Z. M. Mao, D. C. Lucchetti, and P. M. Chen, "Enriching Intrusion Alerts Through Multi-Host Causality," in *NDSS*, 2005.

[15] 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 *ACSAC*. New York, NY, USA: ACM, 2015, pp. 401–410.

[16] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data," in *USENIX Security*. Vancouver, BC: USENIX Association, 2017, pp. 487–504.

[17] CVE, "Win32k Elevation of Privilege Vulnerability, CVE-2015-1701," 2015, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1701.

[18] N. Ruff, "Windows memory forensics," *Journal in Computer Virology*, vol. 4, no. 2, pp. 83–100, May 2008.

[19] S. C. LLC, "Why is notepad.exe connecting to the internet?" 2013, https://blog.cobaltstrike.com/2013/08/08/why-is-notepad-exe-connecting-to-the-internet/.

[20] I. Park and R. Buch, "Event Tracing for Windows: Best Practices." in *CMG*, 2004, pp. 565–574.

[21] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. O'Reilly Media, Inc., 2005.

[22] CVE, "CVE-2008-0081," 2008, http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2008-0081.

[23] ——, "ShellShock, CVE-2014-6271," 2014, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271.

[24] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High Fidelity Data Reduction for Big Data Security Dependency Analyses," in *CCS*. New York, NY, USA: ACM, 2016, pp. 504–516.

[25] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security." in *NDSS*, 2018.

[26] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage." in *NDSS*, 2019.

[27] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," *arXiv preprint arXiv:1810.01594*, 2018.

[28] K. H. Lee, X. Zhang, and D. Xu, "LogGC: garbage collecting audit log," in *CCS*. New York, NY, USA: ACM, 2013, pp. 1005–1016.

[29] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs," in *CCS*. New York, NY, USA: ACM, 2014, pp. 1105–1116.

[30] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *CCS*. New York, NY, USA: ACM, 2007, pp. 116–127.

[31] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," in *S&P*, 1999, pp. 133–145.

[32] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, vol. 51, no. 12, pp. 3448 – 3470, 2007.

[33] I. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," in *BWCCA*, Nov 2010, pp. 297–300.

[34] K. Tabia, S. Benferhat, P. Leray, and L. Mé, "Alert correlation in intrusion detection: Combining AI-based approaches for exploiting security operators knowledge and preferences," *SecArt*, 2011.

[35] C. Zhong, D. S. Kirubakaran, J. Yen, P. Liu, S. Hutchinson, and H. Cam, "How to use experience in cyber analysis: An analytical reasoning support system," in *ISI*, June 2013, pp. 263–265.

[36] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE*. ACM, 2002, pp. 291–301.

[37] J. P. Near and D. Jackson, "Derailer: interactive security analysis for web applications," in *ASE*. ACM, 2014, pp. 587–598.

[38] A. Beaugnon, P. Chifflier, and F. Bach, "ILAB: An Interactive Labelling Strategy for Intrusion Detection," in *RAID*. Springer, 2017, pp. 120–140.

[39] C. Muelder, K.-L. Ma, and T. Bartoletti, "Interactive visualization for network and port scan detection," in *RAID*. Springer, 2005, pp. 265–283.

[40] B. Chandramouli, J. Goldstein, and A. Quamar, "Scalable Progressive Analytics on Big Data in the Cloud," *VLDB*, vol. 6, no. 14, pp. 1726–1737, Sep. 2013.

[41] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online Aggregation," in *SIGMOD*. New York, NY, USA: ACM, 1997, pp. 171–182.

[42] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A Platform for Scalable One-pass Analytics Using MapReduce," in *SIGMOD*. New York, NY, USA: ACM, 2011, pp. 985–996.

[43] J. Gui, X. Xiao, D. Li, C. H. Kim, and H. Chen, "Progressive processing of system-behavioral query," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 378–389.