autoMPI: Automated Multiple Perspective Attack Investigation with Semantics Aware Execution Partitioning

Mohannad Alhanahnah*‖, Shiqing Ma†, Ashish Gehani‡, Gabriela F. Ciocarlie§, Vinod Yegneswaran‡, Somesh Jha*, Xiangyu Zhang¶
*University of Wisconsin-Madison, USA †Rutgers University, USA ‡SRI International, USA
§The University of Texas at San Antonio, USA
¶Purdue University, USA
‖Corresponding author: mohannad@cs.wisc.edu

Abstract

Multiple Perspective attack Investigation (MPI) is a technique to partition application dependencies based on high-level semantics. It facilitates provenance analysis by generating succinct causal graphs. It involves an annotation process that identifies variables and data structures corresponding to the partitions and the communication channels between them. Though the amount of annotation is small, this process requires a detailed understanding of the source code. In this work, autoMPI, we extend the capability of MPI by automating the identifying annotation requirements. We leverage a hybrid analysis approach, performing a differential analysis based on crafted inputs. Static analysis is conducted to identify the annotation sites within the application code afterward automatically. Our evaluation shows the proposed approach can significantly facilitate the annotation process. It correctly identifies all required annotation sites within an average 16 seconds analysis time for the majority of analyzed programs with average precision and recall 72.5% and 100%, respectively.

Index Terms—Static analysis, Dynamic analysis, Annotation, Provenance,

1 Introduction

Provenance tracking is critical for attack investigation, especially for advanced persistent threats (APTs) 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 protecting target systems. Most existing techniques [1–5] 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()*; and 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, which may have dependencies with many objects and other processes during their 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 partitioning execution into units so that only the events within a unit are considered causally related [2, 6]. For instance, the execution of a long-running server is partitioned into individual units, each handling a request. Although existing execution partitioning-based systems such as BEEP [6] and ProTracer [2] have demonstrated great potential, they partitioned execution based on event handling loops. 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 used 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 subset 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 the 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).

In previous work, we proposed MPI [7] [1], which takes the

This work was mostly performed while Gabriela F. Ciocarlie was at SRI International.

1. MPI is short for "Multiple Perspective attack Investigation"

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 are exposed to the underlying provenance tracking systems. MPI allows annotating multiple task/unit structures simultaneously such that the forensic analyst can inspect an execution from multiple perspectives (e.g., tab and domain perspectives for Firefox). MPI optimizes the provenance analysis by generating succinct causal graphs. This optimization is achieved by annotating the data structures required to be tracked. However, the annotation miner in MPI requires manual effort to identify necessary data structures. In this paper, we extend MPI along several important dimensions by proposing AUTOMPI that provides an automated, end-to-end provenance approach, by alleviating the manual effort needed to identify the necessary data structures to be annotated.

This paper describes several new non-trivial extensions to the preliminary version of our work described in [7] by: (1) proposing algorithms to automatically identify the variables required for the annotation process; (2) designing and implementing a hybrid analysis approach for automating the process of the *automated annotations identifier* component of AUTOMPI and thus reducing developer burden; (3) evaluating our approach using real-world applications and comparing its performance against the previous manual annotation approach; (4) making the prototype and associated artifacts available to the research community.

Specifically, this paper makes the following contributions:

- *Automated Annotations Identifier.* We make MPI fully automated by empowering the miner component with the capability to automatically determine the data structures required for the annotation process and the corresponding program partitions that should be tracked.
- *Tool implementation.* We develop AUTOMPI as a stand-alone analysis tool. We make AUTOMPI research artifacts, including the tool and the experimental data [2], available to the research and education communities.
- *Experimental evaluation.* We present our experiences with a thorough evaluation of AUTOMPI, based on real-world programs. Our evaluation shows that the precision, recall, and F1 score of AUTOMPI are 72.5%, 100%, and 84%, respectively. The analysis time is 16 seconds for most of the analyzed programs.

2 Motivation

This section compares state-of-the-art provenance tools, discusses their limitations, and describes our approach to handling these limitations.

2.1 Limitations in Prior Work

Several provenance approaches were proposed by prior work [7–12]. Table 1 presents the merits and limitations of existing causality analysis approaches.

2. https://github.com/Mohannadcse/autoMPI

Table 1: Comparison of causality analysis approaches

Factors	MCI [8]	OmegaLog [9]	UISCOPE [10]	AUTOMPI
Instrumentation	No	No	No	Yes
Training	Yes	No	No	Limited [a]
Granularity	Coarse	Fine	Coarse	Fine
Source Code	No	No	No	Yes
Handling threaded programs	Yes	No	No	Yes
GUI apps	No	Yes	Yes	Yes
OS/Kernel Modification	No	Yes	No	No

[a] minimal training is required to provide sufficient

Training involves performing dynamic analysis, which is used to identify dataflow dependencies, with the resulting coverage completion challenge, whereas AUTOMPI leverages dynamic analysis to identify relevant data structures, requiring a significantly smaller number of traces (minimal training is sufficient). The training process requires instrumenting the targeted programs, while on the other hand, approaches like Omegalog [9] does not require instrumentation and operates on top of the binary code. Specifically, Omegalog captures the logs at the level of an event-handling loop. However, Omegalog requires modifying the Linux kernel (custom kernel module) to intercept write system calls in order to add timestamps and PID. Modifying the kernel introduces additional limitations on the applicability of such techniques. Similarly to BEEP [6] and ProTracer [2], MCI [8] relies on a training phase, but MCI does not require instrumentation. Therefore, MCI may struggle with out-of-order events due to the presence of concurrent or cooperating applications during training runs. Similar to Omegalog, MCI requires modifying the kernel.
Previous work has semantic gaps because they lack knowledge of application-specific behaviors crucial for attack reconstruction. This semantic gap issue has been addressed through instrumentation-based techniques, which either statically or dynamically instrument function calls in the application to disclose function names, arguments, and return values. However, such instrumentation-based systems suffer from several limitations: (1) developers need to specify which functions to instrument; (2) the logging information is captured on a per-application basis and thus cannot be used to connect information flow between different applications; and (3) high-level semantic events may not always be effectively captured at the function call level. Accordingly, there is a pressing need to perform fine-grained analysis.

2.2 Our Goal

The overarching idea of this paper is that high-level tasks are reflected as data structures. AUTOMPI allows the user to automatically annotate the data structures that correspond to such tasks (Section 4.3). It then uses 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, AUTOMPI allows multiple data structures to be annotated, each denoting an independent perspective. To automate the annotation process, AUTOMPI provides a hybrid analysis approach that can automatically identify the critical data structure.
Note that allowing developers/users to insert logging-related

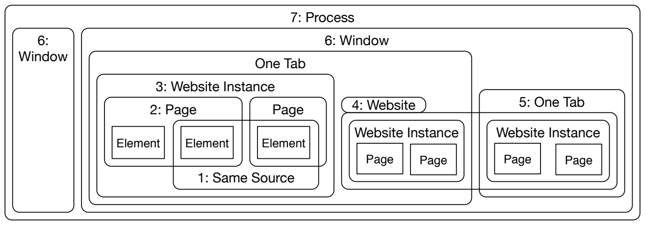


Figure 1: Firefox Partitioning perspectives

annotations/commands to software source code is a practical approach for system auditing. The Windows auditing system, *Event Tracing for Windows* (ETW), requires developers to explicitly plan customized events to their software before deployment [13, 14]. These commands generate system events at runtime. In our design, we only require the developer to annotate (a few) task-oriented data structures; AUTOMPI automatically instruments a much larger number of code places based on the annotations.

Figure 1 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 the exposure of the high-level semantics of the application to the underlying provenance tracking system.*

2.3 autoMPI Assumptions.

This section introduces the main assumptions of AUTOMPI. We also briefly describe the requirements of our approach.

AUTOMPI considers C/C++ programs whether command-line programs (i.e., nano) or programs that support a graphical interface (i.e., Firefox); therefore, all leveraged tools and implementations are built to handle this class of programs.

Our approach applies a set of program analysis techniques including pointer analysis to identify the data structures that should be annotated. Section 2.2 describes the basic annotations and threading annotations required by AUTOMPI. But in this work, we focus on automating the identification of data structures corresponding to the basic annotations (i.e., indicator and identifier variables).

3 autoMPI

This section introduces the workflow of AUTOMPI and describes briefly its major components. The overall AUTOMPI process of analysis and instrumentation is shown in Figure 2. The required annotations are automatically identified through the *automated annotations identifier* component, which is essentially a data structure profiler. The *annotator* component then, implemented as an LLVM pass, takes the annotation candidates and instruments the program (e.g., data structure accesses denoting unit boundaries). The *automated annotations identifier* comprises the following steps:

- Dynamic Trace Analysis: identifies high-level structures (i.e., candidate function names), where potential annotations are hosted and corresponding to the target perspective.
- Static Liveness Analysis: captures low-level variables (i.e., indicator and channel variables) corresponding to the required annotations.
- Heuristic Analysis: narrows down the number of candidate low-level variables.

Section 4.3 provides a detailed discussion of these steps. The next section describes various annotations that can be used to represent target perspectives.

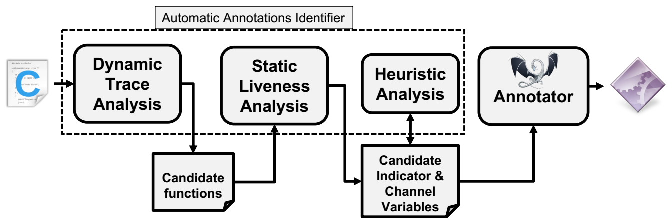


Figure 2: AUTOMPI workflow.

4 Annotations Overview

This section introduces the set of annotations supported by our approach.

4.1 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*. 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 currently 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 (into 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 the 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 with the corresponding indicator, we allow an indicator id to be provided as part of the identifier annotation.
③ *@channel* annotates the variables/fields that serve as “IPC channels” between two *unit data structure* instances (akin to *pipes*). It contains a unique id number, and parameters

```
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;
27
```

Figure 3: Vim data structure and our annotation

indicating which fields store the data inducing inter-unit dependencies.

☐ *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 [15]. 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 3 shows a piece of code that demonstrates our annotations. Vim uses the variable *curbuf* to represent the currently 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. Hence, we can 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 clipboard 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 in 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 clipboard. Our system also supports inter- or intra-process operations through the *system* clipboard by tracking system-level events.

4.2 Threading Annotations

This section introduces the annotations provided by the automated annotator to support threading programs.

Threading Support. 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 into 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 [2, 6], 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.

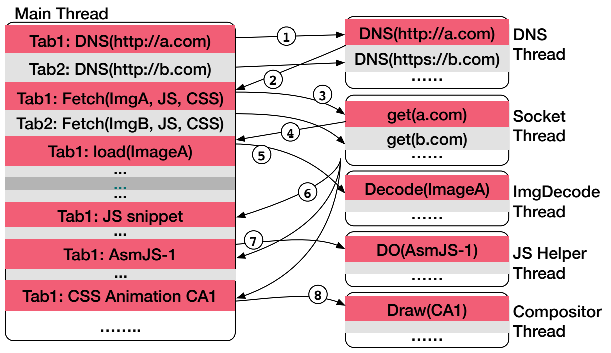


Figure 4: Simplified Firefox execution model

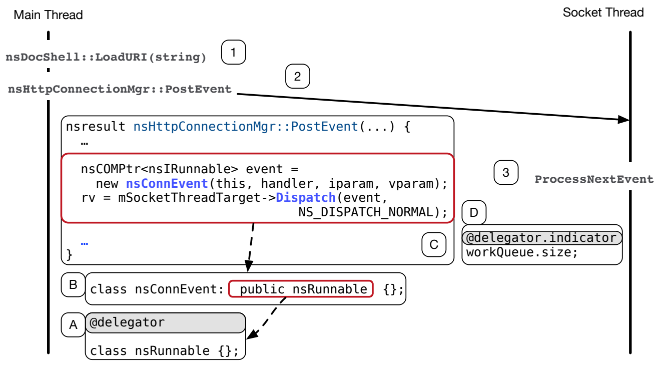
☐ *Example.* Figure 4 illustrates a substantially simplified example of the Firefox execution model. It corresponds to an execution that loads two pages (in two respective tabs). Specifically, each box represents a thread and each colored bar (inside a box) denotes an iteration of the event handling 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. ☐

Different from BEEP/ProTracer, our solution is to leverage annotations and static analysis to partition directly according to the logic tasks (e.g., tabs). 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.

```
@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 5: Tab and window annotations in Firefox

☐ *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 5 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()*.



```
nsresult nsHttpConnectionMgr::PostEvent(...) {
  …
  nsCOMPtr<nsIRunnable> event =
    new nsConnEvent(this, handler, iparam, vparam);
  rv = mSocketThreadTarget->Dispatch(event,
                                     NS_DISPATCH_NORMAL);
  …
}
```

```
class nsConnEvent: public nsRunnable {};
```

```
@delegator
class nsRunnable {};
```

```
@delegator.indicator
workQueue.size;
```

Figure 6: 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 6. 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. 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. ☐

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.

☐ *Example.* Let us revisit the Firefox example in Figure 4. We want to attribute all subtasks to their corresponding tabs (shown in different colors). In Figure 6, 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*, AUTOMPI 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, AUTOMPI is able to recognize all the tab1 subtasks performed by different threads, namely, all the red bars in Figure 4 belong to the same tab1 unit. ☐

4.3 Annotation Challenges:

As described in the previous section, the annotation phase determines three types of annotations, by leveraging the miner. But these annotation types span over different code locations. For example, according to Figure 3, the indicator variable *curbuf* is located in the file *src/globals.h*, while the identifier variable *b_ffname* and channel variable *y_current* are located in the files *src/structs.h* and *src/ops.c*, respectively. Furthermore, these variables are pointers, thus we need to track memory to identify the initialization of the variable required for the annotation. Conducting this analysis manually can be cumbersome. For addressing these challenges, we need to automate this process to capture precisely and effectively the required variables. The next section introduces AUTOMPI and its *Automated Annotations Identifier* that applies a hybrid approach to automatically determine the required data structures and variables.

5 Automatic Annotations Identifier

AUTOMPI primarily relies on two annotations (i.e., indicator and channel variables) to specify the locations where to perform the annotation. Figure 7 depicts our approach to automate the process of identifying the indicator and channel variables. The process comprises the following steps:

1) Dynamic Trace Analysis: determines relevant program regions where the indicator and channel variables are located. The instrumentation step adapts the program to collect visited functions and their parameters. We run the instrumented binary several times to collect different traces. These traces are analyzed by applying differential analysis to determine relevant functions that statistically host the channel/indicator variables.
2) Static Liveness Analysis: analyzes the shortlisted program regions by leveraging kill-variable analysis techniques to obtain indicator or channel variable candidates.

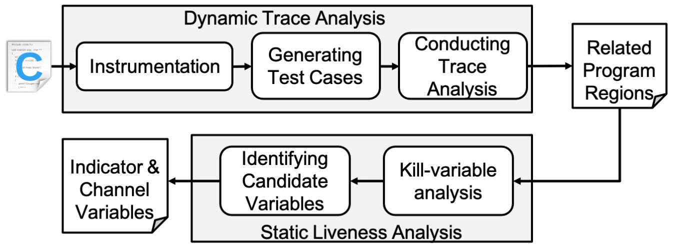


Figure 7: The discovery process of indicator and channel variables.

5.1 Dynamic Trace Analysis

In this step, we instrument the program to record the visited methods and generate a customized executable. This instrumented executable is run with carefully crafted test cases. Specifically, the test case needs to trigger the high-level perspective (cf. described in Section 2) intended by the user.

The format of the generated trace is a list of function calls, including the program entry point and all exits, with the corresponding parameters and return value information. We then perform differential analysis over the generated traces to identify the relevant regions of the application code. Although the same steps are used for identifying indicator and channel variables in the dynamic trace analysis, we realize this phase differently, as explained below, due to the distinctive requirements for determining these annotation data structures.

5.1.1 Indicator Variables

Since the indicator variable represents perspective switching, the test cases should fire a change in the perspective that will be traced in the application. Therefore, generating two test cases is sufficient to serve our goal, where the second test case replicates the actions of the first test case and repeats the same actions in a new context. For 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. Therefore, a natural way to partition its execution is to use two test cases as illustrated in Figure 8. The first one (*Test A*) involves inserting a text, while the second test case (*Test B*) performs the same actions on the same text file used in *Test A*, but also on another text file. *Test B* changes the unit context that we track. This ultimately reflects some data structures and memory regions are accessed twice in the trace generated by *Test B* in contrast to the trace generated by *Test A*, as depicted in Figure 8.

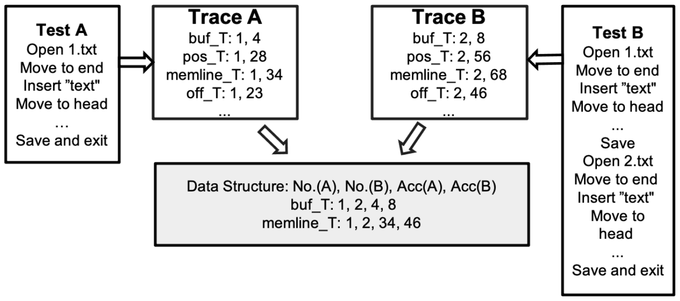


Figure 8: Test cases crafted for identifying the indicator variable in text editor programs like Vim.

Based on these traces, we perform differential trace analysis to obtain candidate callback functions. Thus, we can identify function calls that are relevant to the perspective switch. These relevant calls usually do not appear in both traces, where common function calls are considered irrelevant to the context switch of the intended perspective (i.e., switching between tabs in Chrome).

5.1.2 Channel Variables

Similar to the dynamic trace analysis process for indicator variables, the test cases for discovering the channel variables should trigger accesses of variables or data structures that reflect the IPC channel. This is because one channel variable resembles the IPC channel, where data will be transferred between different perspectives. We still use the same logic to generate multiple traces. However, it is mandatory that the test cases are properly designed, wherein one writes to the channel while the other does not. Therefore, we can discover program regions that contain the channel variable, which is accessed by one but not by the other. Figure 9 illustrates a test case that we used to trigger the IPC channel between two contexts for the Vim application. The test case copies text data from one text file to another text file.

Figure 10 illustrates the differential analysis process for the generated trace. It identifies the memory location where the same data is copied and pasted. This is because the data is loaded in the first file, while the data is stored in the second file. Therefore, in the differential analysis, we identify memory regions where the data is both accessed and stored.

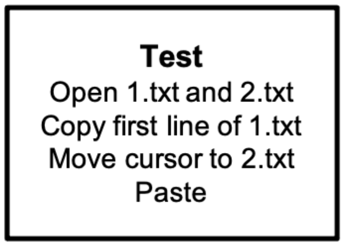

Test
Open 1.txt and 2.txt
Copy first line of 1.txt
Move cursor to 2.txt
Paste

Figure 9: Test case requirements for identifying channel variables in text editor programs like Vim.

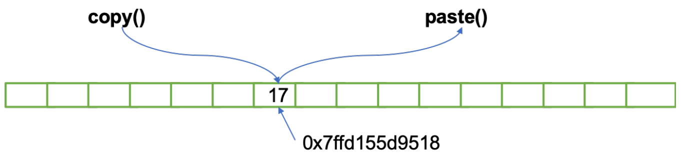


Figure 10: Tracing the data copied for triggering the channel variable.

5.1.3 *Trace Analysis Logic*

A 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 *Automated Annotations Identifier* 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.

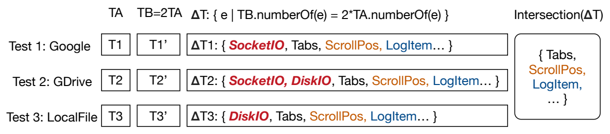

TA
TB=2TA
ΔT: { e | TB.numberOf(e) = 2*TA.numberOf(e) }
Intersection(ΔT)
Test 1: Google
T1
T1'
ΔT1: { SocketIO, Tabs, ScrollPos, LogItem… }
Test 2: GDrive
T2
T2'
ΔT2: { SocketIO, DiskIO, Tabs, ScrollPos, LogItem… }
Test 3: LocalFile
T3
T3'
ΔT3: { DiskIO, Tabs, ScrollPos, LogItem… }
{ Tabs, ScrollPos, LogItem, … }

Figure 11: Trace Analysis

Next, we show how to mine the tab data structure in Firefox (Figure 11). 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 *Automated Annotations Identifier* 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 metadata 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 remaining, many not related to units (e.g., for logging). Hence, in the last step, PageRank is used to rank several top data structures. In our case, the tab data structure has correctly ranked top.

5.2 Static Liveness Analysis

Static liveness analysis receives the identified regions in the previous step, which facilitates and optimizes this analysis. In this step, we identify candidate indicator and channel variables that will be used in the annotation process. The previous phase provides the relevant regions as a shortlist of function calls. The intuition is that indicator and channel variables, which are involved in perspective switching, should be defined and killed within the scope of the identified program regions (i.e., function callbacks). Therefore, we need to analyze the liveness of the variables within the scope of the identified regions.

Similar to the previous phase, we apply the static liveness analysis differently for identifying the indicator and channel variables. Following is a description of our approaches:

5.2.1 *Indicator Variable Liveness Analysis*

The process of identifying indicator variables leverages the fact that the values of these variables are changed within the scope of the shortlisted functions. To this end, we perform value flow analysis [16]. Our analysis is performed on top of the sparse value flow graph (SVFG) [3], which captures def-use chains and value flow via assignments for all memory locations represented by both top-level and address-taken pointers. We devise Algorithm 1 to discover candidate indicator variables. Our algorithm receives the shortlisted functions and the target program P. It returns a list of candidate indicator variables. We first run a static single assignment (SSA) for the program P (Line 1). Next, our algorithm generates the SVFG (Line 2) and then iterates over its nodes to identify the shortlisted function calls (Line 5). It then tracks whether the parameters of each function have been modified since its invocation (Line 7). The parameters of the shortlisted functions are the source nodes of the incoming edges to the function call node in SVFG. We trace if these incoming nodes and outgoing nodes point to the different memory regions, which indicates the value has been changed. We perform pointer analysis to identify all variables pointed to by each incoming node (Line 8). Finally, we trace back the modified parameters to identify their sources in the scope of the caller functions. These sources are considered indicator variables.

Algorithm 1: Static Liveness Analysis to identify Indicator Variables

```
   Input: Regions R = $\{F_1, ..., F_n\}$, Program $P$
   Output: Indicator Variables IV
1  $P' \longleftarrow$ Generate SSA of P
2  SVFG $\leftarrow$ Perform value-flow analysis for $P'$
3  IV $\leftarrow \{\}$
4  foreach $node \in SVFG$ do
5      if $node \in R$ then
6          foreach src connected to node do
7              if value of src is changed after node then
8                  PA $\leftarrow$ Perform PointToAnalysis($node$)
9                  foreach $ptr \in PA$ do
10                     Add $node$ to IV
```

5.2.2 *Channel Variable Liveness Analysis*

3. https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#12-value-flow-graph provides an illustration for SVFG

The process of identifying candidate channel variables involves discovering variables that have been modified once in the scope of the shortlisted functions. Since the behavior of the channel variable reflects accessing the variable in one function and modifying it in another function, as depicted in Figure 10, the channel variable should be global.
Algorithm 2 describes our liveness analysis to discover channel variables. The ultimate goal of this algorithm is to identify global variables that are modified only once within the process of changing perspectives. The shortlisted function calls and the target program are required to perform our analysis. For every shortlisted function (Line 5), we first iterate over all instructions within its scope and filter these instructions (Lines 6-7) to find variables whose values have been modified once. Specifically, we look for store instructions where the destination operand has not been accessed. Then, we perform a pointer analysis to identify the set of variables pointed to by the destination operand of the store instruction (Lines 8-9). Finally, all pointed global variables are considered channel variables.

Algorithm 2: Static Liveness Analysis to identify Channel Variables

```
Input: Regions R = $\{F_1, ..., F_n\}$, Program $P$
Output: Channel Variables CV
1  $P' \longleftarrow$ Generate SSA of P
2  SVFG $\leftarrow$ Perform value-flow analysis for $P'$
3  CV $\leftarrow$ {}
4  foreach $node \in SVFG$ do
5      if $node \in R$ then
6          foreach $instr \in node$ do
7              if instr isn't used by all instructions of node $\wedge$ instr is StoreInstr then
8                  $dst$ $\leftarrow$ Get Dest. Operand of $instr$
9                  PA $\leftarrow$ Perform PointToAnalysis($dst$)
10                 foreach $var \in PA$ do
11                     if var is global variable then
12                         Add $var$ to CV
```

5.3 Heuristic Analysis

For some programs, our static liveness analysis yields several candidate variables. Therefore, we apply a heuristic analysis to reduce the number of candidate indicator and channel variables. This optimizes the annotation overhead. As mentioned earlier, these variables have a compound data type (i.e., struct). As the annotation variables are used by various functions in the program, the scope of candidate variables should be global. Finally, we apply a ranking approach for selecting relevant candidate variables. The ranking approach relies on the frequency of the variables among the list of candidate variables. Specifically, we apply the same logic described in Section 5.1.3 and illustrated in Figure 11, which basically prunes low frequent variables in the list of candidate variables suggested by static liveness analysis.

5.4 Guidance for Designing Test Cases.

The *Automated Annotations Identifier* identifies automatically variables corresponding to the perspectives desired by the user. However, identifying the test cases can be a challenging task. In this section, we discuss AUTOMPI abstraction logic of perspectives and how to eliminate annotating irrelevant data structures.

The selection of test cases ultimately depends on the perspective that the user is interested in capturing its provenance data. Perspectives are represented by high-level tasks involved in an execution. For example, if the intended perspective is a Firefox tab or a Vim file, then the test cases should exercise them, but the strategy of executing the test cases is dependent on the required annotations to be identified:

- **Indicator Variables.** Two test cases (as illustrated in Figure 8) are required. The first execution should trigger the intended unit task (e.g., reading a file, writing to a file, opening a tab), while the second test case is a replication of the first execution but twice.
- **Channel Variables.** One test case is sufficient, but it should trigger IPC behavior (i.e., transferring data between two units/elements within the same perspective). For example, in the test case illustrated in Figure 9, the same text is copied from one file to another.

6 Implementation

The AUTOMPI prototype is developed with LLVM. We directly use the language extension provided by Clang [17] to annotate the source code and provide several LLVM passes to perform the instrumentation tasks. For most of the applications evaluated in §7, we obtained the source code from the Ubuntu official source code repositories to leverage the latest Ubuntu-specific patches and updates. For others that are not in the repositories, we acquired the source code from their official websites. AUTOMPI performs its analysis after converting the source code to LLVM IR using WLLVM [18]. We leverage LOOM [19, 20] to perform the instrumentation in the dynamic trace analysis. The instrumentation policy required by LOOM is constructed to record the entry and exit visited functions in the execution trace. We construct the instrumentation policy by implementing an LLVM pass to collect all functions existing in the LLVM IR code.
The static liveness analysis is conducted using the SVF tool [16]. SVF performs a precise interprocedural dependence analysis for C and C++ programs. SVF generates several graphs, but our analysis leverages the inter-procedural sparse value-flow graph (SVFG) and program assignment graph (PAG). SVFG is a directed graph that captures the def-use chains of both top-level pointers (as direct edges) and address-taken objects (as indirect edges, each from a variable's definition to its usage). One SVFG node can be a statement (i.e., a PAG edge), a memory region definition, or a parameter. While the PAG represents LLVM pointer assignments, a PAG node is a pointer and each edge represents a constraint (i.e., load or store) between two pointers.
SVF expects the LLVM IR code as input and generates various graphs based on different static analysis techniques, such as pointer analysis (e.g., Andersen's analysis). We modified SVF to perform its analysis over the shortlisted program regions (function calls identified), where we expect to discover the indicator and channel variables. We can also optimize our liveness analysis by identifying the specific data type of the candidate indicator and channel variables, by identifying the most used variables within the shortlisted function calls.

7 Evaluation

This section presents our experimental evaluation of AUTOMPI. We address the following research questions:

- **RQ1:** Can AUTOMPI identify indicator and channel variables correctly?
- **RQ2:** What is the analysis time required by the automated miner component in AUTOMPI?
- **RQ3:** What is the overhead of AUTOMPI modifications on the annotated programs?
- **RQ4:** What is the effectiveness of AUTOMPI in attack investigation?

The performance reported RQ1 and RQ2 based on a Ubuntu 16.04 machine with a 3.8GHz Intel(R) Core(TM) i7-6700K CPU and 32GB RAM.

7.1 Accuracy of autoMPI (RQ1)

This section reports the capability of AUTOMPI to identify correctly channel and indicator variables. For constructing the ground truth, we inspected manually a set of programs to identify the channel and indicator variables. This step is already performed in MPI. We then use AUTOMPI to automatically identify the channel and indicator variables. Finally, we compare the variables identified using both approaches. Table 2 and Table 3 show AUTOMPI can accurately identify the channel and indicator variables. The accuracy of identifying the channel variable is 100%, while AUTOMPI identified more indicator variables in contrast to the ground truth since AUTOMPI applies pointer analysis, which provides the capability to uncover more relevant variables to the indicator variable. However, the main indicator variables are identified correctly for all programs in Table 2. For `Vim`, we performed flow-insensitive analysis to reduce the overhead of the pointer analysis, thus yielding more indicator variables.

Table 2: Comparing Indicator Variable identification results between MPI and AUTOMPI.

Program	Size	# Indicator Variable	
		MPI	AUTOMPI
Apache	3.1M	2	2
MC	5.9M	2	5
Vim	17M	3	35
W3m	6.4M	2	2
Yafc	1.3M	3	4

Table 3: Comparing Channel Variable identification results between MPI and AUTOMPI.

Program	Analysis Time	# channel variables	
		MPI	AUTOMPI
nano	6.6s	2	2
vim	83m31s	2	3

7.2 Analysis Time (RQ2)

In this experiment, we measure the overhead of AUTOMPI and describe the performance of the heuristic analysis.

Indicator Variable. The identification of indicator annotation results are shown in Table 4. We present the applications in the first column and their sizes (measured by SLOCCount [21]) in the second column. The second and third columns show the number of nodes and edges in the SVFG. Next, we show the analysis time in seconds and the number of identified indicator variables before and after applying the heuristic analysis. Our analysis is lightweight; on average the analysis time for all programs is 16 seconds, except for the Vim program, where the analysis time is around 150 minutes due to its complexity (though we did not face this problem in the Apline program). This table shows the benefits of the heuristic approach to minimize the number of indicator variables.

The columns #Nodes/Edges provide the characteristics of the SVFG, which on top of it the dataflow analysis is performed. However, other factors can affect the analysis time like the number of pointer variables, stores/loads, indirect call sites [22]

Next, we describe the accuracy of AUTOMPI how the leveraged heuristics analysis improved the accuracy. Table 4 reports the number of indicator variables identified before and after applying the heuristic analysis. For example, the automated annotation miner identifies more candidate variables (in 3 programs out of 5), but these additional variables are marginal in `MC` and `Yafc`. However, more additional variables are identified for `Vim` because we used flow-insensitive pointer analysis, while flow-sensitive analysis is conducted for the rest of the programs. Due to its complexity, the flow-sensitive analysis for `Vim` takes a long time. The last five columns in Table 4 describes the accuracy of AUTOMPI from different perspectives. The #FP column represents the number of incorrectly (False Positive) identified indicator variables after the heuristic analysis. False positive is inevitable in the context of static analysis tools [23]. Therefore, AUTOMPI applies heuristic analysis as a suppression mechanism. #FN is zero for all programs because AUTOMPI did not miss any indicator variable. The precision and recall are computed in the last two columns based on the columns #FP, #TP, and #FN. The average precision and recall are 72.5% and 100%, respectively. Accordingly, the resulting F1 score is 84%.

Channel Variable. Table 3 reports the results of identifying channel variables required for the annotation. Only two programs support the behavior of channel annotation. The automated miner can identify the required channel variables correctly. Similar to the observations discussed in the indicator variables results, `Vim` identified one additional variable and the analysis time is longer in contrast to `Nano`.

7.3 Overhead (RQ3)

We conducted two experiments to evaluate the overhead of AUTOMPI. First, we measure the space overhead of AUTOMPI and compare it with relevant provenance tracking systems. Second, we determine the run-time overhead of the annotated programs.

Space overhead: We measure the space overhead of AUTOMPI and compare it with the overhead of event-loop-based partitioning, on the aforementioned three provenance tracking systems. We measure the overhead of AUTOMPI 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 AUTOMPI. Note that BEEP+ProTracer is equivalent to the original ProTracer system [2] and in

Table 4: Analysis Time and Performance of autoMPI automated miner before and after applying heuristic analysis

Program	LOC	#Nodes	#Edges	Time (s)	# Indicator Variable Before Heuristic	# Indicator Variable After Heuristic	# FP	# TP	# FN	Precision	Recall
Alpine	360,168	411,545	729,195	10	13	2	0	2	0	1	1
Apache	184,401	64,263	74,554	8	15	2	0	2	0	1	1
Bash	113,693	159,807	341,619	6	711	23	20	3	0	0.13	1
Leafpad	5,049	7,605	8,637	1	8	1	0	1	0	1	1
MC	129,603	91,913	130,540	23	10	5	3	2	0	0.4	1
Mutt	80,929	101,776	213,933	2	1	1	0	1	0	1	1
Most	5,983	4,872	6,139	1	8	2	0	2	0	1	1
Nano	16,657	29,425	51,458	29	46	3	0	3	0	1	1
Vim[a]	321,223	327,381	884,705	8,500	770	35	32	3	0	0.085	1
W3m	57,649	93,249	193,210	53	2	2	0	2	0	1	1
Wget	114,852	41,892	71,723	28	59	9	6	3	0	0.33	1
Yafc	19,002	24,707	39,207	14	37	4	1	3	0	0.75	1

[a]Reported numbers of edges and nodes are based on flow-insensitive analysis.

Table 5: Space Overhead

Program	Level	BEEP Space Overhead			autoMPI Space Overhead			BEEP	autoMPI [a]
		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

[a]logs generated by the instrumented binaries by both MPI and autoMPI are very similar and did not incur space overhead.

autoMPI +ProTracer we retain the efficient runtime of the original ProTracer, but replace the partitioning component with autoMPI. Since BEEP supports only one low-level perspective, we only annotate one perspective in autoMPI during comparison.

The results are shown in Table 5. 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 userspace reporter tool; (6-8) Overhead of autoMPI on BEEP and LPM-HiFi; (9) Log size of BEEP on (original) ProTracer; (10) Log size of autoMPI 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 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 that they rarely trigger the instrumented code. Our approach exhibits better results for many programs such as web browsers, P2P clients, HTTP, and FTP programs, including servers and clients, due to a few reasons. In these programs, the events handled by the event handling loop are at a very low level, whereas autoMPI can partition execution at a much higher level. Thus, there are fewer unit context switches in our system, and multiple BEEP execution units 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 autoMPI, multiple requests, including their redirections, of the same connection are grouped together. Thus, the instrumentation (for the 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 autoMPI. Firefox has a 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 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 autoMPI, we can even reduce space overhead for the highly efficient ProTracer system and the reduction is substantial in some cases. This is because autoMPI 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. Note also that all the advantages of AUTOMPI over BEEP (e.g., not requiring extensive training and rich high-level semantics) are also advantages over ProTracer, as the original ProTracer system relies on BEEP. We have run AUTOMPI for 24 hours with a regular workload. The generated audit log has 680MB with 80MB by AUTOMPI.

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* [24] 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 in Figure 12. For each program, we have eight bars. ①AUTOMPI-Native: the overhead of AUTOMPI without any provenance system over native-run; ②AUTOMPI-ProTracer: the overhead of AUTOMPI over ProTracer; ③AUTOMPI-LPM: the overhead of AUTOMPI over LPM-HiFi; ④AUTOMPI-Audit: the overhead of AUTOMPI over Linux Audit. The other four bars denote the overhead of BEEP. As we can see in Figure 12, most applications have less than 1% run time overhead for all situations, which is acceptable. Comparing to BEEP, AUTOMPI shows less overhead in all cases. The low run-time overhead is due to the following factors. First, compared with the original program, the number of instrumented instructions is quite small. Second, most of the instructions are rarely triggered. Third, our instrumentation mainly contains memory operations such as comparing the newly assigned *identifier* value with the cached value.

7.4 Attack Investigation (RQ4)

To evaluate AUTOMPI's effectiveness in attack investigation, we apply it to 13 realistic attack cases used in previous works [2, 6, 25, 26]. The results show that AUTOMPI is able to correctly identify the root causes with very succinct causal graphs for all cases. Moreover, AUTOMPI generates fewer execution units using the perspectives in Table 5, when compared to BEEP/ProTracer. On average, the number of units generated by AUTOMPI 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, AUTOMPI can reduce the number of nodes by 8% and the number of edges by 17% on average. This is due to the fact that these attacks have simple propagation paths such that the BEEP/ProTracer graphs are quite succinct. For complicated cases, AUTOMPI can reduce the graphs by 24%(nodes)/38%(edges). In addition, we evaluate it on a few other realistic attack cases. Next, we present one such case.

Case: FTP Data Leak. Exploiting system misconfiguration to acquire valuable sensitive information is a common attack vector [27, 28]. 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 access. After noticing the problem, he shut down the server and then conducted an investigation to assess the significance of the potential information leak. During 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 13, we show a number of possible investigation perspectives for the FTP server application. Event-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). AUTOMPI 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; hence, we can partition FTP execution based on this information.

Part of the BEEP graph is shown in Figure 14. Observe that each user command is captured as a unit. The simplified graph by AUTOMPI with connection-based partitioning is shown in Figure 15 and user-based partitioning in Figure 16. 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 such that the inspector can hold individual users responsible. Note that a user can use various IP addresses to connect to the server. Without AUTOMPI, such semantic information cannot be exposed to the provenance tracking system. The number of nodes in BEEP, connection (AUTOMPI), and user (AUTOMPI) graphs are 962, 224, and 78, respectively. We want to point out that the AUTOMPI 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.

8 Discussion

Similar to many existing works [2, 6, 26, 29, 30], AUTOMPI trusts the Linux kernel and the components associated with the audit logging system. Attacks that can bypass the security mechanisms of these systems may cause problems for AUTOMPI. Moreover, attacks that target the underlying audit system, such as audit log blurring and log filling may inject noise into logs, making log inspection difficult. As our system is built on top of existing provenance and operating systems, AUTOMPI leverages existing features provided by these systems to mitigate some of the problems. For example, operating systems like Ubuntu now leverage Ubuntu Software Center to deliver trustworthy software, which can be used to protect the AUTOMPI 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 are not the focus of our paper. Instead, the emphasis of AUTOMPI is to address

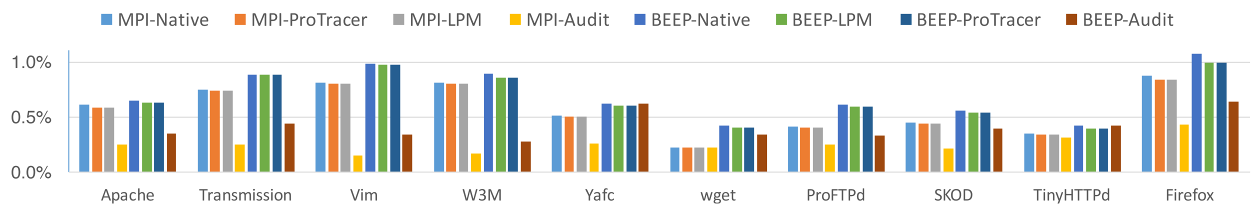


Figure 12: Run time overhead for each application (Overhead percentage v.s. applications)

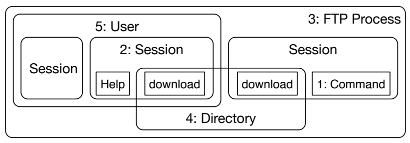


Figure 13: FTP server partitioning perspectives

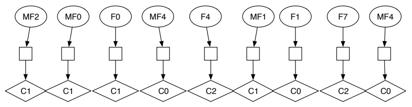


Figure 14: FTP server partitioned by BEEP

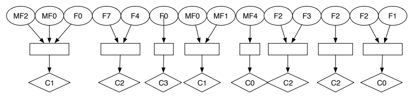


Figure 15: FTP server partitioned by each connection

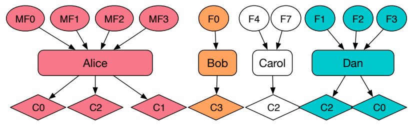


Figure 16: FTP server partitioned by users

the dependence explosion caused by long-running processes with accuracy and flexibility.

AUTOMPI relies on source-code annotations, which are widely used in practice. Windows developers explicitly plant logging commands in their source code to customize ETW auditing. Both GCC and LLVM provide advanced language features [17, 31, 32] 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 throughout the code base. In contrast, we only introduce 36 annotations (of 4 types) in Firefox.

As AUTOMPI is based on source-code level annotation and compiler instrumentation, it cannot find units within dynamic code. Since the instrumentation policy, in the dynamic trace analysis, is statically generated via an LLVM pass that collects names of all functions existing in the WLLVM IR code, there is a possibility the policy will not include dynamically loaded functions. Therefore, visits to these functions are absent from the generated trace. However, in practice, we find those 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. On the other hand, AUTOMPI performs its analysis based on the source code after converting it to LLVM IR; therefore, obfuscation is not a challenge. Moreover, the instrumentation step does not break the program's functionality because it just profiles visited functions according to the instrumentation policy. Finally, the instrumentation step is fully automated as described in Section 6.

The approach used in the automated annotator integrates various program analysis techniques. The conjunction of these techniques can be broadly leveraged in other domains to automatically identify code locations that satisfy certain properties. For example, triggering contexts is the property used to identify indicator variables, while accessing a variable by one function and modifying it in another function represents the property used for discovering channel variables. Therefore, we believe the proposed hybrid approach can be adopted to tackle similar automation purposes, especially since the proposed approach is accurate and imposes a lightweight overhead.

Pointer analysis can easily blow up in bigger programs (i.e., the analysis on `Vim` would not complete for this reason). To handle this issue, we perform a flow-insensitive analysis if the flow-sensitive one times out. However, this imprecise analysis comes at the cost of accuracy, in which some indicator variable candidates are false positives. For example, we found 35 indicator variable candidates for `Vim`, only 3 are indicator variables.

In this work, we extended AUTOMPI to automatically identify the two important annotation variables: indicator and channel. AUTOMPI can also identify the identifier variable since this variable is corresponding to the indicator variable. Therefore, the identifier variable is recognized using the same process applied for capturing the indicator variable. Though AUTOMPI supports annotating threaded applications, the thread-relevant data structures are identified manually. Our future plan is to extend the automation capability of the automated annotator to support multi-threaded applications. We rely on the requirements and data structures defined in MPI [7] to perform the annotation process.

In section 5.4, we provided some tips for selecting test cases required for the dynamic trace analysis. However, we foresee other techniques like symbolic execution can facilitate the identification of indicator and channel variables. For example, LMCAS [33] leverages KLEE to identify the set of variables corresponding to concrete run-time configurations supplied to KLEE. Furthermore, symbolic execution can be employed to generate the test cases [34] required for triggering the desired high-level tasks (perspectives)

9 Related work

Many approaches have been proposed for system-level provenance tracking [35–38]. Another important approach is to monitor the internal kernel objects (e.g., the file system [3–5, 39–42], or LSM objects [25, 29, 30]) to track lineages. The capabilities of these techniques are similar to those of the audit systems.

Thus AUTOMPI is complementary to such systems. For example in §7, we showed the integration of AUTOMPI and LPM-HiFi. System-wide record-and-replay techniques [1, 43–45] 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 [11, 46–49] 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. AUTOMPI can be applied to such systems to overcome the dependency explosion problem and enable multiple perspective inspection.

TRACE [38] used a host monitoring component to collect forensic artifacts at run time. The artifacts were collected from two perspectives, namely, system calls and unit-based selective instrumentation (UBSI). While this could speed up the analyst process, a limitation is that significant data storage is required to maintain the dependency knowledge.

In [50], researchers propose to develop provenance-aware applications. Muniswamy-Reddy *et. al.* [4] 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, AUTOMPI aims to address the partitioning problem. Provenance tracking is through the underlying audit system.

Differential analysis has been used in the body of research. Chen et al. [51] used differential analysis for removing irrelevant variables, while Aafer et al. [52] used differential analysis to detect inconsistent security features in Android images.

10 Conclusion

In this paper, we propose AUTOMPI, a technique that partitions based on high-level tasks. It allows the user to automatically annotate the data structures corresponding to these tasks, and leverages the compiler to instrument operations of the data structures to capture unit context switches and delegations. The automated annotation miner applies a hybrid analysis approach to identifying the required data structures.

We implemented a prototype and evaluated it on three existing systems: Linux Audit, ProTracer, and LPM-HiFi. The results show that AUTOMPI generates much smaller graphs with lower overhead compared to the state of the art and avoids broken provenance due to incomplete training. We also evaluated the performance and accuracy of the automated annotation miner.

11 Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments. We also would like to thank Zhicheng Cai and Stephen Lee for their help with the implementation.

References

[1] S. T. King and P. M. Chen, "Backtracking intrusions," ser. SOSP '03.

[2] S. Ma, X. Zhang, and D. Xu, "Protracer: towards practical provenance tracing by alternating between logging and tainting," ser. NDSS '16.

[3] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." ser. Usenix ATC '06.

[4] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in provenance systems," ser. USENIX ATC'09.

[5] S. Sitaraman and S. Venkatesan, "Forensic analysis of file system intrusions using improved backtracking," ser. IWIA '05.

[6] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition." ser. NDSS '13.

[7] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1111–1128.

[8] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie, A. Gehani, and V. Yegneswaran, "MCI : Modeling-based causality inference in audit logging for attack investigation," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[9] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[10] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications," 2020.

[11] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1172–1189.

[12] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-Supported Cost-Effective audit logging for causality tracking," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/ma-shiqing

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

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

[15] "Vim document: windows," https://goo.gl/Lqp9Gb.

[16] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[17] "Clang language extensions," https://goo.gl/UpniZC.

[18] I. A. Mason, "Whole program llvm," https://pypi.org/project/wllvm/, accessed Feb 06, 2020.

[19] J. Wu, H. Cui, and J. Yang, "Bypassing races in live applications with execution filters," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 135–149.

[20] "Loom: Llvm instrumentation library," https://github.com/cadets/loom, accessed Feb 06, 2020.

[21] "Sloccount," http://www.dwheeler.com/sloccount/.

[22] P. D. Schubert, R. Leer, B. Hermann, and E. Bodden, "Know your analysis: How instrumentation aids understanding static analysis," in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 8–13. [Online]. Available: https://doi.org/10.1145/3315568.3329965

[23] M. Nachtigall, L. Nguyen Quang Do, and E. Bodden, "Explaining static analysis - a perspective," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2019, pp. 29–32.

[24] "Apache benchmark," https://goo.gl/L7bGOK.

[25] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," ser. SOSP '05.

[26] K. H. Lee, X. Zhang, and D. Xu, "Loggc: garbage collecting audit log," ser. CCS '13.

[27] "Leaked data," https://haveibeenpwned.com/.

[28] "The sony hack," https://goo.gl/B4G7Pl.

[29] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 319–334. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/bates

[30] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: Collecting high-fidelity whole-system provenance," ser. ACSAC '12.

[31] "Extensions to the c language family," https://goo.gl/evrruW.

[32] "Extensions to the c++ language," https://goo.gl/pn19Np.

[33] M. Alhanahnah, R. Jain, V. Rastogi, S. Jha, and T. Reps, "Lightweight, multi-stage, compiler-assisted application specialization," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 251–269.

[34] P. Godefroid, "Test generation using symbolic execution," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[35] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *Network and Distributed Systems Symposium*, 2018.

[36] Y. Xie, D. Feng, Y. Hu, Y. Li, S. Sample, and D. Long, "Pagoda: A hybrid approach to enable efficient real-time provenance based intrusion detection in big data environments," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1283–1296, 2020.

[37] C. Sáenz-Adán, B. Pérez, F. J. García-Izquierdo, and L. Moreau, "Integrating provenance capture and uml with uml2prov: Principles and experience," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 53–68, 2022.

[38] H. Irshad, G. Ciocarlie, A. Gehani, V. Yegneswaran, K. H. Lee, J. Patel, S. Jha, Y. Kwon, D. Xu, and X. Zhang, "Trace: Enterprise-wide provenance tracking for real-time apt detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4363–4376, 2021.

[39] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Issues in automatic provenance collection," in *Provenance and annotation of data*.

[40] N. Zhu and T.-c. Chiueh, "Design, implementation, and evaluation of repairable file service," ser. DSN'13.

[41] S. Sundararaman, G. Sivathanu, and E. Zadok, "Selective versioning in a secure disk system," ser. Usenix Security'08.

[42] D. J. Tian, A. Bates, K. R. Butler, and R. Rangaswami, "Provusb: Block-level provenance-based data protection for usb storage devices," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 242–253. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978398

[43] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality." ser. NDSS '05.

[44] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," ser. OSDI'10.

[45] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 525–540.

[46] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," ser. NDSS'05.

[47] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," ser. USENIX SSYM'04.

[48] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," ser. OSDI '06.

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

[50] S. Miles, P. Groth, S. Munroe, and L. Moreau, "Prime: A methodology for developing provenance-aware applications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, p. 8, 2011.

[51] Y. Chen, W. You, Y. Lee, K. Chen, X. Wang, and W. Zou, "Mass discovery of android traffic imprints through instantiated partial execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 815–828.

[52] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android roms via differential analysis," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, Aug. 2016, pp. 1153–1168.

Mohannad Alhanahnah received his Ph.D. in Computer Engineering from the University of Nebraska-Lincoln. He is currently a Research Associate at the Department of Computer Science, University of Wisconsin-Madison. His research interest spans over the area of software security and adversarial machine learning.

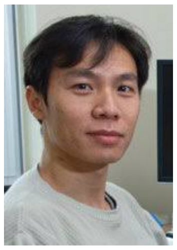
Shiqing Ma is an Assistant Professor in the Department of Computer Science at Rutgers University, the state university of New Jersey. He received his Ph.D. in Computer Science from Purdue University in 2019 and B.E. from Shanghai Jiao Tong University in 2013. His research focuses on program analysis, software and system security, adversarial machine learning, and software engineering. He is the recipient of Distinguished Paper Awards from NDSS 2016 and USENIX Security 2017.

Ashish Gehani is a Senior Principal Computer Scientist at SRI. His research interests are data provenance and security. He holds a Ph.D. in Computer Science from Duke University and a B.S. (Honors) in Mathematics from the University of Chicago.

Dr. Gabriela F. Ciocarlie is an associate professor in the Department of Electrical and Computer Engineering at University of Texas at San Antonio and a vice president for securing manufacturing and secure manufacturing architecture at the Cybersecurity Manufacturing Innovation Institute (CyManII). Gabriela's expertise is in anomaly detection, application level security, cyber-physical and distributed system security. Gabriela received a PhD from Columbia University. She is an associate editor for IEEE Security & Privacy and an IEEE member. Contact her at gabriela.ciocarlie@utsa.edu.

Somesh Jha is the Lubar professor in the Computer Sciences Department at the University of Wisconsin, Madison. His research focused on formal methods for security, privacy and adversarial ML. He received his B.Tech from Indian Institute of Technology, New Delhi in Electrical Engineering. He received his Ph.D. in Computer Science from Carnegie Mellon University under the supervision of Prof. Edmund Clarke (a Turing award winner).

Vinod Yegneswaran is a Principal Computer Scientist at SRI. His research interests are in network and systems security. He holds a Ph.D. in Computer Science from the University of Wisconsin and an A.B. in Computer Science from the University of California at Berkeley.



Xiangyu Zhang is a Samuel Conte professor in the Computer Sciences Department at Purdue University, West Lafayette. His research focuses on software and deep learning security. He received his Ph.D. in Computer Science from the University of Arizona.