MPI: Multiple Perspective Attack Investigation with Semantics Aware Execution Partitioning

Shiqing Ma
Purdue University

Juan Zhai
Nanjing University

Fei Wang
Purdue University

Kyu Hyung Lee
University of Georgia

Xiangyu Zhang
Purdue University

Dongyan Xu
Purdue University

Abstract

Traditional auditing techniques generate large and inaccurate causal graphs. To overcome such limitations, researchers proposed to leverage execution partitioning to improve analysis granularity and hence precision. However, these techniques rely on a low level programming paradigm (i.e., event handling loops) to partition execution, which often results in low level graphs with a lot of redundancy. This not only leads to space inefficiency and noises in causal graphs, but also makes it difficult to understand attack provenance. Moreover, these techniques require training to detect low level memory dependencies across partitions. Achieving correctness and completeness in the training is highly challenging. In this paper, we propose a semantics aware program annotation and instrumentation technique to partition execution based on the application specific high level task structures. It avoids training, generates execution partitions with rich semantic information and provides multiple perspectives of an attack. We develop a prototype and integrate it with three different provenance systems: the Linux Audit system, ProTracer and the LPM-HiFi system. The evaluation results show that our technique generates cleaner attack graphs with rich high-level semantics and has much lower space and time overheads, when compared with the event loop based partitioning techniques BEEP and ProTracer.

1 Introduction

Provenance tracking is critical for attack investigation, especially for *Advanced Persistent Threats* (APTs) that are backed by organizations such as alien governments and terrorists. APT attacks often span a long duration of time with a low profile, and hence are difficult to detect and investigate. A provenance tracking system records the causality of system objects (e.g. files) and subjects (e.g. processes). Once an attack symptom is detected, the analyst can utilize the provenance data to understand the attack including its root cause and ramifications. Such inspection is critical for timely response to attacks and the protection of target systems. Most existing techniques [38, 46, 49, 50, 59] entail hooking and recording important system level events (e.g. file operations), and then correlating these events during an offline investigation process. The correlations have multiple types: between two processes such as a process creating a child process through *sys_clone()*; between a process and a system object, e.g., a process reads a file through *sys_read()*. However, these techniques suffer from the *dependence explosion* problem, especially for long running processes. The reason is that a long running process may have dependencies with many objects and other processes during its lifetime although only a small subset is attack related. For instance, a Firefox process may visit numerous pages over its lifetime while only one page is related to a drive-by-download attack.

Researchers proposed to partition execution to units so that only the events within a unit are considered causally related [43, 46]. For instance, the execution of a long running server is partitioned to individual units, each handling a request. Although existing execution partitioning based systems such as BEEP [43] and ProTracer [46] have demonstrated great potential, they partitioned execution based on event handling loops. That is, each iteration of an event handling loop is considered a unit. Despite its generality, such a partitioning scheme has inherent limitations. (1) Event loop iterations are too low level and cannot denote high level task structure. For instance, in UI programs, an event loop iteration may be to handle some user interaction. (2) There are often inter-dependencies across units. Therefore, BEEP and ProTracer rely on a training phase to detect such dependencies in the form of low level memory reads and writes. Achieving completeness in training is highly challenging. Note that the problem could not be addressed even when source code is provided because there are typically a lot of program dependencies across event loop iterations and only a sub-

set of them are important. (3) A high level task is often composed of many units (e.g., those denoting event loop iterations in multiple worker threads that serve the same high level task). Ideally, we would like to partition execution based on the high level task structure.

Note that high level task structure is application specific. Therefore, developers' input on what denotes a task/unit is necessary. We observe that a high level task/unit has its corresponding data structure in the software. Our proposal is hence to allow the developer/user to inform our system what task/unit structure they desire by annotating a small number of data structures (e.g., the tab data structure in Firefox). Our system MPI[1] takes the annotations and automatically instruments (a large number of) program locations that denote unit boundaries through static program analysis. The analysis handles complex threading models in which the executions of multiple tasks/units interleave. The instrumentation emits special syscalls upon unit context switches so that the application specific task/unit semantics is exposed to the underlying provenance tracking systems. MPI allows annotating multiple task/unit structures simultaneously so that the forensic analyst can inspect an execution from multiple perspectives (e.g., tab and domain perspectives for Firefox). This is highly desirable for attack investigation as we will show later in the paper. Asking for developers/users input in audit logging is a strategy adopted in practice. For example, the audit system on Windows, *Event Tracing for Windows* (ETW) requires the developers to explicitly plant auditing API calls in their source code if they would like to perform any customized logging. Nonetheless, reducing manual efforts is critical to the real world deployment of the technique. MPI is highly automated as the user only needs to annotate a few data structures and then the invocations to logging commands are automatically inserted through program analysis. Most of the programs we use in our experiment require only 2-3 annotations for each perspective. In addition, MPI provides a data structure profiler, called the *annotation miner*, to recommend the potential data structures to annotate. As shown in §4.2, it makes the correct recommendations in most cases.

MPI is a general execution partitioning scheme orthogonal to the underlying OS-level provenance collection system. We integrate it with three different provenance collection systems: the widely adopted Linux audit framework, and two state-of-the-art research projects, ProTracer [46][2] and the LPM [23] enabled HiFi [55] system (LPM-HiFi) which features secure audit logging.

In summary, we make the following contributions:

- We propose the novel idea of partitioning execution based on data structures to support different granularities and facilitate multi-perspective, application-semantics-aware attack investigation.
- We develop program analysis and runtime techniques to enable such partitioning. Given a small number of annotations on data structure definitions, program analysis is conducted to identify places that need to be instrumented to emit events at runtime that denote unit boundaries and unit inheritance. The number of such places may be very large, rendering manual instrumentation infeasible.
- We develop an annotation miner that can recommend the data structures to annotate with high accuracy, substantially alleviating the manual efforts.
- We develop a prototype based on LLVM. The evaluation on a set of commonly used Linux applications and three different provenance systems shows that our approach can effectively partition program execution in different granularities. We also use a number of case studies that simulate real-world attacks to demonstrate the strength of the proposed technique, in comparison with BEEP [43] and ProTracer [46].

[1] MPI is short for "Multiple Perspective attack Investigation"

[2] ProTracer is based on BEEP, we replace the BEEP with MPI.

2 Motivation

In this section, we use an example to illustrate the differences between the classic provenance tracking systems [23, 49, 50, 55], the existing event loop based execution partitioning approaches [43, 46], and the proposed approach. This example simulates an important kind of real-world attacks, *watering hole attack* [18, 19],

2.1 Motivating Example

Watering hole is a popular attack strategy targeting large enterprises such as Apple [11] and Google [12]. The adversaries do not directly attack the enterprise networks or websites, which are well protected. Instead, they aim to compromise the websites that are frequently visited by the employees of the target enterprise, which are usually much less protected. Recently, there have been a number of real incidents of watering hole attacks, e.g., by compromising Github [8] and CSDN [3]. There are exploit kits (e.g., BeEF [2]) to make it easy to conduct such attacks.

In our example case, a developer in an enterprise opens *Firefox*, and then uses Bing to look for a utility program for file copying. The search engine returns a number of relevant links to technical forums, blogs, wikis and online articles. Some of these links further lead to other relevant resources such as pages comparing similar programs. Some pages host software for download. In many cases, the software was uploaded by other developers. After intensive browsing and researching, the developer settles down on a forum that hosts not only the wanted software,

but also many other interesting resources, including torrents for a few tutorial videos. The developer downloads the program and also a few torrents from the forum. After the download, he starts to use the program. He also uses a p2p software *Transmission* to download the videos described by the torrents.

Unfortunately, the forum website was compromised, targeting enterprises whose developers tend to use the forum for technical discussion and information sharing. The program downloaded, *fcopy*, is malicious. In addition to the expected functionality, the malware creates a reverse TCP connection and provides a shell to the remote attacker. The malware causes unusual network bandwidth consumption and is eventually noticed by the administrator of the enterprise. To understand the attack and prepare for response, the administrator performs forensic analysis, trying to identify the root cause and assess the potential damage to the system. At the very beginning, the binary file *fcopy* is the only evidence. Hence, the creation of the file is used as the symptom event.

2.2 Traditional Solutions

Traditional techniques such as backtrackers [38, 39], audit systems [10] and provenance-aware file systems [50, 55] track the lineage of system objects or subjects without being aware by the applications. These techniques collect system subjects (e.g. processes and threads) and objects (e.g. files, network sockets and pipes) information at run time with system call hooking or Linux Security Modules (LSM) [62], and construct dependency graph or causal graph for inspection. Note that these two terms are interchangeable in this paper. While they use different approaches to trace system information, the graphs generated by these systems are similar.

A general workflow for these techniques is as follows. Starting from the given *symptom* subject or object, they identify all the subjects and objects that the symptom directly and indirectly depends on using backtracking. They also allow identifying all the effects induced by the root cause using forward tracking. For the case mentioned in §2.1, the administrator identifies the Firefox process and all its data sources by backtracking, and then discloses the downloaded files and the operations on these files with forward tracking. Figure 1 shows the simplified graph generated. In this graph and also *the rest of the paper*, we use *diamonds* to represent *sockets*, *oval nodes* to represent *files*, and *boxes* to represent *processes* or *execution units*. In Figure 1, many network sockets point to the Firefox process, and the process points to a large number of files including the torrent files and others like *fcopy*, which reflect the browsing and downloading behaviors of Firefox.

While we only show part of the original graph in Figure 1 for readability, the original graph contains more than 500 nodes in total, with most files and network socket accesses being (undesirably) associated with the Firefox and Transmission nodes. These bogus dependencies make manual inspection extremely difficult.

2.3 Loop Based Partitioning Solutions

It was observed in [43] that the inaccuracy of traditional approaches is mainly caused by long running processes, which interact with many other subjects and objects during their lifetime. Traditional approaches consider the entire process execution as a node so that all the input/output interactions become edges to/from the process node, resulting in considerably large and inaccurate graphs. Take the Transmission process as an example. It has dependencies with many torrent files and network sockets, obfuscating the true causalities (e.g., a torrent file and the corresponding downloaded file).

Event loop based partitioning techniques [43, 45, 46] leverage the observation that long running processes are usually event driven and the whole process execution can be partitioned by the event handling loops (through binary instrumentation). They proposed the concept of *execution unit*, which denotes one iteration of an event handling loop. This fine-grained execution abstraction enables accurate tracing of dependency relationship. It was shown that these techniques can generate much smaller and more accurate dependency graphs. However, these techniques still have the following limitations that hinder their application in the real-world.

Units Are Too Low Level. Assume the administrator applies BEEP/ProTracer to the motivation case in §2.1. He constructs the causal graph starting from the file *fcopy*. He acquires the download event in Firefox, which is associated with the web socket *a.a.a.a*. Then, he traces back to the forum website, and eventually the search engine. As part of the investigation, the administrator applies forward tracking from the search engine page to understand if other (potentially malicious) pages were accessed and if other (potentially malicious) programs were downloaded and used. Since the developer visited many links returned by the search engine, the forward tracking includes many web pages and their follow-ups in the resulting graph. The simplified graph is shown in Figure 2.

In this case, Firefox is used for 5 minutes with 11 tabs containing 7 websites. There are thousands of nodes in the graph. This is because all user interactions like scrolling the web pages, moving mouse pointer over a link and clicking links are processed by unique event loop iterations, each leading to a unit/node. Moreover, Firefox has internal events including timer events to refresh pages. As these events operate on DOM elements, they are connected in the dependency graph due to memory

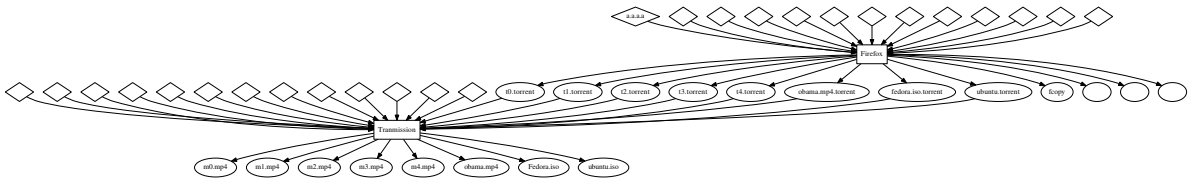

Figure 1: Simplified causal graph for the case in §2.1 generated by traditional solutions (Tool in [16]).

dependencies, making the graph excessive.

The root cause of the limitation lies in that BEEP exposes very low level semantics (i.e., event loop iterations) in partitioning. The onus is on the user to chain low level units to form high level tasks. Unfortunately, BEEP graphs have little information to facilitate this process as they lack high level semantic information such as which high level task (e.g., tab) a low level unit belongs to.

Depending on Training. BEEP and ProTracer are training based due to the difficulty of binary analysis. It requires intensive training to identify the event handling loops and memory accesses that disclose dependencies across units (e.g., one event loop inserts a task to the queue which is later loaded and processed by another event loop). The completeness of the training inputs is hence critical. Otherwise, there may be missing or even wrong causal relations. Note that providing source code does not address this problem as identifying event handling loops and cross-unit dependencies requires in-depth understanding of low level program semantics, which is much easier through dynamic analysis by observing concrete states than static analysis, in which everything is abstract. Specifically, there are a large number of loops in a program. Statically determining which ones are event handling loops is difficult. Furthermore, while static analysis can identify memory dependencies, a lot of cross-unit dependencies should be ignored as they have nothing to do with the high level work flow (e.g., those caused by memory management or statistics collection).

In our motivating example, we did not use the “Go back" button in the initial training of Firefox. As a result, we were not able to get the full causal chain in Figure 2, which was broken at one web page that contains a lot of clicking-link and going-back actions. We had to enhance our training set by providing a going-back case.

Excessive Units. Partitioning based on event handling loops works nicely for server programs, in which one event loop iteration handles an external request and hence corresponds to a high level task. However, in many complex programs, especially those that heavily use threads to distribute workloads or involve intensive UI operations, event loop iterations do not align well with the high level tasks. As a result, it generates excessive small units that do not have much meaning. For example, in GUI programs, units are generated to denote the large number of GUI events (e.g., key strokes), even though all these events may serve the same high level task.

Consider the p2p program Transmission. Figure 3 shows its event handling loop in the main function of the daemon process. After parsing options, loading settings and torrent files (line 2-3), the daemon goes to a loop which exits only when the user closes the program (i.e., set *closing* to *TRUE*). In each iteration of the loop, it waits for 1 second (line 6), updates the torrent status and logs some information (line 7). Due to the nature of p2p protocol, downloading a single file requires thousands of loop iterations, leading to thousands of units in BEEP.

In many situations, there may not be any system events within these small units. For example, GUI programs monitor and handle frequent events such as page scroll. However, not all of them lead to system calls. Thus BEEP ends up with many “UNIT_ENTER” and “UNIT_EXIT” events without any system calls in between. These useless units waste a lot of space and CPU cycles. While existing techniques [22, 44, 46, 67] can remove redundant events, they cannot prevent these events from being generated in the first place.

These limitations are rooted at the misalignment between the rigid and low level execution partitioning scheme based on event loops. Ideally, the units generated by a partitioning scheme would precisely match with the high level logic tasks. MPI *aims to achieve this goal.*

2.4 Our Approach

The overarching idea of this paper is that high level tasks are reflected as data structures. MPI *allows the user to annotate the data structures that correspond to such tasks. It then leverages program analysis to instrument a set of places that indicate switches and inheritances of tasks to achieve execution partitioning.* Note that there may be multiple perspectives of the high level tasks involved in an execution, denoted by different data structures. Hence, MPI allows annotating multiple data structures, each denoting an independent perspective. To reduce the annotation efforts, MPI provides a profiler that can automatically identify the critical data structures (Figure 3.2). Note that allowing developers/users to insert logging related annotations/commands to software source code is a practical approach for system auditing. The Windows auditing system, *Event Tracing for Windows* (ETW), requires the developers to explicitly plan customized events to their

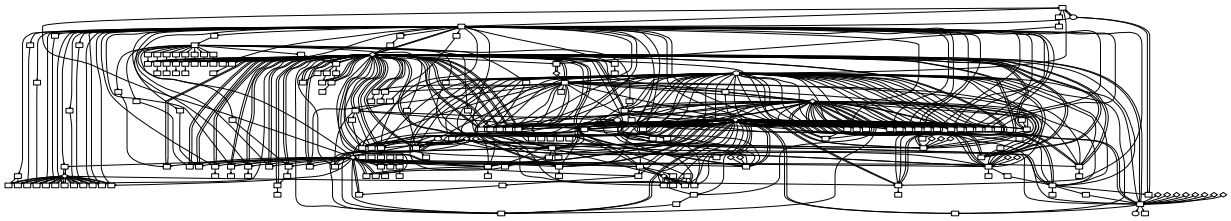

Figure 2: Simplified backtracking causal graph for the case in §2.1 with event loop based partitioning technique. It only shows the causal relationship within the Firefox process (runs for 5 minutes with 11 tabs and 7 websites). The tool used can be found in [16].

```
int main( int argc, char ** argv ) {
    // parse options and session, load torrents
    torrents = tr_sessionLoadTorrents(mySession, ctor, NULL);
    // event loop
    while( !closing ) {
        tr_wait_msec( 1000 ); /* sleep one second */
        // update and log and so on
    }
    // close program and sessions
    return 0;
}
```

Figure 3: Event handling loop of *Transmission* (version 2.6)

software before deployment [5, 20]. These commands generate system events at runtime. In our design, we only require the developer to annotate (a few) task oriented data structures, MPI automatically instruments a much larger number of code places based on the annotations.

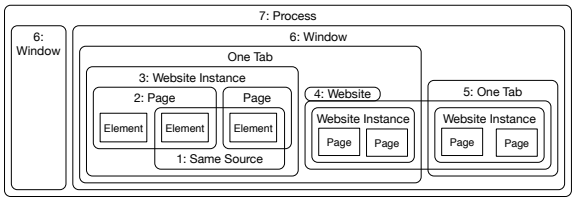

Figure 4: Firefox Partitioning perspectives

Figure 4 presents a few possible perspectives of Firefox execution. By annotating the appropriate data structures, we can partition a Firefox execution into sub-executions of various windows (perspective 6), tabs (perspective 5), websites/domains (perspective 4), website instances (perspective 3), individual pages (perspective 2), and even the sources of individual DOM elements (perspective 1). Observe that some of the perspectives are cross-cutting. For instance, a tab may show pages from multiple domains whereas pages from the same domain may appear in multiple tabs. *A prominent benefit of such partitioning is to expose the high level semantics of the application to the underlying provenance tracking system.*

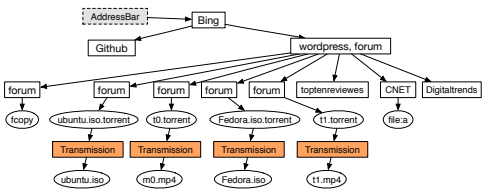

Figure 5: Simplified MPI causal graph for the case in §2.1 with Firefox partitioned by tabs

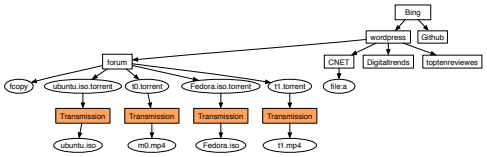

Figure 6: Simplified MPI causal graph for the case in §2.1 with Firefox partitioned by web sites

Figure 5 shows the causal graph for the attack example when we partition the execution of Firefox by its tabs and Transmission by the files being downloaded. Each rectangle represents the life time of a tab. Observe that the Bing tab leads to the wordpress tab, which also shows the forum main page. A number of forum pages are displayed on separate tabs, each of which leads to the download of a torrent file through a Transmission unit. In contrast Figure 6 shows the causal graph when we partition the execution of Firefox by the websites/domains it visits. Observe that all the forum tabs are now collapsed to a single forum node. It clearly indicates that fcopy and the torrent files are downloaded from the same domain. Compared to the BEEP graph in Figure 1, these graphs are much smaller and cleaner, precisely capturing the high level workflow of the execution. *Note that these graphs cannot be generated by directly querying/operating-on the BEEP log, which has only very low level semantic information (i.e., event loop iterations).*

Advantages Over Event Loop Based Partitioning. We can clearly see data structure based partitioning system MPI addresses the limitations of event loop based partitioning. ① Units are no longer based on low level loop iterations. The inspector does not need to manually chain many such low level units to form a high level view of the execution. ② Dependency identification is made easy. Training is no longer needed. The memory dependencies that are needed to chain the low level event loop units are no longer necessary because these low level units are automatically classified to a high level unit in MPI. The incidents of missing causality due to incomplete training can be avoided. For instance, Firefox uses multiple threads to load and render the many elements on a page, which induces lots of memory dependencies across event loop units. But if we look at the execution from the tab perspective, these memory dependencies are no longer inter-unit dependencies that need to be explicitly cap-

tured. ③ Excessive (small and non-informative) units are prevented from being generated. All nodes representing timer event for Transmission will be merged into one node. Moreover, MPI provides great flexibility for attack investigation by supporting multiple perspectives. Enabling these perspectives is impossible if the appropriate semantic information is not exposed through MPI.

One may argue that event loop based partitioning can be enhanced by annotating event loops and cross-unit memory dependencies. However, such annotations are so low-level that (1) they require a lot of human efforts due to the large number of places that need to be annotated (e.g., the memory dependencies), and (2) they expose low-level and sometimes non-informative semantics such as mouse moves and timer events. In addition, the partitioning is solely based on event handling and hence cannot provide multiple perspectives.

3 Design

3.1 Overview

The overall process of analysis and instrumentation is shown in Figure 7. The user first annotates the program source code to indicate unit related data structures under the help of the annotation miner, which is essentially a data structure profiler. The analysis component, implemented as a LLVM pass, takes the annotations and analyzes the program to determine the places to instrument (e.g., data structure accesses denoting unit boundaries). The graph construction is using a standard algorithm, and details can be found in Appendix B.

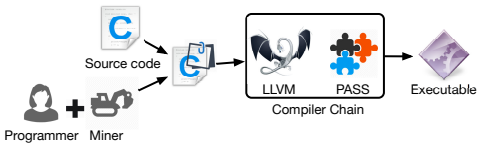

Figure 7: MPI workflow

3.2 Annotations

Basic Annotations. Let us review how the Linux kernel conducts context switching internally, which inspires our approach to unit switching. Specifically, ① a *task_struct* with a unique *pid* identifies an individual process; ② a variable *current* is used to indicate the current active process. Processes can communicate through inter-process communication (IPC) channels like *pipes*. In order to perform unit switching, we need to identify the *unit data structure* that is analogous to *task_struct* and used to store per-unit information, a field/expression that can be used to differentiate unit instances as the identifier, and a variable that stores the current active unit. Note that there may not be an explicit task data structure in a program. Any data structure that allows us to partition an execution to disjoint autonomous units can serve as a unit data structure. Also, we need to know the variables that serve as communication channels between different unit instances. Thus we need the following types of annotations.

① *@indicator* annotates the variable/field that is used to indicate the possible switches between different unit data structure instances (similar to the variable *current* in Linux kernel). The user can choose to annotate multiple indicator variables/fields, one for each perspective. A unique id is assigned to each type of indicator.

② *@identifier* is an expression used to differentiate the instances of a unit data structure (similar to the data field *pid*). This expression can be a field in the data structure or a compound operation over multiple fields. Since an identifier must be paired up with the corresponding indicator, we allow providing an indicator id as part of the identifier annotation.

③ *@channel* annotates the variables/fields that serve as "IPC channels" between two different *unit data structure* instances (similar to *pipes*). It contains a unique id number, and a parameter indicating which field stores the data that induces inter-unit dependencies.

```
1  // in file src/globals.h
2  @indicator=1
3  EXTERN buf_T*curbuf INIT(= NULL);
4
5  // in file src/structs.h
6  typedef struct file_buffer buf_T;
7  // buffer: structure that holds information about one file
8  @identifier=b_ffname, indicator=1
9  struct file_buffer{
10   // associated memline
11   memline_T b_ml;
12   // buffers are orgnized as a linked list
13   buf_T *b_next;
14   buf_T *b_prev;
15   char_u *b_ffname;      // full path file name
16   // TRUE if the file has been changed and not written out
17   int b_changed;
18   // variables for specific commands or local options
19   char_u *b_u_line_ptr;  // for 'U' command
20   int b_p_ai;            // 'autoindent', local opts
21   // other data field like change time or so
22 }; /* file_buffer */
23
24 // in file src/ops.c
25 @channel=channelID, data=(y_current->array)
26 static struct yankreg *y_current;
```

Figure 8: Vim data structure and our annotation

□ *Example.* Vim is a *tabbed* editor with each *tab* containing one or multiple *windows*. Each *window* is a viewpoint of a *buffer*, with each *buffer* containing the in-memory text of a file [17]. A file buffer can be shared by multiple *windows* in the backend, and buffers are organized as a linked list. A natural way to partition its execution is to partition according to the file it is working on, each represented by a *file_buffer* data structure. Figure 8 shows a piece of code which demonstrates our annotations. Vim uses the variable *curbuf* to represent the current active buffer. Consequently, we use *curbuf* as our *indicator* variable.

Line 2 shows the indicator annotation. The annotation has an id to distinguish different indicators for various granularities/perspectives. The id is used to match with the corresponding *@identifier* annotation. Vim creates a buffer for each file. We can hence use the absolute file path in the OS to identify each file buffer instance. Line 8 shows the *@identifier* annotation. It has two parts: ① an expression used to differentiate instances; and ② an indicator id used to match with the corresponding *@indicator* annotation. In this case, field *b_ffname* is the identifier with id 1. Vim maintains its own clip board to support internal copy(cut)-and-paste operations. When the user cuts or copies data from a *file_buffer*, it sets the field *y_current→array*. When the user performs a paste operation, it reads data from the variable and puts the data to the expected position. In this case, *y_current→array* can be considered as the IPC channel between the two different *file_buffer* instances. Line 25 shows the channel annotation. It contains a unique id for the channel (analogous to a file descriptor), and the reference path to the field. Note that this is to support communication using the Vim clip board. Our system also supports inter- or intra-process operations through the *system* clip board by tracking system level events.

Threading Support. In order to improve responsiveness, modern complex applications heavily rely on threads to perform asynchronous sub-tasks. More specifically, the main thread divides a task into multiple subtasks that can proceed asynchronously and dispatches them to various (background) worker threads. A worker thread receives sub-tasks from the main thread and also other threads and processes them in the order of reception. It can also further break a sub-task to many smaller sub-tasks and dispatch them to other threads, including itself. This advanced execution model makes partitioning challenging because *we need to attribute the interleaved sub-tasks to the appropriate top level units*. In event loop based partitioning techniques [43, 46], all the event handling loops from various threads need to be recognized during training. More importantly, multiple event loop iterations (across multiple threads but within an application) may be causally related as they belong to the same task. The correlations are reflected by memory dependencies. As such, the training process needs to discover all such dependencies. Otherwise, the provenance may be broken. Unfortunately, memory dependencies are often path-sensitive and it is very difficult to achieve good path coverage. It is hence highly desirable to directly recognize the logic tasks, which are disclosed by corresponding data structures, instead of chaining low level event loop based units belonging to a logic task through memory dependencies. □ *Example.* Figure 9 illustrates a substantially simplified example of the Firefox execution model. It corresponds to an execution that loads two pages (in two respective tabs). Specifically, each box represents a thread and each colored bar (inside a box) denotes an iteration of the event handling loop (and hence a unit in BEEP/ProTracer). Observe that at step ①, the loading of tab1 first dispatches a Domain Name Server (DNS) query to a DNS thread, and then (step ③) posts a connection request to the socket thread to download the page. At step ④, the socket thread informs the main thread that the data is ready. The main thread leverages other threads such as the image decode thread, JS helper thread, and compositor thread to decode/execute/render the individual page elements. Note that every thread has interleaved sub-tasks belonging to various tabs. Edges denote memory dependencies across sub-tasks that need to be disclosed during training and instrumented at runtime in BEEP/ProTracer. □

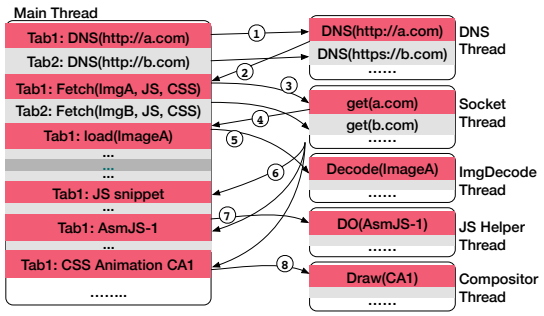

Figure 9: Simplified Firefox execution model

Different from BEEP/ProTracer, our solution is to leverage annotations and static analysis to partition directly according to the logic tasks (e.g. tabs). In order to precisely determine the membership of a sub-task. We introduce the *@delegator* annotation. This annotation is associated with a data structure to denote a sub-task (e.g., the HTTP connection request posted to the socket thread). Intuitively, it is a *delegator* of a top level task (e.g., the HTTP connection request delegates the unit of its owner tab). At runtime, upon the dispatching of a delegator data structure instance (e.g., adding a sub-task to a worker thread event queue), it inherits the current (top level) unit identification. Later when the delegator is used (in a worker thread), the system knows which top level unit the current execution belongs to. There could be multiple layers of delegation. Similar to a unit, a delegator data structure also has an indicator, which is a variable like *current* whose updates may indicate delegation switches. More details can be found in Section 3.3.

□ *Example.* Consider the Firefox execution model. The user can annotate a tab, a window, and/or an iframe as a top level unit. Internally, these are all represented by the same *nsPIDOMWindow* class. They are differentiated by the internal field values. Hence, we provide multiple perspectives by annotating the *nsPIDOMWindow* data structure and using different expressions in the identifier annotations to distinguish the perspectives. Figure 10

```
@identifier=this->GetOuterWindow(2)->mWindowID, indicator=1
@identifier=this->GetTop()->mWindowID, indicator=2
class nsPIDOMWindow {
  @indicator=1
  @indicator=2
  nsCOMPtr<nsIDocument> mDoc;
  // Tracks activation state
  bool mIsActive;
  virtual already_AddRefed<nsPIDOMWindow> GetTop() = 0;
  nsPIDOMWindow *GetOuterWindow()
  { return mIsInnerWindow ? mOuterWindow.get() ?  this; }
  // The references between inner and outer windows
  nsPIDOMWindow          *mInnerWindow;
  nsPIDOMWindow          *mOuterWindow;
  // A unique  (64-bit counter)
  // id for this window.
  uint64_t mWindowID;
  /* other methods and data fields */
};
```

Figure 10: Tab and window annotations in Firefox

shows the annotations for tabs and windows. The indicator id 1 is for tabs and 2 for windows. Any tab or window changes must entail the change of the *mDoc* field, which is used as the indicator. The expressions in the corresponding identifier annotations mean that we can acquire the tab of any given window by getting the second layer outer window, and the top level window by calling *GetTop()*.

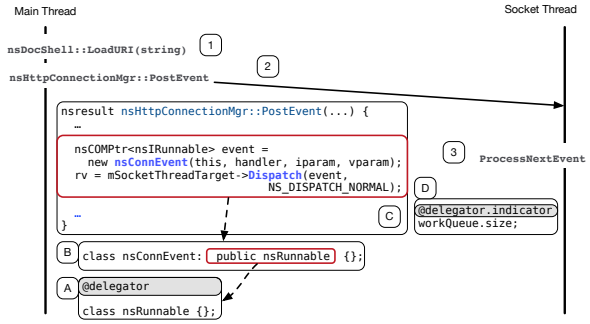

Figure 11: Firefox main thread posts events to the socket thread

The connection request data structure (in the SocketThread), the image data structure (in the image decoder thread), etc. are annotated as delegators. As such, when a connection request is created in the main thread, the request inherits the current tab/window id. When the request is used/handled in a SocketThread, the execution duration corresponding to the request belongs to the owner tab/window of the request. An example is shown in Figure 11. In Firefox, all delegator data structure classes have the same base class *nsRunnable*. As such, we only need to annotate *nsRunnable* as the delegator class (box A). When the main thread tries to load a new URI (step 1), it posts an *nsConnEvent* to the SocketThread (step 2) by calling the *PostEvent* method (box C). Since *nsConnEvent* is a sub-class of *nsRunnable* (box B), the delegator class, the newly created *nsConnEvent* inherits the tab/window id. The *nsRunnable* class provides a function *Run()*, which is implemented by its child classes to perform specific tasks. And each thread maintains its own work queue containing all such class instances. Thus the size of the worker queue is annotated as the indicator of the delegator. Whenever it changes, there may be a unit context switch. □

Annotation Miner. We develop an annotation miner to recommend unit and delegator data structures to annotate. The miner works as follows. The user provides a pair of executions to denote an intended unit task, one execution containing one unit and the other containing two units. Then, differential trace analysis is performed to prune data structures that are common in both traces and hence irrelevant to the unit (e.g., global data structures). The miner leverages the points-to relations between data structures to narrow down to the top data structures (i.e., those that are not pointed-to by other data structures). PageRank is further used to determine the significance of individual top data structures. A ranked list of data structures is returned to the user. Note that this mining stage is much less demanding than the training process in BEEP/ProTracer, which requires extracting code locations that induce low level memory dependencies. Since we focus on identifying high level data structures, which are covered by the provided inputs, completeness is not an issue for us in practice.

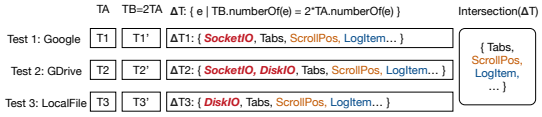

Figure 12: Annotation Miner

Next, we show how to mine the tab data structure in Firefox (Figure 12). We first use a pair of runs to visit the Google main page. `T1` has one tab and `T1'` has two tabs. ΔT shows the data structures in the trace differences. Note that there are data structures specific to the page content but irrelevant to the intended unit, such as `SocketIO`. To further prune those, we use another two pairs of executions that visit Google Drive and a local file, respectively. The miner then takes the intersection of the trace differences to prune out `SocketIO` and `DiskIO`. The resulting set contains the top level data structures and their supporting meta data structures (e.g., the `ScrollPos` data structure to support scrolling in a tab). The trace-based points-to analysis then filters out the low level supporting data structures. There may be multiple top level data structures remained, many not related to units (e.g., for logging). Hence in the last step, PageRank is used to rank the several top data structures. In our case, the tab data structure is correctly ranked the top.

3.3 Runtime

Unit Context. At runtime, each thread maintains a vector called the *unit context*. Each element of the vector denotes

the current unit instance for each unit type (or each perspective). Note that MPI allows partitioning an execution in different ways by annotating multiple unit data structures. If the user has annotated n unit data structures (with n indicators and n identifiers), there are n elements in the vector. Each time the indicator of a unit data structure is updated, the identifier of the data structure is copied to the corresponding vector element.

Delegation. MPI runtime provides a global hash map that is shared across all threads, called the *delegation table*. The delegation table projects a delegator data structure instance to a unit context vector value, denoting the membership of the delegator. Upon the creation/initialization of a delegator data structure instance, MPI inserts a key-value pair into the delegation table associating the delegator to the current unit context. Upon an update of the indicator of a delegator data structure (in a worker thread that handles the subtask represented by the delegator), the unit context of the current thread is set to the unit context of the delegator, which is looked up from the delegation table. Intuitively, it means the following execution belongs to the unit of the delegator until a different delegator is loaded to the indicator variable. The optimization of this process can be found in Appendix A. □ *Example.* Let us revisit the Firefox example in Figure 9. We want to attribute all subtasks to their corresponding tabs (shown in different colors). In Figure 11, we show a detailed workflow of the main thread posting the connection event to the socket thread. The main thread first calls the *LoadURI* method (step 1), which invokes the *PostEvent* method. Within *PostEvent* (box C), it creates an *nsConnEvnet* and posts it to the socket thread. Since data structure *nsRunnable* (box A) is annotated as a delegator and the HTTP connection request *nsConnEvent* (box B) is a subclass of *nsRunnable*, MPI propagates the current unit id in the main thread to the worker thread, namely, the socket thread. Specifically, the request is associated with the current unit context of the main thread in the delegation table. Inside the socket thread that receives and processes the request (i.e., step 3), loading the request from the task queue causes the change of the queue size indicating a possible unit context switch. As a result, the current unit context of the socket thread is set to that of the request, namely, tab1. With a chain of delegations, MPI is able to recognize all the tab1 subtasks performed by different threads, namely, all the red bars in Figure 9 belong to the same tab1 unit. □

3.4 Analysis

The analysis component of MPI is a pass in LLVM responsible for adding instrumentation to realize the runtime semantics mentioned earlier. It takes a program with the four kinds of annotations mentioned in §3.2, and produces an instrumented version of the program that emits additional syscall events denoting unit context switches and channel operations.

MPI needs to identify the following a few kinds of code locations: (1) all the updates (i.e., definitions) to indicator variables, including unit indicators and delegator indicators, to add instrumentation for unit context updates; (2) all the creation/initialization locations of delegator data structures to add instrumentation for the inheritance of unit context; (3) reads/writes of channel variables/fields to add instrumentation for channel event emission and redundancy detection; (4) all the system/library calls that may lead to system calls to add instrumentation for unit event emission and redundancy detection. We use a type based analysis to identify (2) and (3). For (4), we pre-define a list of library functions (e.g., libc functions) that may lead to system calls of interest and then scan the LLVM bitcode to identify all the system calls and the library calls on the list. Details are elided. A naive solution to (1) is to perform a walk-through of the LLVM bitcode to identify all definitions to indicator variables or to their aliases (using the default alias analysis in LLVM). However, this may lead to redundant instrumentation. Specifically, an indicator may be defined multiple times and there may not be any system calls (or library calls that can lead to system calls) in between. As such, the unit context switch instrumentations for those definitions are redundant.

```
1  /* Match a regexp against multiple lines. */
2  long im_regexec_multi(...) {
3    buf_T *save_curbuf = curbuf;
4    // initilize local variables
5    // switch to buffer "buf" to make vim_iswordc() work
6    curbuf = buf;
7    r = vim_regexec_both(NULL, col, tm);
8    curbuf = save_curbuf;
9    return r;
10 }
```

Figure 13: Instrumentation example (VIM, *op_yank* function)

□ *Example.* The function *im_regexec_multi()* in Figure 13 searches for a regular expression in Vim. The *indicator* variable is updated at line 6, and then again at line 8. The operations inside function *vim_regexec_both()* are all on memory. In other words, it does not make any system calls directly or indirectly. As such, the instrumentation for line 6 is redundant. □

The problem is formulated as a reaching-definition problem, which determines the set of definitions (of a variable) that can reach a program point. We say a definition of variable x can reach a program point ℓ, if x is not redefined along any paths from the definition to ℓ. In our context, we only instrument the definitions that can reach a system call or a library call that can lead to a system call. In Figure 13, the definition at line 6 cannot reach any point beyond line 8. Since line 7 does not denote any system call, line 6 is not instrumented. Appendix B discusses how to construct attack graphs from MPI logs.

4 Evaluation

In this section, we present the evaluation results including the annotation efforts needed, the runtime and space overheads of the prototype, and a number of attack cases to show the advantages of MPI compared to the event loop based partitioning technique in BEEP [43] and ProTracer [46]. For comprehensive comparison, we integrate both MPI and event loop based partitioning with three underlying provenance tracking systems, the Linux Audit system, ProTracer and LPM-HiFi.

4.1 Overhead

Space overhead: We measure the space overhead of MPI and compare it with the overhead of event loop based partitioning, on the aforementioned three provenance tracking systems. We measure the overhead of MPI and BEEP on Linux Audit and LPM-HiFi by comparing the logs generated by the original binaries and the instrumented binaries. ProTracer requires unit information to eliminate redundant system events (e.g., multiple reads of a file within a unit). Therefore, it needs to work with an execution partitioning scheme. We hence compare the ProTracer logs by BEEP and by MPI. Note that BEEP+ProTracer is equivalent to the original ProTracer system [46] and in MPI+ProTracer we retain the efficient runtime of the original ProTracer but replace the partitioning component with MPI. Since BEEP supports only one low-level perspective, we only annotate one perspective in MPI during comparison. The overhead of multiple perspectives is in Appendix D.

The results are shown in Table 1. The table contains the following information (column by column): 1) Application. 2) Perspective for partitioning. 3) Overhead of BEEP on Linux Audit, i.e., comparing the Linux Audit log sizes with and without BEEP. 4) Overhead of BEEP on LPM-HiFi with the raw log format. 5) Overhead of BEEP on LPM-HiFi with its Gzip enabled user space reporter tool. 6-8) Overhead of MPI on BEEP and LPM-HiFi. 9) Log size of BEEP on (original) ProTracer. 10) Log size of MPI on ProTracer. Note that Linux Audit and LPM-HiFi have different provenance collection mechanisms, i.e. system call interception for Linux Audit and LSM for LPM-HiFi. This leads to different space overheads. LPM-HiFi provides different user space reporters, and the Gzip enabled reporter has less space overhead.

Observe that for most programs our approach has less overhead on all the three platforms. For programs like document readers and video players, both approaches show very little overhead. These programs do not need to switch between different tasks frequently, which means they rarely trigger the instrumented code. Our approach shows significant better results for many programs like web browsers, P2P clients, HTTP and FTP programs including servers and clients due to a few reasons. Firstly, in these programs, the events handled by the event handling loop are at a very low level, whereas MPI can partition execution at a much higher level. Thus there are fewer unit context switches in our system, and multiple execution units in BEEP are grouped into one in our system without losing precision. For example in Apache, a remote HTTP request can lead to redirection, and the Apache server needs a few BEEP execution units to handle it. This triggers the instrumented code several times. But in MPI, multiple requests, including their redirections, of a same connection are grouped together. Thus, the instrumentation (for unit context switch) is triggered less frequently. Another reason is that we avoid meaningless execution units. For example in benchmark Transmission, BEEP execution units are based on time events, leading to many redundant units. This is avoided in MPI. Firefox has high overhead in both systems. When multiple tabs are opened, Firefox processes them in the background with threads. Since most of the requests involve network or file I/O, a lot of system/unit context switches are triggered, leading to the overhead. Despite this, the overhead of our system is about one third of that of BEEP. Note that there is another advantage of MPI that cannot be quantified –MPI does not require extensive training to detect low level memory dependencies. During our experiments, we had to add test inputs to the training sets of BEEP to ensure the provenance was not broken for a number of applications (e.g., Firefox).

We want to point out that with MPI, we can even reduce space overhead for the highly efficient ProTracer system and the reduction is substantial for a few cases. This is because MPI produces higher level execution units (compared to BEEP/ProTracer), leading to fewer units, more events in each unit and hence more redundancies eliminated by the ProTracer runtime. Also note that all the advantages of MPI over BEEP (e.g., without requiring extensive training and rich high-level semantics) are also advantages over ProTracer as the original ProTracer system relies on BEEP. We have ran MPI for 24 hours with a regular workload. The generated audit log has 680MB with 80MB by MPI. Details can be found in Appendix D.

Run time overhead: We measure the run time overhead caused by our instrumentation. For server programs, we use standard benchmarks. For example, for the Apache web server, we use the *ab* [1] benchmark. For programs that do not have standard test benchmarks, but support batch mode (e.g., Vim), we translate a number of typical use cases to test scripts to drive the executions. We preclude highly interactive programs.

For each application, we choose the same perspectives as the previous experiment, and the results are shown

Table 1: Space Overhead

Application	Level	BEEP Space Overhead			MPI Space Overhead			BEEP	MPI
		Linux Audit	LPM-HiFi (Raw - Gzip)		Linux Audit	LPM-HiFi (Raw - Gzip)		ProTracer (MB)	
Apache	HTTP Connection	15.38%	12.87%	0.64%	5.37%	3.75%	0.16%	22.12	20.08
Bash	Command	0.45%	0.34%	0.01%	0.41%	0.34%	0.01%	1.01	0.78
Evince	Document File	3.72%	4.98%	0.25%	0.04%	0.04%	0.00%	0.22	0.21
Firefox	Tab	42.16%	38.23%	1.01%	18.20%	13.24%	0.52%	593.23	228.54
Krusader	Command	26.54%	24.53%	0.09%	5.71%	4.89%	0.24%	2.31	2.31
Wget	Request	0.43%	0.33%	0.01%	0.42%	0.33%	0.01%	4.33	4.33
Most	File	0.05%	0.04%	0.00%	0.05%	0.04%	0.00%	1.78	1.78
MC	Command	0.93%	0.75%	0.01%	0.90%	0.75%	0.01%	3.43	1.89
Mplayer	Video File	0.04%	0.04%	0.00%	0.04%	0.04%	0.00%	0.34	0.34
MPV	Video File	0.09%	0.03%	0.00%	0.09%	0.03%	0.00%	0.58	0.58
Nano	File	0.29%	0.11%	0.01%	0.01%	0.01%	0.00%	8.23	2.46
Pine	Command	8.11%	6.09%	0.27%	7.28%	4.09%	0.13%	34.23	14.32
ProFTPd	FTP Connection	4.61%	3.45%	0.17%	2.11%	1.27%	0.06%	24.98	20.35
SKOD	FTP Connection	5.99%	3.89%	0.17%	2.68%	1.99%	0.10%	25.35	22.73
TinyHTTPd	HTTP Connection	8.94%	5.32%	0.32%	2.72%	1.08%	0.04%	43.24	37.48
Transmission	Torrent File	18.41%	18.33%	1.03%	0.12%	0.12%	0.01%	8.34	8.23
Vim	File	2.23%	2.32%	0.12%	0.13%	0.13%	0.01%	17.23	9.48
W3M	Tab	38.74%	30.45%	1.07%	24.67%	18.23%	0.19%	145.26	73.26
Xpdf	Document File	0.03%	0.07%	0.00%	0.03%	0.07%	0.00%	0.45	0.45
Yafc	FTP Connection	3.44%	1.78%	0.09%	2.60%	0.87%	0.04%	26.34	18.27

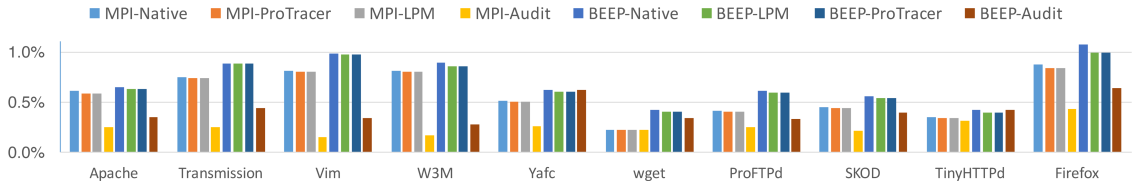

Figure 14: Run time overhead for each applications (Overhead percentage v.s. applications)

in Figure 14. For each program, we have eight bars. ①MPI-Native: the overhead of MPI without any provenance system over native run. ②MPI-ProTracer: the overhead of MPI over ProTracer. ③MPI-LPM: the overhead of MPI over LPM-HiFi. ④MPI-Audit: the overhead of MPI over Linux Audit. The other four bars denote the overhead of BEEP. As we can see from the graph, most applications have less than 1% run time overhead for all situations, which is acceptable. Comparing with BEEP, MPI shows less overhead in all cases. The low run time overhead is due to the following factors. Firstly, compared with the original program, the number of instrumented instructions is quite small. Secondly, most of the instructions are rarely triggered. Thirdly, our instrumentation mainly contains memory operations like comparing the newly assigned *identifier* value with the cached value.

4.2 Annotation Efforts

Table 2: Annotation Efforts

Application	LOC	Annotation				Inst
		ID	IND	Chann	DEL	
Vim	313,283	3	3	2	0	878
Yafc	22,823	2	3	0	1	111
Firefox	8,073,181	3	32	0	1	6,867
TuxPaint	41,682	2	2	0	0	121
Pine	353,665	2	2	2	0	746
Apache	168,801	2	2	0	1	2,437
MC	135,668	2	2	1	0	3,332
ProFTPd	307,050	3	3	0	1	4,905
Transmission	111,903	2	4	0	1	66
W3M	67,291	2	2	0	1	3,718

In this experiment, we measure the effectiveness of the annotation miner and the number of annotations eventually added. The annotation results are shown in Table 2. We only show some representative programs as the others have similar results. We present the applications in the first column, and their sizes (measured by SLOCCount [13]) in the second column. In the next four columns, we show the number of annotations needed for @identifier, @indicator, @channel, and @delegator. For each program, we provide two or more perspectives, as denoted by the number of @identifier annotations. In the last column, we show the instrumentation places automatically identified by our compiler pass. Less than 20% of these places were covered by our profiling runs. In other words, a training based method like that in BEEP/ProTracer would not be able to cover all these places.

Figure 15: Annotation miner results

To evaluate the annotation miner, we use the 20 programs in Table 1. For each program, we report the ranking of the unit/delegator data structures that we eventually choose to annotate. There are totally 52 of them. All the 6 delegator data structures are correctly ranked the top. That is because they are mainly used in worker threads,

which have relatively fewer data structures. For the 46 unit data structures that we eventually annotate, 36 of them are ranked at the first place, 8 at the second place, and the remaining 2 at the third place. Figure 15 shows the reported data structures for Vim, Firefox and HTTPd. Each plane denotes the results for a perspective. The highlighted data structures are the ones that we eventually choose to annotate. The reason why we do not always annotate the top data structures is that they are typically the *shadow* data structures of the real unit data structures. They usually store meta-data related to units, causing them to have higher ranks than the real unit data structure. With the help of the miner, we spent minutes to hours to finalize the annotations. We argue that such efforts are manageable. More importantly, they are one-time efforts.

4.3 Attack Investigation

To evaluate MPI's effectiveness in attack investigation, we apply it on 13 realistic attack cases used in previous works [32, 43, 44, 46]. The results show that MPI is able to correctly identify the root causes with very succinct causal graphs for all cases. Moreover, MPI generates fewer execution units using the perspectives in Table 1, when compared to BEEP/ProTracer. On average, the number of units generated by MPI is only 25% of that by BEEP/ProTracer. For attacks involving GUI programs (e.g., Firefox), the number is 8%, and in an extreme attack case involving Transmission, it is less than 1%. In terms of the generated attack graphs, MPI can reduce the number of nodes to 92% and the number of edges to 83% on average. Note that it is because these attacks have simple propagation paths such that the BEEP/ProTracer graphs are quite succinct. For complicated cases, MPI can reduce the graphs to 76%(nodes)/62%(edges). In addition, we evaluate it on a few other realistic attack cases. Next, we show one such case. Two more cases are presented in Appendix C to demonstrate the advantages of MPI over BEEP/ProTracer in an insider threat and in tracking complex browsing behaviors in Firefox.

Case: FTP Data Leak. Exploiting system misconfiguration to acquire valuable sensitive information is a common attack vector [9, 14]. It is important to assess and control damages once the problem is noticed. In the following incident, an FTP administrator accidentally configured the root directory of many users to a folder containing classified files, and gave them read accesses. After noticing the problem, he shut down the server and then conducted investigation to figure out the significance of the potential information leak. In the duration of the misconfiguration, there are thousands of connections from a large number of users. The number of classified files is also large.

In Figure 16, we show a number of possible investigation perspectives for the FTP server application. Event

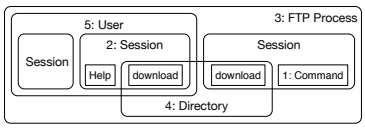

Figure 16: FTP server partitioning perspectives

loop based partitioning techniques are based on each command or user request (box 1), and traditional auditing approaches are based on the whole process (box 3). MPI provides choices that align better with the logical structures of the application, such as the session perspective (box 2), i.e., all the commands/requests from a session belong to a unit, the directory perspective (box 4), i.e., all the commands on a given directory are considered a unit, and the user perspective (box 5), i.e., all commands/requests from a user (not limited to an IP address) belong to a unit. Note that all FTP commands are associated with some file or directory as part of its context, and hence we can partition FTP execution based on this information.

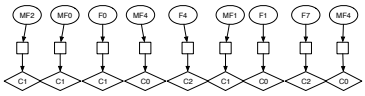

Figure 17: FTP server partitioned by BEEP

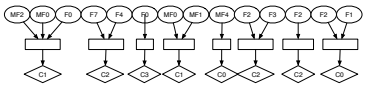

Figure 18: FTP server partitioned by each connection

Part of the BEEP graph is shown in Figure 17. Observe that each user command is captured as a unit. The simplified graph by MPI with connection based partitioning is shown in Figure 18, and user based partitioning in Figure 19. The connection perspective alleviates the inspector from going through the individual commands. The user perspective can aggregate all the behaviors from a specific user over multiple sessions so that the inspector can hold individual users for responsibilities. Note that a user can use various IP addresses to connect to the server. Without MPI, such semantic information cannot be exposed to the provenance tracking system. The number of nodes in the BEEP, connection (MPI), and user (MPI) graphs are 962, 224, and 78, respectively. We want to point out that the MPI graphs cannot be generated from the BEEP graph by post-processing because of the subtask delegation in this program, i.e., it is difficult to attribute a sub-task to the top level unit that it belongs to with only the low level semantic information in the BEEP graph.

5 Discussion

Similar to many existing works [23, 43, 44, 46, 55], MPI trusts the Linux kernel and the components associated with the audit logging system. Attacks that can bypass the

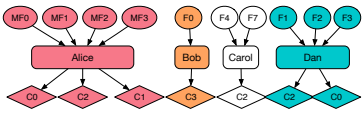

Figure 19: FTP server partitioned by users

security mechanisms of these systems may cause problems for MPI. Moreover, attacks that target the underlying audit system, such as audit log blurring and log filling, may inject noise to logs, making log inspection difficult. As our system is built on top of existing provenance and operating systems, MPI leverages existing features provided by these systems to mitigate some of the problems. For example, operating systems like Ubuntu now leverages Ubuntu Software Center to deliver trustworthy software which can be used to protect the MPI binaries for benign software. Provenance systems like Hi-Fi uses reference monitor guarantees to protect audit logs, and LPM provides a general framework for trustworthy provenance collection. We argue these are orthogonal challenges to all existing provenance tracking techniques and a complete solution to all these challenges is not the focus of our paper. Instead, the emphasis of MPI is to address dependence explosion caused by long running processes with accuracy and flexibility.

MPI is essentially an add-on service to the OS-level provenance collection system (e.g. the Linux Audit system, LPM-HiFi, and ProTracer). System calls can be too coarse-grained. Fine-grained events, such as library calls or even instruction level dependencies, may need to be captured for some sophisticated attacks. We argue that the multiple perspective partitioning enabled by MPI is orthogonal. It is independent of the granularity of the events captured by the underlying provenance system. It can be easily integrated with systems of various granularities.

MPI requires program source code. We believe that the semantic information needed to enable multiple perspective partitioning is difficult to acquire through binary analysis for complex programs such as Firefox. If it is necessary to partition the execution of a binary, training and event loop based approaches such as BEEP could be used together with MPI. In the worst cases, MPI treats the entire process execution as a unit. Note that this approximation is only problematic for long running processes. Many malware executables are likely not long running such that treating a whole process as a unit does not introduce a lot of bogus dependencies. Also note that such approximation does not miss provenance so that the attack path is still captured. It is just that more efforts may be needed to go through the causal graph.

MPI relies on source code annotations, which are widely used in practice. Windows developers explicitly plant logging commands in their software source code to customize ETW auditing. Both GCC and LLVM provide advanced language features [4, 6, 7] that are triggered by annotations. For example, Firefox has 926 different types of annotations. The stack-only class annotation "NS_-STACK_CLASS" has 406 uses through out the code base. In contrast, we only introduce 36 annotations (of 4 types) in Firefox. As MPI is based on source code level annotation and compiler instrumentation, it cannot find units within dynamic code. However, in practice, we find that unit boundaries mostly lie in static code. For example, JavaScript code can be grouped into different tabs. Thus, dynamic code can be attributed to tab units.

6 Related work

Many approaches have been proposed for system level provenance tracking. Detailed comparison of MPI with existing audit systems [10, 31, 43–45] can be found in §2. Another important approach is to monitor the internal kernel objects (e.g., the file system [27, 49, 50, 59–61, 69], or LSM objects [23, 32, 55]) to track lineages. The capabilities of these techniques are similar to those of the audit systems. Thus MPI is complementary to such systems. For example in §4, we showed the integration of MPI and LPM-HiFi. System wide record-and-replay techniques [30, 37–39] can also track provenance. These systems record the inputs for all programs, and replay the whole system execution when needed. Such systems require deterministic record-and-replay techniques, which are open research problems, and cause more space overhead. Whole system tainting [28, 35, 52, 68] is another method of tracking provenance. By tainting all inputs to a system and tracking their propagation, such systems can record the needed provenance data. These techniques need to deal with the granularity problem as the taint set may be explosive for a long living system objects/subjects. MPI can be applied to such systems to overcome the dependency explosion problem and enable multiple perspective inspection.

In [48], researchers propose to develop provenance aware applications. Muniswamy-Reddy *et. al.* [49] provide a library with provenance tracking APIs so that programmers can develop provenance aware applications. Such an approach relies on the programmers to intensively modify their code to leverage the APIs. In contrast, MPI aims to address the partitioning problem. Provenance tracking is through the underlying audit system.

Many works [22, 27, 44, 67] are proposed to reduce the space overhead of provenance tracking based on reachability analysis, Mandatory Access Control (MAC) policies and so on. Provenance visualization [25, 26, 47, 53, 57] and graph compression [34, 54, 58, 63–65] are also proposed to correlate events and reduce graph size to facilitate investigation. These approaches work on generated graphs to compress them for better visualization. As such, they are complementary to MPI, and

can be directly applied to MPI, its provenance logs and graphs. Researchers proposed many machine learning methods [21, 24, 33, 40, 41, 51, 66] to investigate provenance data to find abnormal behaviors. We envision that the multiple perspectives provided by MPI may substantially improve their effectiveness.

7 Conclusion

Execution partitioning is important for addressing dependency explosion in audit logging. However, existing techniques are event loop based. They generate too many small units, require training to detect dependencies across units, and lack information about high level logic tasks. We propose MPI, a technique that partitions based on high level tasks. It allows the user to annotate the data structures corresponding to these task, and leverages compiler to instrument operations of the data structures in order to capture unit context switches and delegations. We implemented a prototype and evaluated it on three existing systems: Linux Audit, ProTracer and LPM-HiFi. The results show that MPI generates much smaller graphs with lower overhead comparing to the state-of-the-art, and avoids broken provenance due to incomplete training.

8 Acknowledgment

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668 , 1320444, and 1320306, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Apache benchmark. https://goo.gl/L7bGOK.

[2] The browser exploitation framework. http://beefproject.com/.

[3] Chinese hacker arrested for leaking 6 million logins. https://goo.gl/AO2Qlz.

[4] Clang language extensions. https://goo.gl/UpniZC.

[5] Event tracing for windows (etw). http://msdn.microsoft.com/en-us/library/windows/desktop/aa363668(v=vs.85).aspx.

[6] Extensions to the c++ language. https://goo.gl/pn19Np.

[7] Extensions to the c language family. https://goo.gl/evrruW.

[8] Github hacked, millions of projects at risk of being modified or deleted. https://goo.gl/EdguGO.

[9] Leaked data. https://haveibeenpwned.com/.

[10] Linux audit subsystem. https://goo.gl/WSwnJB.

[11] Many watering holes, targets in hacks that netted facebook, twitter and apple. https://goo.gl/NIg2Va.

[12] More details on "operation aurora". https://goo.gl/p76ovs.

[13] Sloccount. http://www.dwheeler.com/sloccount/.

[14] The sony hack. https://goo.gl/B4G7Pl.

[15] Tuxpaint. www.tuxpaint.org.

[16] Ubsi. https://github.com/kyuhlee/UBSI.

[17] Vim document: windows. https://goo.gl/Lqp9Gb.

[18] Watering hole attack. https://goo.gl/AcN0dv.

[19] Watering hole attack. https://goo.gl/aw1t9l.

[20] Windows event log. https://msdn.microsoft.com/en-us/library/windows/desktop/aa385780(v=vs.85).aspx.

[21] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., AND RIECK, K. Drebin: Effective and explainable detection of android malware in your pocket. NDSS'14.

[22] BATES, A., BUTLER, K. R., AND MOYER, T. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. TaPP '15.

[23] BATES, A., TIAN, D. J., BUTLER, K. R., AND MOYER, T. Trustworthy whole-system provenance for the linux kernel. Usenix Security'15.

[24] BESCHASTNIKH, I., BRUN, Y., SCHNEIDER, S., SLOAN, M., AND ERNST, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. ESEC/FSE'11.

[25] BEVAN, C. F., AND YOUNG, R. M. Planning Attack Graphs. In *ACSAC* (2011).

[26] BORKIN, M. A., YEH, C. S., BOYD, M., MACKO, P., GAJOS, K. Z., SELTZER, M., AND PFISTER, H. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (Dec. 2013), 2476–2485.

[27] BRAUN, U., GARFINKEL, S., HOLLAND, D. A., MUNISWAMY-REDDY, K.-K., AND SELTZER, M. I. Issues in automatic provenance collection. In *Provenance and annotation of data*.

[28] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding data lifetime via whole system simulation. USENIX SSYM'04.

[29] CUMMINGS, A., LEWELLEN, T., MCINTIRE, D., MOORE, A. P., AND TRZECIAK, R. Insider threat study: Illicit cyber activity involving fraud in the us financial services sector. Tech. rep., DTIC Document, 2012.

[30] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 525–540.

[31] GEHANI, A., AND TARIQ, D. Spade: Support for provenance auditing in distributed environments. Middleware '12.

[32] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The taser intrusion recovery system. SOSP '05.

[33] GU, Z., PEI, K., WANG, Q., SI, L., ZHANG, X., AND XU, D. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2015), pp. 57–68.

[34] GUO, Z., ZHOU, D., LIN, H., YANG, M., LONG, F., DENG, C., LIU, C., AND ZHOU, L. G^2: A graph processing system for diagnosing distributed systems. USENIX ATC'11.

[35] JIANG, X., WALTERS, A., XU, D., SPAFFORD, E. H., BUCHHOLZ, F., AND WANG, Y.-M. Provenance-aware tracing of-worm break-in and contaminations: A process coloring approach. ICDCS '06, IEEE.

[36] KEENEY, M., KOWALSKI, E., CAPPELLI, D., MOORE, A., SHIMEALL, T., ROGERS, S., ET AL. Insider threat study: Computer system sabotage in critical infrastructure sectors. *US Secret Service and CERT Coordination Center/SEI* (2005).

[37] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F.

Intrusion recovery using selective re-execution. OSDI'10.

[38] KING, S. T., AND CHEN, P. M. Backtracking intrusions. SOSP '03.

[39] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. NDSS '05.

[40] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X.-Y., AND WANG, X. Effective and efficient malware detection at the end host. USENIX'09.

[41] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. CCS '11, ACM.

[42] KOWALSKI, E., CONWAY, T., KEVERLINE, S., WILLIAMS, M., CAPPELLI, D., WILLKE, B., AND MOORE, A. Insider threat study: Illicit cyber activity in the government sector. *US Department of Homeland Security, US Secret Service, CERT, and the Software Engineering Institute (Carnegie Mellon University), Tech. Rep* (2008).

[43] LEE, K. H., ZHANG, X., AND XU, D. High accuracy attack provenance via binary-based execution partition. NDSS '13.

[44] LEE, K. H., ZHANG, X., AND XU, D. Loggc: garbage collecting audit log. CCS '13.

[45] MA, S., LEE, K. H., KIM, C. H., RHEE, J., ZHANG, X., AND XU, D. Accurate, low cost and instrumentation-free security audit logging for windows. ACSAC '15.

[46] MA, S., ZHANG, X., AND XU, D. Protracer: towards practical provenance tracing by alternating between logging and tainting. NDSS '16.

[47] MEHTA, V., BARTZIS, C., ZHU, H., CLARKE, E., AND WING, J. Ranking Attack Graphs. *9th International Symposium on Recent Advances in Intrusion Detection (RAID'06) 4219* (2006), 127–144.

[48] MILES, S., GROTH, P., MUNROE, S., AND MOREAU, L. Prime: A methodology for developing provenance-aware applications. *ACM Transactions on Software Engineering and Methodology (TOSEM) 20*, 3 (2011), 8.

[49] MUNISWAMY-REDDY, K.-K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. USENIX ATC'09.

[50] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. Usenix ATC '06.

[51] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured comparative analysis of systems logs to diagnose performance problems. NSDI'12.

[52] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. NDSS'05.

[53] OU, X., BOYER, W. F., AND MCQUEEN, M. A. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06* (2006), p. 336.

[54] OU, X., GOVINDAVAJHALA, S., AND APPEL, A. MulVAL: A logic-based network security analyzer. *14th USENIX Security ...*, August (2005), 8.

[55] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: Collecting high-fidelity whole-system provenance. ACSAC '12.

[56] RANDAZZO, M. R., KEENEY, M., KOWALSKI, E., CAPPELLI, D., AND MOORE, A. Insider threat study: Illicit cyber activity in the banking and finance sector. Tech. rep., DTIC Document, 2005.

[57] SAWILLA, R. E., AND OU, X. Identifying critical attack assets in dependency attack graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2008), vol. 5283 LNCS, pp. 18–34.

[58] SHEYNER, O., HAINES, J., JHA, S., LIPPMANN, R., AND WING, J. M. Automated generation and analysis of attack graphs. In *Proceedings - IEEE Symposium on Security and Privacy* (2002), vol. 2002-January, pp. 273–284.

[59] SITARAMAN, S., AND VENKATESAN, S. Forensic analysis of file system intrusions using improved backtracking. IWIA '05.

[60] SUNDARARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. Usenix Security'08.

[61] TIAN, D. J., BATES, A., BUTLER, K. R., AND RANGASWAMI, R. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 242–253.

[62] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 17–31.

[63] XIE, Y., FENG, D., TAN, Z., CHEN, L., MUNISWAMY-REDDY, K.-K., LI, Y., AND LONG, D. D. A hybrid approach for efficient provenance storage. CIKM '12.

[64] XIE, Y., MUNISWAMY-REDDY, K.-K., FENG, D., LI, Y., AND LONG, D. D. Evaluation of a hybrid approach for efficient provenance storage. *ACM Transactions on Storage (TOS) 9*, 4 (2013), 14.

[65] XIE, Y., MUNISWAMY-REDDY, K.-K., LONG, D. D., AMER, A., FENG, D., AND TAN, Z. Compressing provenance graphs. TaPP'11.

[66] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. SOSP'09.

[67] XU, Z., WU, Z., LI, Z., JEE, K., RHEE, J., XIAO, X., XU, F., WANG, H., AND JIANG, G. High fidelity data reduction for big data security dependency analysis. CCS '16.

[68] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. OSDI '06.

[69] ZHU, N., AND CHIUEH, T.-C. Design, implementation, and evaluation of repairable file service. DSN'13.

A Run Time Optimization

MPI emits special syscall events to denote unit context switches, and channel reads/writes. During causal graph construction Appendix B, the unit context switch events are used to derive unit boundaries and the channel events are used to derive inter-unit dependencies. Note that channel operations are essentially memory reads and writes that need to be exposed as system events. Otherwise, they are invisible to MPI. Inter-unit communication through system resources such as files, sockets, and the system clipboard can be captured by the default underlying system event tracking module without the intervention of MPI.

A naive solution is to emit a unit context switch event upon any indicator update and a channel event upon any channel read/write. However in practice, we observe that (1) an indicator update may not imply the change

of the unit context and (2) even though the unit context changes, there may not be any system events that happen in between the two unit context switches. Both cases lead to redundant unit context switch events. Similarly, there are often multiple accesses to the same channel object within the same unit. These accesses must induce the same causality and hence cause redundancy. Since emitting an event entails a system call and hence a context switch, preventing redundant event emission is critical to the efficiency of MPI. We have two approaches to address this problem. One is through the static analysis (§3.4) and the other is runtime optimization. MPI does not emit any event upon an indicator update. Instead, it simply updates the current unit context (in memory), which has much lower overhead compared to a system call. Upon a regular system call (e.g., file read), it checks if the current unit context is the same as the previous context that was emitted. If not, it emits a unit context switch event right before the system call. Otherwise, it does not emit. Similarly, upon a channel operation, MPI checks if a channel operation by the same unit was logged before. If so, it avoids logging the channel operation.

B Causal Graph Construction

In this section, we discuss the causal graph construction algorithms for backward tracking starting from a symptom event and forward tracking starting from a root cause event. Algorithm 1 shows how to generate the *backward tracking* causal graph for a specific perspective with a given log file and a symptom event. Generating the graphs for all perspectives only requires an easy extension.

We use an $objs$ set to represent the system objects, subjects, and channels between units that are directly or indirectly related to the symptom event. The overall procedure of the algorithm is to traverse the log in a reverse order to populate the set and identifies events causally related to the symptom by correlating to some entity in $objs$. At line 1, the algorithm initializes the set to contain the system object accessed by the symptom event and the system subject (i.e., the process of the event). It also marks the current unit as correlated to the symptom (line 2). Then it traverses all the events in the log file in a reverse order, starting from the symptom event (lines 3-17). If the current event e is not a unit context switch event, the algorithm saves it in a temporary list of events for the current unit (line 4-5). If e updates an object (e.g., file and pipe) or spawns a subject (i.e., process) that was identified as related to the symptom (and hence in the $objs$ set), a flag is set to indicate that the current unit is correlated (lines 6-7). If e is a unit context switch, the algorithm further tests if e switches to a unit in the given perspective. If not, the switch event is irrelevant and simply skipped (lines 9-10). Otherwise, it indicates a unit boundary of our interest. The algorithm checks the flag to see if the current unit is causally related to the symptom (lines 11-12). If so, it adds all the events in the current unit to the result graph. It also updates $objs$ with all the objects read by any event in the current unit and all the subjects spawned in the unit (lines 13-14). The temporary event list and the flag are then reset (lines 15-16). Note that when the events are added to the graph, nodes are created and further connected to existing nodes in the graph by the dependencies implied by the events. For example, a file read event entails connecting to the (previously created) file node. Details are elided for brevity.

Algorithm 1 Backward Causal Graph Construction

Input:	L - the event log l - unit type (i.e., perspective) given in the @*indicator* annotation e_s - symptom event
Output:	G_l - the generated causal graph for perspective l
Variable:	$objs$ - system objects/subjects relevant to e_s s_e, pid_e - the system object/pid of event e $bUnit$ - if the current unit causally related with e_s $eventUnit[pid]$ - the events in the current unit of process pid

```
1:  objs ← { pid_es, s_es }
2:  bUnit ← true
3:  for each event e ∈ L in reverse order, starting from e_s do
4:      if e is not a unit context switch event then
5:          eventUnit[pid].add(e)
6:      if e updates any object or subject in objs then
7:          bUnit ← true
8:      if e is a unit context switch event then
9:          if e does not switch to a l unit then
10:             continue
11:         else
12:             if bUnit then
13:                 add events in eventUnit[pid_e] to G_l
14:                 add accessed objects/subjects in eventUnit[pid_e] to
    objs
15:         eventUnit[pid_e] ← ∅
16:         bUnit ← false
17: return G_l
```

□ *Example.* Figure 20 shows an example of constructing the backward causal graph. The simplified log entries are shown on the left while the generated graph is shown on the right. The graph is also annotated with events to explain why nodes/edges are introduced. The algorithm generates the graph starting from the symptom event at line 8, which is a write event to the socket *a.a.a.a*. It traverses back and reaches line 7, which is a unit context switch (UCX) event whose *indicator* is 5 and the *identifier* value is 7. Two nodes are hence created representing that a process (node) wrote to a socket (node) whose value is *a.a.a.a*. Going backward, the algorithm further identifies another unit represented in lines 4-6 with the *indicator* value 5 and the *identifier* value 3. This is a different unit

instance of the same type and it has no causal relation with the object set that currently contains the socket object and the process. Therefore, all the events in this unit are dropped. The algorithm continues to traverse backward and encounter another unit in lines 1-3. Line 2 indicates that it reads file *index.html*, so the subgraph for lines 1-3 is file *index.html* being read by the process. Note that the value of *identifier* indicates lines 1-3 and lines 7-8 belong to the same unit (instance), which means that the application is working on the same task. Hence, the global causal graph is updated by joining the two subgraphs. The result graph is shown on the right hand side.

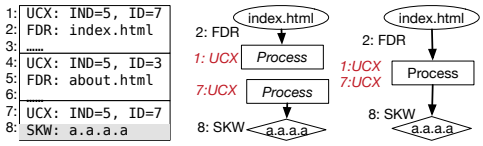

Figure 20: An example of constructing backward causal graph. (*UCX* is short for *Unit Context Switch*, *FDR* is short for *File Descriptor Read*, and *SKW* is short for *Socket Write*.)

The forward graph construction algorithm is similar and hence omitted.

Essence of MPI and Memory Dependencies. From the graph construction Algorithm 1, one can observe that all the events in a unit are considered correlated. If there is a single event (within a unit) that has any direct/indirect dependency with the symptom, all the events in the unit are added to the graph and all the objects/subjects accessed by the unit are considered correlated. As such, MPI does not need to track any fine-grained (memory) dependencies *within a unit*. Dependencies across units are either captured through system level dependencies (e.g., file/socket reads and writes) or explicitly indicated by the user through the channel annotation.

C Case Studies

Case: Insider Threat. In attacks such as watering hole and phishing emails, the adversaries apply external influences and wait for the employees to make mistakes. However, it is also very common that attacks are launched from inside the enterprise (e.g., by malicious or former employees). In fact, a large number of such cases had been reported [29, 36, 42, 56]. Next, we simulate such an attack.

A computer game development company noticed that the graphical design of a to-be-announced game was leaked on an online gaming forum. The company started investigation, trying to understand how this design was leaked and who should be held responsible. The investigator first conducted forward tracking from the design file but found that the file was neither sent outside by any email nor copied by any employee to their own devices. She further suspected that some old version of the file was leaked instead of the current version. Even though the old versions of the design file did not explicitly exist any more, the provenance of the file was tracked by the audit system.

She first conducted backward tracking to disclose all the past versions (with the name "p_v" plus the version number) and then forward tracking to see how these versions were propagated/used. Assume that she used BEEP first. She quickly noticed a number of problems in the BEEP graph that makes manual inspection difficult.

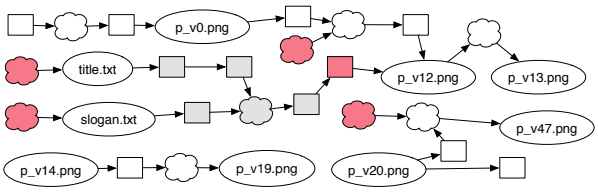

Figure 21: Event handling loop based solution

The resulting graphs by BEEP are shown in Figure 21. White boxes represent units for TuxPaint [15], gray boxes are for the editor, Vim, and red boxes are for other apps. *First of all*, the graph is very large (containing 1832 nodes). This is because many people had contributed to the file in the past using TuxPaint, a graph drawing tool. There were a lot of interactions (e.g., copy & paste) among multiple image files, some of which were from Internet. The various historic versions of the design file were propagated to other places. *Second*, there are many "empty" execution units, which are execution units just have boundary events. This is because many operations in UI intensive program TuxPaint have no real effects on the provenance. These operations include, but are not limited to, switching painting tools (frequently), clicking menu bars and so on. *Third*, she found that most execution units for TuxPaint only have memory dependency events. This is because TuxPaint stores the image buffers in memory, and flushes them to disk only when the user clicks the save button. In the editing units (e.g., choosing tools and drawing figures), TuxPaint only operates on the image buffers. These units are only connected by memory dependency and do not invoke any system calls. However, these units are important as they are responsible for chaining up the important behaviors.

After inspecting such a large graph, the inspector still could not spot any suspicious behavior. The reason is that there are broken links in the graph such that some updates to the design file are missing from the graph. Specifically, some of the editing actions were not in the BEEP training set such that the corresponding memory dependencies are not visible, leading to broken provenance, e.g., "p_v14.png" and "p_v20.png".

The inspector switched to MPI. She used individual image files as the perspective. The resulting (simplified)

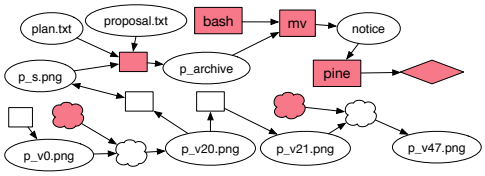

Figure 22: MPI solution

graph is in Figure 22. Now each white box represents all the editing operations on a single file. It can be clearly seen that a version of the design file, "p_v20.png", was read by a TuxPaint unit that operated on file "p_s.png", which was later archived with a number of text files. The archive was renamed and sent through an email. The link from the design file to file "p_s.png" was missed by BEEP because the attacker opened the design file, conducted a few editing actions whose memory dependencies are missed by BEEP such that the later *save-as* unit is disconnected from the file read unit. Note that all these actions are individual units in BEEP that need to be chained up by memory dependencies, whereas they belong to the same unit in MPI. Overall, the MPI graph is precise, much smaller (152 nodes) and cleaner. We also want to point out that a graph similar to the MPI graph cannot be generated by post-processing the BEEP graph as the missing links cannot be inferred and it is difficult to determine which low-level nodes belong to an image file.

Figure 23: Firefox browsing history of page perspective

Case: Complex Browsing Behavior in Firefox. In this case study, we show how MPI precisely captures the causality of complex browsing behavior of Firefox. During browsing, the user first opened Bing from the bookmark bar, and searched a key word, and then used different ways to open new pages including clicking links, choosing "open page in a new tab/window" in the right-click menu, going back to the previous page, and opening new pages from Javascript code automatically. In the end, the user downloaded a PDF file. We collected the log with the page perspective and generated a causal graph by conducting backward traversal starting from the PDF file. The graph is shown in Figure 23. Observe that the entire browsing history is precisely captured by the graph, including visiting the LinkedIn page from the search result page and then going back to the search result page. In contrast, the BEEP's graph only includes the page hosting the PDF file, missing all the other pages along the causal chain, due to missing memory dependencies.

D Additional Experimental Results

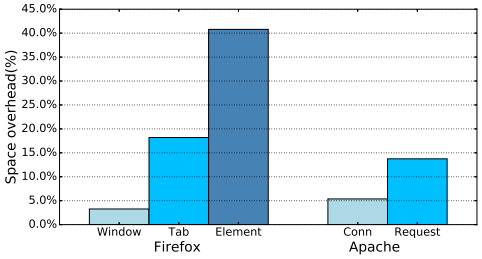

Figure 24: Overhead for applications with different partitioning

We also conduct experiments to measure space overhead for the same application with different partitioning choices, and the results are shown in Figure 24. We select two programs, Firefox and Apache. For Firefox, we choose three different ways to instrument: windows, i.e., a unit for a top level residence window for tabs (note that multiple windows may be driven by the same Firefox process internally); tabs and elements (inside a page). We do not show the numbers for each web site instance, because the instrumentations are similar to those of tabs, and the only difference lies in the expressions used in the @*identifier* annotation (see §3). For Apache, we use two ways to instrument: each connection (each client instance), and each request. The results show that with different levels of instrumentation, the overhead is significantly different. Instrumenting the applications at a higher level causes less overhead. For both cases, a lower level suggests 2-3 times overhead increase.

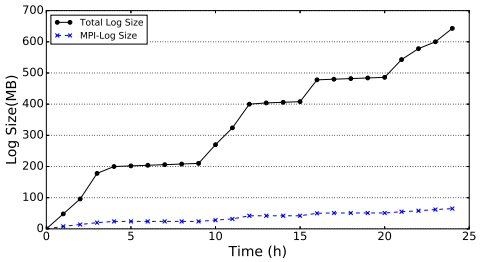

Figure 25: Overhead for a whole day

The last space overhead experiment we did is to run the instrumented applications on our machine for a whole day with Linux audit system enabled and measure the events generated by MPI. The workload includes regular uses such as web surfing, checking and responding emails. The result is shown in Figure 25. The black solid line shows the log size generated by the Linux audit system, and the dashed blue line shows the log size generated by MPI. From the graph, we can see that the log size generated by the Linux audit is more than 600 MB while our instrumentation issues less than 80 MB.