Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows

Shiqing Ma[†], Kyu Hyung Lee[‡], Chung Hwan Kim[†], Junghwan Rhee[§],
Xiangyu Zhang[†], Dongyan Xu[†]
[†]Purdue University, [‡]University of Georgia, [§]NEC Laboratories America

ABSTRACT

Audit logging is an important approach to cyber attack investigation. However, traditional audit logging either lacks accuracy or requires expensive and complex binary instrumentation. In this paper, we propose a Windows based audit logging technique that features accuracy and low cost. More importantly, it does not require instrumenting the applications, which is critical for commercial software with IP protection. The technique is build on Event Tracing for Windows (ETW). By analyzing ETW log and critical parts of application executables, a model can be constructed to parse ETW log to units representing independent sub-executions in a process. Causality inferred at the unit level renders much higher accuracy, allowing us to perform accurate attack investigation and highly effective log reduction.

1. INTRODUCTION

Cyber attacks are increasingly targeting enterprise environments, including corporate, financial, government, and educational institutions. These attacks range from traditional single-vector, "blanket" attacks to advanced targeted attacks (also called advanced persistent threats or APTs). Cyber attacks against enterprises are launched by sophisticated attackers, often backed by adversarial groups or organizations, for financial gain, intelligence collection, or social/political disruption. They tend to be stealthy, low-and-slow, and sometimes disguised via psychosocial campaigns. The traditional signature based scanning for individual applications may become suboptimal for APT attack detection and investigation.

Audit logging is hence an important approach. With audit logs, one can trace back the "entry point" (i.e., provenance) of an attack and understand how it leads to the detected anomaly. Audit logging also reveals ramifications of the attack, i.e., what damages have been inflicted and which data assets are being targeted. In state-of-the-art audit logging techniques [11, 16, 8, 14, 21], processes are defined as subjects; whereas files, sockets, and other passive entities are defined as objects. A system/library call event creates a causal relation (edge) between a subject and an object (e.g., a process reading a file), or between two subjects (e.g., a process spawning a child process). Based on log entries recording such events, a *causal graph* can be constructed – triggered by an anomalous event – to reveal the provenance and ramifications of an attack.

However, existing techniques have the following limitations: (1) *Dependency explosion.* More specifically, the coarse granularity of processes and objects leads to the following problem: When a long-running process reads from many input objects and creates/modifies many output objects, each output object will be conservatively considered causally dependent on all preceding input objects. Meanwhile, with many processes reading/writing-to an object (e.g., a file), each reading process will be considered causally influenced by all the preceding writing processes. Dependency explosion makes causal analysis inaccurate or, in practice, unhelpful. (2) *Requiring application instrumentation and tedious training.* Some recent techniques such as BEEP [21] is able to mitigate the dependence explosion problem. However it requires analyzing a few training executions and instrumenting the executables. While the former requires substantial human efforts, the latter may not even be applicable on the Windows platform[1]. Moreover, it requires training and instrumentation after every software updates or patches which is a heavy burden to an administrator, and if the training does not cover some code section, it might produce inaccurate results. (3) Most of these techniques are based on Linux audit system that has substantial *space and runtime overhead* [21, 20].

In this paper, we develop a novel audit logging technique based on Event Tracing for Windows (ETW) [1], which is a event tracing mechanism provided by Windows. It does not require instrumenting applications. Our technique analyzes the ETW logs and critical parts of the application executables (e.g. the event handling loop) such that models for applications can be constructed. A model can be used to parse the execution of an application to independent units that are autonomous and process individual external requests. Causality analysis becomes much more precise at the unit level, which enables a number of critical applications.

Our contributions are summarized as follows.

- We propose a novel instrumentation-free audit logging technique based on ETW. It can automatically recognize event loops whose iterations denote autonomous

[1]See discussion in Section 2

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA

 ISBN 978-1-4503-3682-6/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2818000.2818039

execution units.

- The accurate causal inference enabled by units allows us to perform effective garbage reduction on ETW logs.
- We have built a prototype. Our experiments show that the overhead of our technique is trivial. The space reduction by garbage reduction ranges from 12X to 95X. The attack causal graphs are accurate and very concise, orders of magnitude smaller than most existing techniques.

2. MOTIVATING EXAMPLE

In this section, we present a concrete, realistic insider attack scenario to motivate our work and show how our technique works. The attack we will demonstrate in this section involves multiple applications such as Apache webserver (*httpd*) and *notepad++*, and also human efforts.

Scenario Description: An employee in a company is working on a project to develop a new product. He was bought by a competitor of his home company to steal some critical data related to the product. During the final test of the project, he was granted the access to the data and finally had the chance to steal it. He opened the file containing the secret, copied the confidential part and pasted it to a publicly accessible HTML file that he downloaded before using *Chromium* from *y.y.y.y*. As such, he releases the secret information without making or transporting a hard copy of the secret file, which may be easily detectable. During the procedure, the employer used two different editors, *notepad* and *notepad++*, to go through multiple copy and paste operations to cover his trail.

Traditional Investigation Process: Assume the company finds out that the information got stolen and wants to start investigation of the incident. Traditional forensic analysis techniques [15, 8, 14] collect and analyze system-level events (e.g., syscalls) to locate the source of attack and the damages caused in the system. Fig. 1(a) illustrates the causal graph generated by previous forensics techniques, in which ovals, diamonds, and boxes represent processes, sockets, and files, and are annotated by process name, socket IP address, file name, respectively. An edge denotes the direction of information flow, the causal relation, between two entities. Note that the copy and paste operations (through the clipboard) are the key events to detect the attack. However, they cannot be detected by current syscall-based techniques because the confidential information is transferred through memory operations rather than syscalls. We use dotted lines to represent copy and paste operations that cause dependence between *notepad* and *notepad++*. In this graph, we use gray boxes to represent the real attack path. The secret was originally stored in file *sec.txt*, which is edited by *notepad*. Then the employee copies the sensitive information to the *clipboard* buffer, and then pastes it to the file *htdocs/index.html*, which is opened by *notepad++*. When the attacker downloads this file from outside the company, the *httpd* server reads it and sends to the attacker, whose IP is *x.x.x.x*.

Assume now the administrator wants to identify how the secret file was leaked. She first analyzes causal dependences originating from the secret file, *sec.txt*. From the causal graph in Fig 1(a), she finds that the file is only read by process *notepad*. However, the attacker carefully designed the attack to avoid being detected by system-level audit logging. Particularly, he used copy and past operations which are not implemented by syscalls but rather memory operations.

Another approach the administrator can take is backward analysis, in which she first identifies the suspicious IP address through which the sensitive information is leaked. She then wants to figure out the provenance of the file (i.e., which processes updated the file and what operations have been performed on the file). Assume in this case, the administrator is able to recognize that IP *x.x.x.x* belongs to the competitor and the file *htdoc/index.html* was downloaded by the IP. She was able to further track down that the file was updated by both *Chromium* and *notepad* from the causal graph. However, it becomes very difficult to further traverse backward for the following two reasons. (1) *Chromium* is a long running process. During its lifetime, it accessed many IPs. It is very difficult to figure out the IP from which *htdoc/index.html* was downloaded. The root cause is that in a long running process, an output syscall has to be considered dependent on the large number of preceding input syscalls. This is called the *dependence explosion* problem [21]. (2) Since the copy-paste operations between *notepad* and *notepad++* cannot be captured. The path back to *sec.txt* is broken. As a result, the backward analysis does not help to identify the attack either.

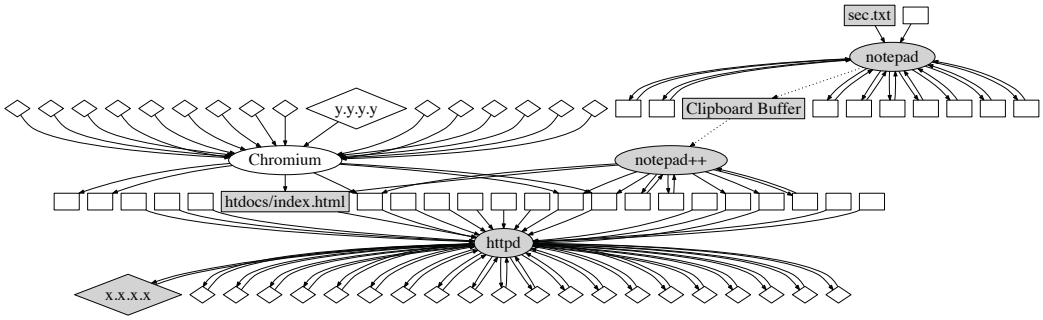
Existing Approach: Recently, researchers have proposed various approaches [15, 8, 16, 21, 25] to mitigate the dependence explosion problem. Many of them use simple heuristics such as using timestamps to approximate causality, using a white list to preclude unnecessary dependences, trying to distinguish processes that operate on different segments of a file, and separating an execution to segments bounded by two consecutive sockets reads to reduce false dependences.

BEEP [21] showed simple heuristics based approaches are ineffective in practice, mainly due to the asynchronous design of many long-running programs. This work [21] addressed the dependence explosion problem by partitioning a long execution into execution units, which are essentially iterations of an event processing loop.

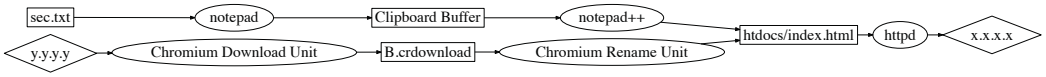
However, BEEP has some fundamental limitations, especially on the Windows platform. First, BEEP requires to insert additional logging commands at a small number of critical places such as event handling loop boundaries to facilitate execution partitioning. Instrumenting COTS Windows executables is not legitimate in many cases and hence prohibitive in the enterprise environment. Secondly, BEEP requires an extensive training phase, in which applications have to run in a runtime profiler which incurs slowdown of two orders of magnitude. Moreover, the same training has to be repeated every time the application is updated, which is a heavy burden for the system administrator. Third, BEEP is built on the default Linux audit logging system, which incurs high runtime and space overhead. Lastly, BEEP cannot detects causal dependences between processes through memory operations (e.g., copy and paste via clipboard).

3. OVERVIEW

Our technique aims to address most of the aforementioned limitations. The technique does not require any binary instrumentation nor runtime training. Instead, it leverages *Event Tracing for Windows* (ETW), which is a build-in logging facility for any Windows system, which allows user to configure and log system level events with negligible over-



(a) A simplified graph generated by traditional methods. The original graph contains 165 files, 9982 sockets and 75 processes.



(b) Insider Attack graph generated by our technique.

Figure 1: Insider Attack: diamond, box, and oval represent communication channel, file and process, respectively.

head. Given an application binary, our technique analyzes the binary *statically*, with the help of one (or a few) ETW log(s) for the application, to construct a log parsing model. The model allows us to partition *any* ETW log of the same application to units, each unit containing the system events belong to an autonomous sub-execution (e.g., the handling of an independent external request). Causal graphs are constructed from the partitioned ETW logs, allowing forensic analysis. Inter-process memory based communications are handled by wrapping the corresponding Windows APIs. The wrappers emit special system events that are captured and handled by the underlying ETW system.

Fig. 2 presents the overview of our approach:

Step 1. Log Collection: The user simply uses ETW to collect an execution log for normal system execution that includes all the commonly used applications. ETW allows us to collect system-level events such as system calls. We extend ETW to additionally record important non-system events such as copy and paste through clipboard and their corresponding stack frames.

Step 2. Prefix Analysis: In this step, for each application, we take its executable binary and extract the log entries belonging to its execution from the ETW log(s) we collected. Then the application logs are analyzed to identify the *target function*, which is a function that contains the event processing loop, an autonomous execution unit that handles independent external event/request. This analysis is based on the observation that the event handling loop of an application is usually present in some top level function, which means that it must appear as the common *prefix* of the stack frames of all the events occurring inside the event handling loop, which corresponds to the dominant phase of execution. This technique is named *prefix analysis*.

Step 3. Program Analysis: Third, we analyze the application binary to build a model. In particular, we disassemble the program binary and analyze the target function. Some Windows binaries are difficult to disassemble or analyze [6] due to their characteristics, such as packing, runtime self-modification, and extensive use of function pointers. For those programs, the models are directly constructed from the ETW logs of the applications. Note that ETW logs are oblivious to the aforementioned obfuscation due to their dynamic nature. Intuitively, the reader can consider the models constructed in this phase are automata that allow parsing ETW logs to units.

Step 4. Unit Recognition: Fourth, we use the models from the previous step to partition any runtime ETW logs to units. At the top level, the models allow us to partition the log entries belonging to individual applications to three phases: the starting phase, closing phase, and event handling phase. The starting phase initializes program resources and configuration, and the closing phases releases resources before the program termination. The models further parse the log entries in the event handling phase to log units, each denoting an iteration of the event handling loop. The aforementioned steps 2, 3, and 4 are tightly connected. We will discuss detail of these steps in Section 4.2.

Step 5. Causality Analysis: In this step, the *unit-based log* from the previous step is analyzed to generate an accurate causal graph. In the graph, a process is decomposed into many autonomous units with the corresponding partitions of the objects (e.g., files, sockets) accessed by each unit. An output event is considered dependent on the preceding input events in the same unit instead of the preceding input events in the whole execution. As such, dependence explosion can be avoided. Fig. 1(b) presents the graph generated in this step for the insider attack example. Now our graph has a much smaller number of nodes and edges than the graph generated by other approaches that are not unit based. Still it precisely and concisely captures the attack path. Note that although theoretically BEEP can produce casual graphs of similar quality, its requirement of binary instrumentation and extensive training, and the entailed overhead make it unsuitable for deployed Windows systems.

Step 6. Applications: Our technique is general and can be used in different applications that require accurate logging. In this paper, we discuss two of these applications: (1) generating attack analysis reports as shown in this section, and (2) removing unnecessary log entries (Section 4.4) for better space efficiency.

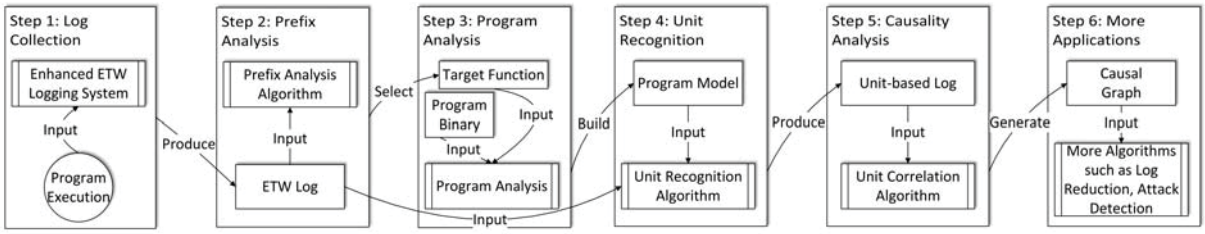


Figure 2: Approach Overview.

```
1  Timestamp:      0x20462fbd1fb                    (1)
2  EventType:      WinsockTcpIpReceive
3  -Event Detailes-----------------------------------
4  TransferSize:   8000                             (2)
5  -Process------------------------------------------
6  Process:        3112 httpd.exe                   (3)
7  Thread:         3948 msvcrt.dll!endthreadex+0x29
8  -Stack Trace--------------------------------------
9    ...(Library and kernel functions)              (4)
10   libapr-1.dll!apr_socket_recv+0x42
11   ...(Application functions)
12   libhttpd.dll!worker_main+0x9c
13   ...(Kernel functions)
```

Figure 3: An example log entry of our enhanced ETW logging system. StackTrace entry is either in the format of `<image>+<offset to image>` or `<image>!<symbol>+<offset to symbol>` depending on the availability of the symbolic information.

4. SYSTEM DESIGN

In this section, we present the details of each component of our system.

4.1 Log Collection

We leverages the *Event Tracing for Windows(ETW)* [1] system that is a build-in event tracing mechanism on all deployed Windows systems. We choose ETW because of its availability (supported by all Windows systems since Windows XP), its capabilities (e.g. supporting thread level tracking, all operations on files/sockets, and stack tracing) and its low overhead (demonstrated by our experimental results in Section 5).

However, ETW events alone cannot support the discovery of all important causal relations. For example, the dependency of copy-and-paste operations are not captured. With such missing dependency, the whole attack path for the insider attack as demonstrated in Section 2 cannot be analyzed. We will shortly present how we solve this problem via the enhancement of monitoring.

Log Format: Fig. 3 shows an example log entry of our logging system. The first block, `(1)`, includes the basic information of the event, such as the timestamp of the event (`Timestamp`), the event type (`EventType`) and so on. The second block, `(2)`, lists the details of the event. The content of this block is specific to the type of the event, so it varies for different event types. For example, for a *FileCreate* event, we will have the name of the file created in the block. The third block, `(3)`, shows what process and thread were running when the event occurred. Lastly, the last block, `(4)`, has the stack trace that ETW collected at the time of the event. Each entry of the stack trace represents a callee function with its upper entry being its caller function.

ETW collects kernel events and the corresponding stack walk events separately. Specifically, when a kernel event occurs, ETW initiates a stack walk request, resulting in a separate stack walk event which contains the stack trace and some basic information such as timestamp and process ID. Our system preprocesses the raw ETW logs, associating a kernel event with the corresponding stack walk event by correlating their process IDs and timestamps. The resulting log is stored in the aforementioned format.

Enhancement: ETW does not cover all OS events. For example in Section 2, we show that ETW cannot trace clipboard operations. We, therefore, enhance ETW by adding the monitoring of non-kernel IPC operations; to monitor clipboard operations, we treat the clipboard buffer as a special file shared by all processes. In particular, we leverage two important observations. First, all read operations on the clipboard file only depend on the most recent write operation. Second, a write operation has no dependency with other clipboard operations. Therefore, we intercept the Windows APIs that access the clipboard to emit special kernel level events that can be trapped and logged by ETW. We note that the overhead of the clipboard monitoring is low since the frequency of copy and paste operations is small in most cases. We envision that if new IPCs are added to Windows in the future, the system can be extended to support them with the same idea.

4.2 Log Analysis

As we have mentioned earlier, a unit is an iteration of the event handling loop. Our ultimate goal is to be able to parse a log file to units such that an output event is only causally related to the preceding input events in the same unit, not the same process. A possible approach is to instrument binaries like in [21] such that special events are emitted and logged at unit boundaries. However, this is impractical for Windows binaries due to the possible legal concerns and the involved complexity in Windows binary instrumentation [6]. Hence, we propose an algorithm that combines log analysis and binary program analysis to derive models that can be used to parse logs to units. First, for an application in the system, we analyze the logs to find the function which contains the event handling loop of the application. Then based on the disassembled function information we construct an automata model used to parse log files to units.

4.2.1 Prefix Analysis

The prior task to build an automata model is to find the function that contains the event handling loop. We call this function a *target function* in this paper. Recall that the execution of a long running process can be divided into three phases:

1. Phase 1 (Prologue): the starting phase that initializes the execution including reading configuration files, allocating memory etc.

2. Phase 2 (Unit): the event handling phase that processes a long sequence of events.

3. Phase 3 (Epilogue): the closing phase that releases all resources and ends the execution.

Units are typically generated only in the second phase. Phase 1 and 3 do not handle external events, although they also generate ETW events in the log file. Hence, we need to filter the log to preclude the events of phase 1 and 3. We observe that phase 1 and 3 are usually the same for different runs of the same program, whereas phase 2 depends on the inputs, and usually generates different ETW events. Thus we just take two different traces of the same program. By comparing the two event sequences, we identify the common event subsequences at the beginning and the end of the log files. However, this may mis-classify some log entries that belong to the second phase to phase 1 or 3. It happens when the first or last a few external events in phase 2 share the same event handler with similar parameters. Also, such small errors do not affect the precision of the models that we generate because the models are eventually generated based on program analysis. When the log file is parsed with the precise models, the misclassification can be corrected, as to be shown shortly. In this step, as long as the majority of the log is formed by log entries from phase 2, we are able to identify the *target function*. After pruning the prologue and epilogue phases, the remaining events are supposed to be spawned from the event handling loop. There are two main observations about the event handling loop: (1) an event handling loop is most likely a top level loop, meaning that it is not nested in any other loop; (2) event handling loop body will receive inputs and/or produce outputs such that it will generate ETW events like socket reads and writes. These two observations help us locate the *target function*. In particular, the *target function* will always be present in the first a few stack frame entries of the phase 2 events. As such, we first compute the common prefix of all the phase 2 event stack frames and consider the user function closest to the end of the prefix as the target function. Due to the nosies from phases 1 and 3, we rank the candidates according to their frequency of appearance, and analyze the caller-callee relationship to find the correct *target function*.

```
1   void main() {
2       init();
3       while(True) {
4   F1:     read_cmd();
5           if (cmd==FileDownload) {
6               if(file ok to download) {
7   F2:             fd = open_file(file_name);
8                   if(open fails)
9   F3:                 errmsg_continue(MSG2);
10                  buf = memory_allocation(size);
11                  while(transfer not done) {
12  F4:                 read_file(fd, buf);
13  F5:                 write_data(socket, buf);
14                  }
15                  memory_free(buf);
16  F6:             close_file(fd);
17              } else
18  F7:             errmsg_continuemsg(socket, MSG3);
19          } else if (cmd==...) {...}
20      } // end while
21      server_exit();
22  }
```

Figure 4: An example of event handling loop: the *main* function of AdvancedFTP. F1-F7 are the labels for syscalls.

```
1  SocketRead:   windows_runtime->main->read_cmd->...
2  FileOpen:     windows_runtime->main->open_file->...
3  FileRead:     windows_runtime->main->read_file->...
4  SocketWrite:  windows_runtime->main->write_data->...
5  FileClose:    windows_runtime->main->close_file->...
```

Figure 5: Illustration of common prefix (windows_runtime→main) and the target function (main) in stack frames of *AdvancedFTP* events.

Fig. 4 present an example of these observations, regarding the target function of a FTP server, *AdvancedFTP*, using a simplified code snippet. The event handling loop starts from line 3. First it reads the command (line 4). If it gets the download command, it first checks if the request is valid (line 6). If so, the FTP server opens the file (line 7), allocates buffer (line 10), and then transfers the file to client (line 11 to 14). After the file is transferred, the server frees the buffer (line 15) and closes the file (line 16). If the user cannot download the file, for example, due to access control, the server returns an error message (line 18). Fig. 5 shows the stack frames of sample events. Observe that the common prefix is *windows_runtime* → *main* and the target function *main* is correctly identified. *Windows_runtime* denotes the sequence of functions invoked when the process started to setup the runtime environment.

4.2.2 Model Construction

Once we determine the target function via prefix analysis, we construct a model for the program's behavior. This model is based on automata and hence can be represented as a regular expression. It describes the possible system behavior for an iteration of the event handling loop. It is later used to parse a log file to units.

Regular Expression Model. The vocabulary of the regular expression is a set of function invocation relations from the target function to a callee. For example, $(target \rightarrow foo)$ means that the function *target* calls the function foo. We use ϵ to represent an empty expression. The other notations such as kleene closure are following the standard rules. For example, the expression $(target \rightarrow foo)(target \rightarrow gee)$ means that *target* first calls function foo, and then calls *gee*. The expression $(target \rightarrow foo|\epsilon)$ means that function *target* may call foo or may not, guarded by some predicate. The expression $((target \rightarrow foo)(target \rightarrow gee))^*$ means that the pattern $(target \rightarrow foo)(target \rightarrow gee)$ may appear 0, 1, or multiple times, denoting a loop.

Identifying Event Handling Loop. Given the *target function*, we need to find the event handling loop. There could be multiple loops within the target function. Fig. 6(left subgraph) shows an example that the target function, *main*, contains multiple loops, which are used to parse input parameters, request and release resources (e.g. memory) and handle events. All of them generate events, but some events will be part of phases 1 or 3. The phase 2 events must belong to the event handling loop. To recover the loop from the phase 2 events, we identify the program counters (PCs) of the invocations to all callees of the target function inside the phase 2 event stack frames. The enclosing loop body of these PCs is the event handling loop.

Model Construction. Once the event handling loop is identified, we construct a model for the loop body via program analysis. In particular, we analyze the intra-procedural program paths inside the loop and identify all the possible

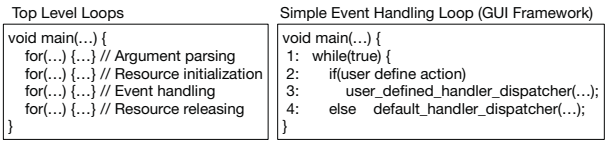


Figure 6: Examples of Top Level Loops.

sequences of function calls that may lead to syscalls and represent them as a regular expression. Our approach relies on existing program analysis techniques. Binary analysis is known hard problem. There are cases that our approach are not able to handle. In this case we treat the whole program execution as one unit so that we do not lose any information during the investigation. Thus the system is not worse than traditional approaches in the worst case. Algorithm 1

Algorithm 1 Model Construction: function *model*

Input: I - instruction sequence
Output: the model for I

```
    // termination condition
1:  if I == {} then
2:      return ϵ
3:  end if
    // call instruction
4:  if I.head is a call instruction then
        // I.head.callee is the callee
5:      return ((F → I.head.callee)*|ϵ) · model(I.tail)
6:  end if
    // conditional jump
7:  if I.head is a conditional jump then
        // CD(n) returns the sequence of instructions directly or transitively control dependent on n
8:      tmodel ← model(CD(I.head^t))
9:      fmodel ← model(CD(I.head^f))
10:     if I.head^t denotes a loop predicate then
11:         m ← (tmodel)*
12:     else
13:         if I.head^f denotes a loop predicate then
14:             m ← (fmodel)*
15:         else
16:             m ← (tmodel | fmodel)
17:         end if
18:     end if
19:     return m · model(I.tail − CD(I.head^t) − CD(Ihead^f))
20: end if
    // other instructions
21: return model(I.tail)
```

describes this procedure. It takes the binary instruction sequence representing the loop body as an input and produces the regular expression. It relies on precomputed control dependence information. Note that we only need to disassemble the target function and compute its control dependences. The computation is recursive.

In the algorithm, I denotes the instruction sequence with $I.head$ the first instruction and $I.tail$ the remaining sequence. The method returns ϵ when I becomes empty (lines 1-2). When the first instruction is a function cathe epsilon means that the function invocation may or may not lead to any syscall such that it may or may not be present in the stack frame of an ETW event. The kleene closure means that there may be loops inside the callee function such that the invocation appears in stack frames of a consecutive sequence of ETW events. If $I.head$ is a conditional jump, the algorithm recursively computes the true branch model and the false branch model, leveraging the precomputed control dependence (CD) information (lines 8-9). Note that if one of the branches is empty, the corresponding CD set is empty and hence the model is simply ϵ. It is possible that the conditional jump denotes a loop predicate. Hence, depending

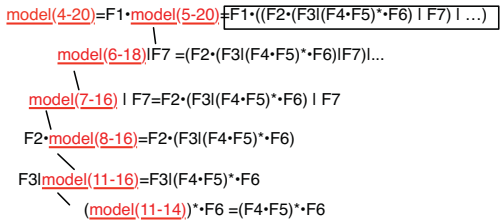


Figure 7: Model construction for the example in Fig. 4. An arrow shows that the *model()* invocation at the source (of the arrow) is divided into the computation at the destination. We use *F1* to denote $main \rightarrow F1$ to save space.

on if the true or false branch leads to the loop body, the true branch model or the false branch model is put into a kleene closure, respectively (lines 11 and 14). Otherwise, the model is the disjunction of the two branch-models (line 16). The model for the predicate and the branches guarded by the predicate is concatenated with the model for the continuation (line 19). If $I.head$ is any other instruction, the model is simply that of $I.tail$.

Fig. 7 shows the model construction procedure for the FTP example in Fig. 4. The event loop body is in lines 4-20. So the *model*() function is invoked with 4-20, as shown on the top. Since line 4 denotes a function call that may lead to syscall, the model is hence $main \rightarrow F1$, or $F1$ for short, concatenated with the model of lines 5-20. Since line 5 is a predicate (or, conditional jump at the binary level), the model of 5-20 is the disjunction of the model of 6-18 and the model of 19. The division continues. Note that the model of 11-16 is the model for the loop (line 11-14), which is $(F4 \cdot F5)^*$, concatenated with the model for lines 16, which is $F6$. Eventually, the entire regular expression is computed in the box, which describes the possible ETW event stack frame sequences. Note that we know memory library functions such as *memory_free()* is of no interest to our analysis so that they are not part of the regular expression vocabulary.

There may be cases that the body of the event handle loop is extremely simple, in which the program calls another complex function to handle the different I/O operations. As such, the model of the event handling loop alone may not be sufficient. We may need to further analyze callee functions inside the loop to construct more informative model. A typical example is graphical user interface (GUI) applicaitons built on top of specific GUI libraries. A typical case is shown in Fig. 6. Within the event handling loop, the library first checks if the user application defines a dispatcher (line 2). If so, it will use the application dispatcher. Otherwise, it uses the default dispatcher that does some basic event processing. Assume we only construct the model from the event handling loop, which is $(main \rightarrow user\_dispatcher) \mid (main \rightarrow default\_dispatcher)$. Further assume the user application does provide its own dispatcher so that the handling of various external events actually happens inside the user dispatcher. Note that handling an external event may lead to multiple ETW events. Assume two external events are sent and processed by the application, the ETW events belong to two units but their ETW event stack frames are not distinguishable based on the simple model. Therefore, our algorithm will recursively disassemble and analyze callee functions when the constructed model is not sufficiently informative. We use the size of the regular expression as an indicator, and we will continue doing that if the current generated model is not sufficient.

4.3 Log Partitioning

Each model is a regular expression describing the possible syscall sequences within the event handling loop of an application. Given an ETW log that contains system events from all the active applications in the system, our technique parses the log to units, each representing a legitimate sequence from the regular expression of the corresponding application. In particular, for each ETW event entry, we identify the application from the process ID. The corresponding model is used to parse the entry. When the end of a legitimate sequence is reached (according to the model), the end of a unit is reached and a new unit starts, indicating the end of an event loop iteration and the start of a new iteration.

For example, consider the model shown in Fig. 7. For a sequence $F1{\cdot}F2{\cdot}F4{\cdot}F5{\cdot}F6{\cdot}F1...$, we follow the model from $F1$ to $F6$ which indicates the end of the loop. Then we know that one unit ends, and a new unit starts from $F1$ again. There is a possible challenge that some functions in the loop may not generate events. For many library functions with well-known semantics, like lines 11 and 16 in Fig. 4, we can remove them from our model. However for many other functions, we are not sure if they will lead to syscalls. Thus we conservatively include those functions in our model. In the mean time, we also place an empty symbol ϵ as part of the regular expression describing the function behavior (line 5 in Algorithm 1), indicating that the function may not lead to any syscall. The possibility that a function may not generate a syscall event poses another challenge: the generated automata is non-deterministic, meaning that it may parse the log differently, yielding different unit partitions. To address the problem, we use a simple heuristic that is very effective in practice. We always parse the longest possible subsequence of events as a unit.

4.4 Dependency Analysis and Log Reduction

After units are recognized, we can construct a graph to denote the causality between events. In particular, a node is introduced to denote a unit or a system object like a network session or a file object. If a unit receives input from some system object, an edge from the object to the unit is introduced. If a unit writes/updates an object, an edge from the unit to the output object is also introduced. As such, an output event is only connected (and hence dependent) on the preceding input events within the same unit. Note that the event timestamps are also annotated on the edges to determine the temporal order.

One of the key challenges when applying log-based attack analysis in practice is to store the large volume of log. According to existing work [20, 15, 8], audit logs grow at the rate of 0.8 GB $\sim$ 3.2 GB per day with moderate workload. The key contribution of our technique is the identification of precise causal dependences between events. These precise causal dependences allow us to accurately determine the events that are unnecessary or redundant for forensics analysis. Intuitively, if an event is not causally related to any live object in the system (e.g., a file) or any destructive behavior that has global effect such as deleting a non-temporary file, it is not useful in any (future) forensics analysis and hence can be removed from the log file to save space. We first leverage algorithms from existing work [20] to remove dependency-free objects, which are *created and deleted by the same execution unit and never accessed by other units.* For example, when a FTP client tries to download files from a FTP server, the server creates a number of temporary files during file downloading. However those temporary files are deleted shortly in the same execution unit and never affect future system behavior. Web browsers also intensively use temporary files to store remote resources, but they never affect the system. Thus we can safely delete events related to them without degrading forensic analysis effectiveness. We also observe that ETW generates a sequence of events of the same type that contains redundancy. For example, when we open a large file in *NotePad++*, the editor continuously reads a small portion of the file into its memory buffer. Those `FileRead` events can be merged into one ETW event without causing any degradation. We evaluate the effectiveness of our log reduction technique using various applications. The result is presented in Section 5.3.

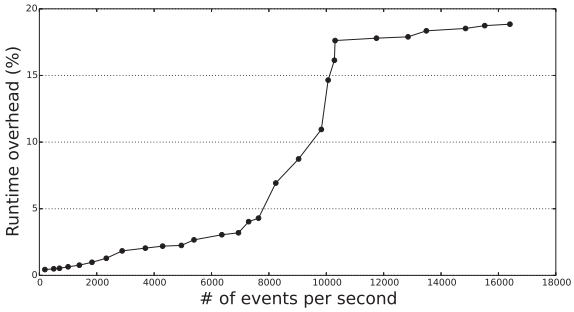


Figure 8: Event Tracing Overhead.

5. EVALUATION

We will present our evaluation results in this section. All experiments were conducted on Intel i7-3880 CPU with 12GB of RAM running Windows Server 2008 R2 64-bit. We use different types of Windows applications for the evaluation, including default applications shipped with Windows such as *Paint* and *Notepad*, third party user interactive (UI) applications such as *DrawTool*, *notepad++*, and *Chromium*, and also server applications such as *Apache* web server.

5.1 Event Tracing Overhead

Benchmark: In the first experiment, we measure the runtime overhead of the event tracing system ETW. The runtime overhead of ETW hinges on the number of events generated in a unit of time, which depends on the ETW configuration and the workload. Since our configuration is fixed, we measure overhead with different workloads. We use the *Apache* web server and multiple *ApacheBench (ab)* clients to send HTTP requests simultaneously. We generate various numbers of requests per second. Fig. 8 shows the ETW runtime overhead with the different numbers of requests from the *ApacheBench* clients. The X-axis in this graph represents the number of ETW events per second, and the Y-axis shows the runtime overhead. We observe that runtime overhead is smaller than 10% when we have under 10,000 events per second. It goes up to 18% under a heavy workload (over 15,000 events per second).

Regular systems: The previous experiment shows that the runtime overhead is directly related to the number of events in a time interval. In this experiment, we collect workloads from different machines to observe typical event generation patterns. We collect ETW logs from three different end-user machines and two different server systems. Each system has been used for different purposes. For example, user 1

Instance	# of Event	Time(s)	# Event per sec
User 1	1,391,555	10,460	133.04
User 2	1,412,453	10,959	128.89
User 3	173,425	941	184.30
Server 1	842,879	2,566	328.48
Server 2	1,897,303	3,352	566.02

Table 1: Event generation in regular systems

is a software developer who heavily uses a text editor, the *Chromium* web browser and a complier to develop software and write document. User 2 is another software developer that uses different software and shows different use patterns. User 3 is a regular user who mainly uses *Chromium* for web surfing. Server 1 and 2 run the *Apache* web server to host different sets of web pages.

Table 1 presents the number of ETW events generated by each system. The second column shows the total number of event entries and the third column presents the period of time in which the events were generated, and the fourth column shows the number of ETW events per second. In this experiments, we observe that the regular user and the server systems generate 130 $\sim$ 600 ETW events per second, which only incurs less than 1% of runtime overhead. In most cases, the systems generate 100 $\sim$ 5,000 events per second, causing about 0.4% $\sim$ 2.5% runtime overhead. This is in accordance with previously reported results [13].

5.2 Model Construction

Table 2 shows the result of model construction. The second column shows the depth of the event handling loop. A depth of one means that the loop is on the top level, and we can build the event model by analyzing only the top-level loops. A depth of two means that the event-handling loop is nested within another loop, and we need to analyze the second level of loops to find the target function. The third column shows the number of functions we analyzed while constructing the model. The number of functions analyzed include the target function and sometimes the callees of the target function. The latter occurs when the target function is very simple and the event processing logic mainly resides in the callees of the target function. For those cases, our technique automatically analyzes the callees of the target function. The forth column shows the function invocations in the model (i.e., terminals in the regular expressions). The fifth column shows the number of transitions in the model (e.g., concatenations and kleene closures). They essentially denote the complexity of control flow. Observe that the generated models are quite complex, representing the possible sequences of system events.

From this experiment, we also observe the number of events per unit varies a lot for different types of programs. For example, the FTP client program, *NetFTP*, shows typical characteristics of a network application. In this program, each execution unit handles a separate request. If the request from the user is file download, the unit receives data from the server and writes it into a local file. The number of file write events could be different, depending on the file size. Another popular type of programs is UI applications. The image editing program, *DrawTool*, shows a representative pattern. It frequently accesses temporary files to support undo/redo and recover operations. In this program, the last event in an execution unit is a *save file* operation. In a unit (delimited by two *save file* operations), multiple *FileWrite* events on temporary files are generated.

Application	Depth	Function analyzed	Model	
			Function	Transition
TextTransfer	1	1	48	72
Chromium	1	4	124	167
DrawTool	1	1	78	104
NetFTP	1	2	64	92
AdvancedFTP	1	2	56	88
Apache httpd	1	2	36	54
IE	2	3	94	144
Paint	2	3	82	132
Notepad	2	3	64	92
Notepad++	1	1	144	208
SimpleHTTP	1	2	22	34
Sublime Text	1	1	132	178

Table 2: Model Construction.

5.3 Log Reduction

We evaluate performance of the log reduction technique that we have discussed in the Section 4.4. Table 3 shows the results. The second column presents a number of ETW events. The third column shows the number of removable events related to dependency-free objects and their percentage. The fourth column presents the number of removable events classified as redundant and their percentage. The fifth column shows that our log reduction algorithm reduces 96.85% of the original logs on average. The sixth and seventh columns show the number of recognized units in the logs described in column 2 and column 5 respectively, using the generated models. The eighth and ninth columns represent the average number of events for each unit.

5.4 Attack Forensics

In this experiment, we study the effectiveness of our technique in attack forensics. We emulate popular attack scenarios and collect attack logs. We then construct the attack causal graphs, using both the original logs and the reduced logs. In the first scenario, *mis-configured server*, an administrator misconfigured the *httpd* server such that some confidential files were made publically accessible. The investigator wanted to identify what confidential information were accessed, and by whom. Thus, the investigator performs a forward reachability query on the causal graph, starting from the confidential files.

The second scenario is a *phishing attack*. The victim received a phishing email and he clicked a malicious link in it. The web browser in the victim's system opens the phishing web site. Then the victim downloaded a program from the web page, which turns out to be a malware. Assume this malware is later discovered by an anti-virus tool. The investigator wants to trace to the source the attack and understand the behavior of the malware. Thus the investigator performs both forward and backward reachability queries from the malware process.

The third scenario is *information leak*, which was already discussed in Section 2. To understand behavior of the inside attacker, a forward reachability query is performed from the secret file, and a backward query is performed from the IP address to which the secret file was downloaded.

In the fourth scenario *spyware*, a spyware is triggered. The spyware detects if the system has *Firefox* or *Chrome*. If one of them exists, the spyware tries to steal files in which the

Application	# of Events / Ratio				# of Units		# of Events per unit	
	Original	Dependency-free	Redundant	Final	Original	Final	Original	Final
TextTransfer	316	310 / 98.10%	0 / 0.00%	6 / 1.90%	2	2	158.00	3
Chromium	102,206	61,193 / 59.87%	36,834 / 36.04%	4,179 / 4.09%	255	248	400.81	16.85
DrawTool	15,438	13,382 / 86.68%	1,982 / 12.84%	74 / 0.48%	11	11	1,403.45	6.73
NetFTP	10,621	8,909 / 83.88%	1,132 / 10.66%	580 / 5.46%	8	8	1,327.63	72.5
AdvancedFTP	1,615	1,161 / 71.89%	411 / 25.45%	43 / 2.66%	3	3	538.33	14.3
Apache httpd	37,171	27,035 / 72.73%	8,084 / 21.75%	2,052 / 5.52%	46	46	808.07	44.61
IE	29,969	12,983 / 43.32%	14,711 / 49.09%	2,275 / 7.59%	10	10	2,996.90	227.5
Paint	7,085	6,772 / 95.58%	235 / 3.32%	78 / 1.10%	28	28	253.04	2.78
Notepad	11,704	8,506 / 72.68%	3,168 / 27.07%	30 / 0.26%	6	6	1,950.67	5.00
Notepad++	5,516	4,976 / 90.21%	404 / 7.32%	136 / 2.47%	9	9	612.89	15.11
SimpleHTTP	779	559 / 71.76%	180 / 23.11%	40 / 5.13%	13	13	59.92	3.08
Sublime Text	30,372	24,419 / 80.40%	5,637 / 18.56%	316 / 1.04%	11	11	2,761.09	28.73

Table 3: Effectiveness of Log Reduction.

Scenario	# of events			# of nodes			# of edges			Correctness	
	Before	After	Ratio	Original	Unit	GC	Original	Unit	GC	Backward	Forward
Mis-configured server	986,563	85,042	8.62%	173	10	10	204	10	10	-	Match
Phishing attack	523,385	44,593	8.52%	573	21	21	693	32	32	Match	Match
Information leak	1,947,485	260,857	13.39%	10,222	11	11	20,532	10	10	Match	Match
Spyware	1,284,748	102,523	7.98%	9,282	9	9	11,244	8	8	Match	Match

Table 4: Attack scenarios summary with original log and reduced log.

browser stores user passwords. If both *Firefox* and *Chrome* are installed in the system, it would randomly choose one to steal the user passwords. Assume the user detects the presence of the spyware and the investigator performs a forward analysis starting from the spyware process to understand its behavior, and a backward analysis to find its source.

The query results are presented in Table 4. The second column shows the number of ETW events in the logs, and the third column presents the number of ETW events after we reduce unnecessary and redundant events. The forth columns shows the reduction rate we achieved in each scenario. The next three columns show the number of the nodes in the generated graph after applying existing non-unit based approaches, our unit based approach, and the log reduction approach, respectively. Then in the next three columns, we show the number of edges shown in the generated graph using these three methods. The graphs we use are the combinations of the forward and backward query results. The results show that the unit based approach can significantly reduce the size of the graph, and make it easier to investigate. And our log reduction method does not affect the effectiveness. The last two columns show the correctness of the graphs generated (after log reduction). We manually compare the results with our prior knowledge of the attacks. The evaluation shows that we can correctly uncover all the attack behavior.

6. RELATED WORK

There exists a line of work in tracking system dependence using system-level audit logs for attack analysis [9, 15, 11, 2, 7, 8, 16, 14, 29, 24, 5]. These approaches use backward and forward tracking to locate the entry point of an attack and to identify the damage happened to a victim system. However, these techniques may suffer from imprecision caused by dependence explosion, where an event is unnecessarily dependent on too many other events and the corresponding causal graph is excessively large for human inspection. Our technique complements these techniques by partitioning a process execution to fine-grain units to avoid a dependence explosion problem.

BEEP [21] divides a long-running process into fine-grain execution units to mitigate the dependence explosion problem. BEEP pro-actively analyzes and instruments application binaries, that requires a number of test runs conducted by a user. In this paper, we propose log and binary analysis techniques that provide fine-grain units without any instrumentation or beforehand analysis.

There are also different efforts try to mitigate the dependence explosion problem with additional system level information. Sitaraman *et al.* [25] proposes a technique that additionally logs file offsets for file read/write system calls to provide more accurate file dependences. In [19], besides system call, they also record accessed memory page for each process. Although it provides better accuracy, the drawback to this approach is the runtime overhead. Detecting and logging page-level memory access incurs high run-time overhead. Moreover, it introduces false dependences due to the limitation of the page level granularity.

Dynamic information flow tracking and taint analysis techniques [12, 10, 24, 28, 26, 22] have been proposed to track information propagation in the runtime to prevent information leak or zero-day attacks. Our system is designed for forensics, and comparing with these systems, we have less runtime overhead.

In recent years, significant progress has been made on log-based attack detection techniques [17, 18, 3]. Other than the ones we discussed in Section 2, there are still some interesting works. These approaches use system level logs to detect malicious behaviors in the system. There are also machine learning based approaches [4, 23, 27] to detect anomalies from system execution events. LogGC [20] proposed garbage-collectable audit logging system by removing unnecessary or duplicated events from system logs without affecting accuracy. Our technique complements these techniques by providing accurate causalities without affecting compatibility or practicality.

7. CONCLUSION

We develop a low cost Windows audit logging system, which collects system level events and the corresponding stack frames with very low runtime overhead. The system does not require any application instrumentation. The events and stack frames are analyzed and partitioned into units to support accurate causual analysis. The generated attack causal graphs are precise and concise. The high-quality causal graphs also enable highly effective garbage collection, which reduces the space consumption of audit logs by orders of magnitude.

8. ACKNOWLEDGMENTS

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 award 1409668, ONR under constract 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.

9. REFERENCES

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

[2] Ammann, P., Jajodia, S., and Liu, P. Recovery from malicious transactions. *Knowledge and Data Engineering, IEEE Transactions on* (2002).

[3] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS'14*.

[4] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., and Ernst, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. FSE'11.

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

[6] Egele, M., Woo, M., Chapman, P., and Brumley, D. Blanket execution: Dynamic similarity testing for program binaries and components. Usenix Security'14.

[7] Goel, A., Feng, W.-c., Feng, W.-c., and Maier, D. Automatic high-performance reconstruction and recovery. *Computer Networks 51*, 5 (2007), 1361–1377.

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

[9] Hasan, R., Sion, R., and Winslett, M. Preventing history forgery with secure provenance. *ACM Transactions on Storage (TOS) 5*, 4 (2009), 12.

[10] Jee, K., Portokalidis, G., Kemerlis, V. P., Ghosh, S., August, D. I., and Keromytis, A. D. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *NDSS* (2012).

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

[12] Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. libdft: practical dynamic data flow tracking for commodity systems. VEE '12.

[13] Kim, C. H., Rhee, J., Zhang, H., Arora, N., Jiang, G., Zhang, X., and Xu, D. Introperf: transparent context-sensitive multi-layer performance inference using system stack traces. SIGMETRIC'14.

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

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

[16] King, S. T., Mao, Z. M., Lucchetti, D. G., and Chen, P. M. Enriching intrusion alerts through multi-host causality. In *NDSS* (2005).

[17] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. Effective and efficient malware detection at the end host. Usernix Security '09.

[18] Kolbitsch, C., Kirda, E., and Kruegel, C. The power of procrastination: Detection and mitigation of execution-stalling malicious code. CCS '11.

[19] Krishnan, S., Snow, K. Z., and Monrose, F. Trail of bytes: efficient support for forensic analysis. CCS'10.

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

[21] Lee, K. H., Zhang, X., and Xu, D. High accuracy attack provenance via binary-based execution partition. In *NDSS* (2013), Citeseer.

[22] 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'09.

[23] Nagaraj, K., Killian, C., and Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. NSDI'12.

[24] Newsome, J., and Song, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS* (2005).

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

[26] Tak, B. C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B., and Chang, R. N. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. USENIX'09.

[27] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. Detecting large-scale system problems by mining console logs. SOSP'09.

[28] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: capturing system-wide information flow for malware detection and analysis. CCS '07.

[29] Zhu, N., and cker Chiueh, T. Design, implementation, and evaluation of repairable file service. DSN'03.