MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation

Yonghwi Kwon[1], Fei Wang[1], Weihang Wang[1], Kyu Hyung Lee[2], Wen-Chuan Lee[1], Shiqing Ma[1], Xiangyu Zhang[1], Dongyan Xu[1], Somesh Jha[3], Gabriela Ciocarlie[4], Ashish Gehani[4], and Vinod Yegneswaran[4]
[1]Department of Computer Science, Purdue University
{kwon58, feiwang, wang1315, lee1938, ma229, xyzhang, dxu}@cs.purdue.edu
[2]Department of Computer Science, University of Georgia
kyuhlee@cs.uga.edu
[3]Department of Computer Sciences, University of Wisconsin-Madison
jha@cs.wisc.edu
[4]SRI International
{gabriela, gehani, vinod}@csl.sri.com

Abstract*—In this paper, we develop a model based causality inference technique for audit logging that does not require any application instrumentation or kernel modification. It leverages a recent dynamic analysis, dual execution (LDX), that can infer precise causality between system calls but unfortunately requires doubling the resource consumption such as CPU time and memory consumption. For each application, we use LDX to acquire precise causal models for a set of primitive operations. Each model is a sequence of system calls that have inter-dependences, some of them caused by memory operations and hence implicit at the system call level. These models are described by a language that supports various complexity such as regular, context-free, and even context-sensitive. In production run, a novel parser is deployed to parse audit logs (without any enhancement) to model instances and hence derive causality. Our evaluation on a set of real-world programs shows that the technique is highly effective. The generated models can recover causality with 0% false-positives (FP) and false-negatives (FN) for most programs and only 8.3% FP and 5.2% FN in the worst cases. The models also feature excellent composibility, meaning that the models derived from primitive operations can be composed together to describe causality for large and complex real world missions. Applying our technique to attack investigation shows that the system-wide attack causal graphs are highly precise and concise, having better quality than the state-of-the-art.

I. INTRODUCTION

Cyber-attacks are becoming increasingly targeted and sophisticated [2]. A special kind of these attacks, called Advanced Persistent Threat (APT), can infiltrate into target systems in stages and reside inert for a long time to remain undetected. It is important to trace back attack steps and understand how an attack unfolds [4]. In the mean time, identifying the entry point of the attack and understanding the damage to the victim can be critical to recovering the victim system from the intrusion and also preventing future compromises.

Causality analysis techniques [25], [16], [24], [26], [29] are widely used in attack investigation. They analyze audit logs generated by operating system level audit logging tools (e.g., Linux Audit [17], Event Tracing for Windows [38], and DTrace [13]) and correlate system events, e.g., system calls (syscalls) to identify causal relations between system subjects (e.g., processes) and system objects (e.g., files, network sockets). Such capability is particularly important in cyber-attack investigation where causality of malicious events reveals attack provenance. For example, when an attacker exploits vulnerabilities and executes malicious payloads, causality analysis can identify such vulnerable interfaces including input channels that accept malicious inputs from the user or the network. Moreover, given a set of malicious or suspicious events, it can identify all the events that are causally related to the given set of events. Essentially, these events depict the source of the attack and/or the damage induced by the attacker. However, syscall based analysis has a major limitation: dependence explosion [32]. For a long-running process, an output event (e.g., creating a malicious file) is assumed to be causally related to all the preceding input events (e.g., file read and network receive). This conservative assumption causes significant false causal relations.

Some recent works [32], [37], [35], [36] focus on collecting enhanced information at run-time to avoid dependence explosion and enable accurate attack investigation. For instance, BEEP [32] and ProTracer [37] train and instrument long-running applications to capture information of fine-grained execution units in addition to syscalls. MPI [36] asks the user to annotate important data structures in applications' source code to enable semantic aware execution partitioning. Additionally, Bates at el. [6] propose a general provenance-aware framework called Linux Provenance Module (LPM) that allows users to define custom provenance rules. The major hindrance of these techniques in practice is their requirements of changing end-user systems, such as instrumenting user applications, installing new runtime support, kernel modules, and even changing the kernel itself.

Taint analysis [22], [21], [20] is another approach that can track causal relations (e.g., information flow) between system

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23306
www.ndss-symposium.org

components (e.g., memory objects, files, and network sockets). However, whole system tainting is too computationally expensive (over 3x slow down [19], [39]) to be deployed on production systems. Additionally most taint analysis techniques cannot handle implicit flow, resulting in false-negatives.

In this paper, we propose MCI, a novel causality inference technique on audit logs. Our technique *does not require any changes on the end-user system, nor any special operations during system execution*. The end-user only needs to turn on the audit logger shipped with the operating system (e.g., Linux Audit, Event Tracing for Windows, and DTrace). If the user detects a security incident, she only needs to provide the syscall log and program binaries from the victim system (or a disk image) to a forensic expert.

In off-line attack investigation, which is often done by the forensic expert, MCI precisely infers causality from a given system call log by constructing causal models and parsing the log with the models. Fig. 1 shows a high level overview of how MCI works. MCI consists of two phases: (1) causality annotated model generation, and (2) model parsing. First, MCI generates causal models by leveraging LDX [31] which is a dual-execution based system that can infer causality by mutating input syscalls and then observing output changes. In this phase, MCI takes two inputs: a program binary and typical workloads. MCI's model constructor automatically runs LDX and analyzes its results to construct models. Models are expressive and capable of representing fine-grained dependencies including invisible at the syscall level (e.g., dependencies induced by memory operations). The models can be pre-generated (for widely used applications) or generated on demand after an incident. Second, during investigation MCI identifies causal relations between events in a given syscall log collected from a victim system by parsing the log with the models. The derived precise dependencies are critical for attack investigation. In summary, we make the following contributions:

- We propose a novel technique for precise causality inference that directly works on audit logs without requiring any changes or setup on end-user systems. We only require program binaries and the audit log from the victim system after the incident.
- We perform a comparative study using a real-world example to illustrate the merits and limitations of existing approaches.
- We propose to leverage LDX [31] to identify fine-grained causality from program execution. Using the generated causality information, we construct causal models annotated with fine-grained dependencies. We study the model complexity needed to describe causalities in audit logging.
- We develop a novel model parsing algorithm that can handle multiple model complexity levels and substantially mitigate the ambiguity problem inherent in model based parsing.
- We perform thorough evaluation of MCI on a set of real-world applications. The results show that the generated models can recover causality with close to 0% FP and FN for most applications and the worst FP rate 8.3% and the worst FN rate 5.2%. Model construction and model parsing have reasonable overhead and scale to week-long and even month-long workloads. Applying MCI to attack investigation shows that our models have very nice composibility such that small models can be composed together to describe complex system-wide attack behaviors. Our attack causal graphs are even more precise than those generated by a state-of-the-art system [32].

II. BACKGROUND AND MOTIVATION

In this section, we use an insider information leak attack case to illustrate the limitations of existing attack provenance analysis techniques, and then to motivate our work.

A. Motivating Example

We use a data exfiltration of confidential company data by an employee. Insider attacks are the dominant reason for data breach incidents in 2016 [1], [18].

Assume John is a project manager who has access to confidential data. John was bribed by a competitor company and attempts to breach some confidential data. However, John's company forbids copying data to removable media such as USB stick. Furthermore, the company inspects all incoming/outgoing network traffic via deep packet inspection (DPI) [30], [44], [45] to prevent exfiltration of confidential data and to block malicious network traffic from outside. To bypass the packet inspection, John decides to use the GPG encryption algorithm [27] to encrypt data before sending it.

GnuPG Vim plug-in. To use GPG encryption, John installed a `Vim` plug-in `GnuPG` [7], which enables transparent editing of gpg encrypted files. When he opens a file encrypted by gpg [27] which is an encryption utility supported by most operating systems with the GNU library (e.g., Linux, FreeBSD, and MacOS), the `GnuPG` plug-in automatically decrypts and passes the decrypted data to `Vim` so that the user can edit the contents of the encrypted file. The plug-in automatically encrypts the contents when the user saves the gpg file.

Attack Scenario. John uses `Vim` equipped with the `GnuPG` plug-in to open three confidential files, *data1, data2*, and *data3*. He also opens *out.gpg* in order to store confidential data in an encrypted format. Then he copies a few lines from *data2* using the `Vim` command `'v'` to select characters and `'y'` to copy them to the clipboard buffer (i.e., `Vim`'s default register). Then he finds out the information in *data3* is more up-to-date. He thus copies lines from *data3* that overwrite the contents from *data2*. Later, he pastes the copied lines to *out.gpg*, saves the file in an encrypted format and terminates `Vim`. Note that, when he saves *out.gpg*, the `GnuPG` plug-in actually creates a new file (inode:8) and renames it to *out.gpg* so that the original *out.gpg* file (inode:4) is replaced by a new file (inode:8). Observe that the inode numbers of the original *out.gpg* file and the new file are different. Finally, he sends the encrypted *out.gpg* to a server outside the enterprise network.

This data breach incident is later detected, and a forensic analysis team starts to investigate the incident. Now, we introduce existing causal analysis based forensic techniques and discuss how they work on this attack.

B. Existing Approaches and Limitations

System Call based Analysis. Most causal analysis techniques use syscall logging tools to record important system events at

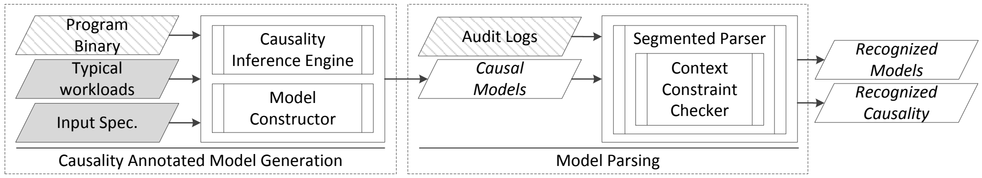


Fig. 1. Overview of MCI's off-line causality inference. Audit Logs and Program Binaries are provided from the end-user, workloads and input specifications are generated by an attack investigator (e.g., a forensic expert), and other components are automatically generated by MCI.

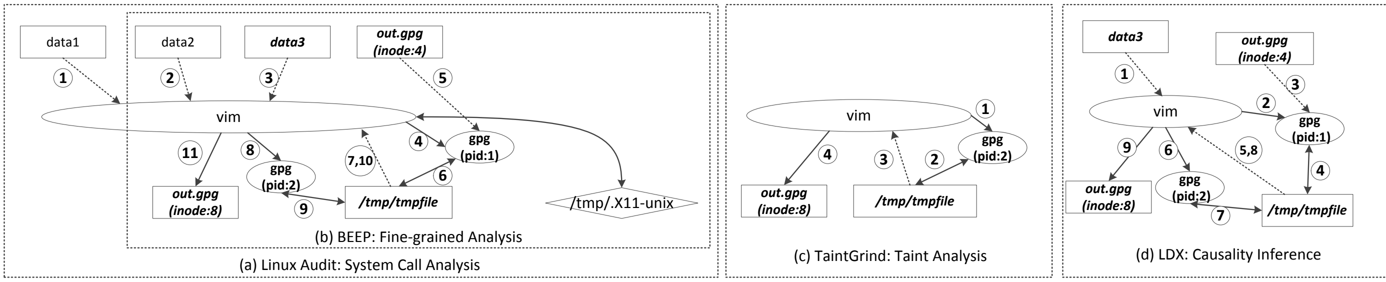


Fig. 2. Motivating Example: Insider theft breaches confidential data using `VIM` and `gpg`

runtime and then analyze recorded events to identify causal relations between system subjects (e.g., process) and system objects (e.g., file or network socket). Syscall logging tools are shipped with most operating systems. For example, Linux Audit [17] is a default package in Linux and MacOS distributions, DTrace [13] is available in FreeBSD, and Event Tracing for Windows (ETW) [38] comes with Windows.

Syscall based analysis has been studied in a number of works [25], [16], [24], [26], [29]. For instance, BackTracker [25] and Taser [16] propose backward and forward analysis techniques in order to analyze syscall logs and construct causal graphs for effective attack investigation. The constructed causal graphs show system subjects and objects that involved in attacks, and their causal relations.

Fig. 2-(a) shows a provenance graph generated from the syscall log collected during the data breach incident discussed in the previous section. To understand the incident in detail, a security analyst first identifies the *out.gpg* file (inode:8) which contains confidential data. Then the analyst finds the system components that are causally related to the file from the graph in the backward direction (time-wise). Observe that it was `Vim` that wrote the file (11). Before that, `Vim` read */tmp/tmpfile* (10) which was written by "gpg" (9). The "gpg" process (pid:2) was forked by `Vim` (8). Before the fork, `Vim` read */tmp/tmpfile* (7) which was written by another "gpg" process (pid:1) (6). "Gpg" previously read the original *out.gpg* file with a different inode number (inode:4) (5) and the "gpg" process (pid:1) was forked by `Vim` (4) as well. There are also other files that `Vim` read, including *data3* (3), *data2* (2), and *data1* (1).

Note that Fig. 2-(a) contains many false dependencies such as dependencies between the `Vim` process and files *data1*, *data2*, and */tmp/.X11-unix* which is a socket for XWindow. The coarse-granularity of processes leads to this false dependency problem as it simply considers an output event is dependent on all the preceding input events in the process.

Execution Unit based Analysis. False dependencies in syscall based analysis are a major obstacle for attack investigation as it often causes the dependency explosion problem [32], which is a problem of having an excessive number of dependencies, with most of them being bogus. It makes investigation challenging, often leading to wrong conclusions. To address the problem, BEEP [32] and ProTracer [37] propose to divide a long-running process to autonomous execution units. In this way, an output event is only dependent on the preceding input events within the same execution unit. BEEP and ProTracer also detect inter-unit dependencies introduced via memory objects. ProTracer is a variant of BEEP that can significantly reduce runtime and space overhead while the effectiveness of attack analysis remains the same because they share the same mechanism to partition a long process.

Unfortunately, BEEP and ProTracer require complex binary program analysis in order to instrument a target application for execution partitioning at runtime. To detect the inter-unit dependencies, they need to identify memory dependencies across units by analyzing training runs, and instrument the target program to monitor the relevant memory accesses in production runs. Note that identifying all relevant memory accesses that induce dependencies across execution units in complex binary programs via training is challenging. Missing memory accesses in training leads to false-negatives in attack investigation. They also generate a large number of additional syscalls to denote unit boundaries and memory accesses, increasing the storage pressure.

In addition, while BEEP can prune out some false dependencies as shown in Fig. 2-(b) (e.g., between *data1* and `Vim`) by leveraging fine-grained execution units, there are still false dependencies such as those involving *data2* and */tmp/.X11-unit*. This is because, in this example, BEEP considers each file read/write event as a separate unit and detects dependencies between units through memory objects. For example, BEEP considers units that read *data2* (2) and *data3* (3) are causally related to a unit that writes *out.gpg* (11) as texts from *data2* and *data3* are copied into a buffer for copy-and-paste in `Vim`. However, the cross-unit dependency between the unit with *data2* (2) and another unit with *out.gpg* (11) is bogus because the contents copied from *data2* are not pasted to *out.gpg*. The bogus dependency is introduced because BEEP simply detects `memory read` and `memory write` events with a same memory address without checking if there is true information flow between the two. In short, while BEEP can narrow down the scope of investigation, there are still unnecessary files and events in the graph.

Taint Analysis. Taint analysis techniques [22], [21], [20] track information flow between a set of system components (e.g., file, memory, and network), called taint sources, to another set of system components, called taint sinks. Given a set of input related system components to track, taint analysis keeps track of how data from the specified input components are consumed and propagated by individual instructions that operate on the data, in order to identify how they impact other system components. However, most taint tracking approaches including the state-of-the-art tools such as TaintGrind [22] and libdft [21] are expensive as they monitor each instruction to track information flow. Furthermore, they are often not able to track implicit flows caused by control dependencies, introducing false-negatives.

To illustrate the merits and limitation of taint analysis techniques, we use a state-of-the-art open source tool, TaintGrind, to analyze the aforementioned incident. Fig. 2-(c) shows the result from TaintGrind. In this example, TaintGrind fails to identify the dependency between the *data3* and */tmp/tmpfile*. Note that the most important part of the attack (i.e., the leaked confidential data) is not revealed in the attack investigation due to the missing dependency.

```
1  int tripledes_ecb_crypt(..., const byte* from, ...) {
2    ...
3    work = from ^ *subkey++;
4    to ^= sbox8[  work      & 0x3f ];
5    to ^= sbox6[ (work>>8)  & 0x3f ];
6    to ^= sbox4[ (work>>16) & 0x3f ];
7    to ^= sbox2[ (work>>24) & 0x3f ];
8    ...
9  }
```

Fig. 3. Information flow through a table look-up in gpg.

We investigate the case in depth, and find that `gpg` decrypts values through a table lookup operation. Unfortunately, TaintGrind is not able to handle information flow through the table lookup, resulting in missing dependencies. Fig. 3 shows a code snippet extracted from `gpg`. Specifically, the function argument `from` contains an piece of encrypted text. At line 3, the encrypted text is used to calculate the value of `work`, and TaintGrind successfully propagates taint information to the variable. However, at lines 4-7, `work` is used to look-up a table `sbox2-8`, and TaintGrind loses track of taint information at this point because it does not handle information flow via array indexing. Note that most taint analysis techniques do not track information flow through array indexing to avoid the over-tainting problem. Specifically, the over-tainting problem often leads to an excessive number of taint tags, resulting in false-positives. Hence, most taint analysis tools decide not to track such information flow. In addition to table look-up, explicit data flows through computations (e.g., bitwise and arithmetic) and implicit data flows caused by control dependency are often disregarded to avoid the over-tainting problem. Moreover, the significant overhead of taint analysis prohibits its application in practical forensic analysis that requires always-on monitoring to capture attacks in-the-wild.

Causality Inference. Recently, Kwon et al. propose a light-weight causality inference technique LDX [31] using a dynamic analysis called *dual execution*. For a given original execution, LDX derives a slave execution in which it mutates values of input source(s). It then compares the corresponding outputs from the original execution and the slave execution to determine whether the outputs are causally dependent on the source(s). Specifically, if the two executions have different values for an output, LDX considers that the output is causally dependent on the mutated input source(s). To address execution path divergence caused by input perturbation, LDX leverages its novel on-the-fly execution alignment scheme. Unlike dynamic taint analysis techniques (e.g., TaintGrind [22] and libdft [21]), LDX can detect explicit and implicit information flow and has much lower runtime overhead (about 6%).

Fig. 2-(d) shows the graph generated by LDX. Note that it contains only the objects and events related to the attack, without any false dependences. While LDX produces concise and accurate graphs, it requires the dual-execution framework available on the end-user system which doubles the consumption of computational resources (e.g., CPU and memory).

C. Goals and Our Approach

Table I presents merits and limitations of existing causality analysis approaches. In summary, syscall analysis techniques suffer from high false-positive rates due to dependence explosion. While BEEP and ProTracer mitigate the dependence explosion problem, they require complex static, dynamic binary analysis and instrumentation and incur non-trivial space overhead. MPI is efficient and effective, but requires access to source code and domain knowledge for annotation. Taint analysis techniques generally incur significant runtime and space overhead and suffer from the over-/under-tainting problems. LDX requires the dual-execution framework in production run that doubles computational resource consumption.

Our Goal. The goal of this paper is to provide a causality analysis technique with the same accuracy as LDX, but *does not require any changes of end-user systems*, such as instrumenting user applications, modifying the kernel or installing special runtime. Specifically, the end-user only needs to turn on the default audit logging tool that comes with their system, such as Linux Audit, Event Tracing for Windows, and DTrace to collect syscall logs. Upon a security incident, MCI can generate precise causal graphs from the raw log to explain attack causality and assess system damages. We believe such a design would substantially improve applicability.

Our Approach. As shown in Fig. 1, the key idea of MCI is to use causal models to parse raw logs to derive precise causality information. Specifically, in the offline phase, we use LDX [31] as the causality inference engine to construct models for the applications that will be deployed on an end-user system. A causal model is essentially a sequence of inter-dependent syscalls and their causal relations. Such causalities/dependencies can be induced by system objects, called *explicit dependencies*, as they can be determined by analyzing syscalls alone, or induced by memory operations and control dependences, called *implicit dependences*, which are not visible by analyzing syscall events. Note that LDX can detect both explicit and implicit dependencies.

During deployment, given a syscall log collected from the incident, MCI can precisely infer causality between events in the log by parsing the log using the pre-generated models.

D. MCI *on Motivating Example*

We demonstrate the effectiveness of MCI on investigating the incident. Assume the causal models of applications have

TABLE I. COMPARISON OF CAUSALITY ANALYSIS APPROACHES.

	Syscall Analysis [25], [26], [16]	Fine-grained Analysis			Taint Analysis [22], [21], [20]	Causality Inference: LDX [31]	**MCI**
		BEEP [32]/ProTracer [37]	MPI [36]	WinLog [35]			
Space overhead	Low	Mid	Low	Low	High	Low	**Low**
Runtime overhead	Low	Low	Low	Low	High	Low	**Low**
Resource overhead	Low	Low	Low	Low	High	Mid	**Low**
False-positive	High	Mid	Low	Mid	Low	Low	**Low**
False-negative	Low	Low	Low	Low	Low-Mid	Low	**Low**
Granularity	Coarse	Mid	Fine	Mid	Fine	Fine	**Fine**
End-user requirements	None	Training/instrumentation	Code annotation	None	Tainting framework	Dual-execution framework	**None**

been derived offline. Note that generating models does not require any particular expert knowledge on target programs, but rather the typical user level workloads. Model generation is a one-time effort such that models generated for a program can be used for all installations of the program.

Fig. 4-(a), (b), (c), (d), and (e) show the graphical representations of some models from `Vim`. A node is denoted by a letter which represents a syscall, with a superscript (*) representing a sequence of syscalls. A subscript represents the (symbolic) system object (e.g., file or socket) operated by the syscall. For example, model (a) is for the behavior of opening and decrypting a gpg file. Specifically, as shown in the legend in Fig. 4, the first node of (a) r_α indicates a `read` syscall on α which is *stdin*. Note that each model has its own legend for the subscript. The first node is a syscall that causes the entire behavior. Intuitively, the model represents reading from a command line that loads a gpg file. The second node, s_β, represents a `stat` syscall on a file β (output file). The `GnuPG` plug-in uses a temporary file to store decrypted contents and then informs `Vim` to open. Subscript β symbolizes the temporary file which contains decrypted contents. The second node essentially checks whether the file exists. After that it loads a key file to prepare decryption which is represented as a third node (r^*_γ). Then, it checks (`stat`) the output file again (s^*_β). Finally, the fifth node (r^*_δ) represents reading a gpg file which is an encrypted file. The sixth node (w_β) indicates that the decrypted contents are written onto the output file (β). Then, the `GnuPG` plug-in sends a notification to `Vim` via a pipe which is shown in the last node (w_ϵ). Note that symbols in subscript (e.g. α, β) can be instantiated to any concrete file handler during parsing. The same subscript β in s_β and the later nodes s^*_β and w_β dictate that these syscalls must operate on the same file. The third and fifth nodes are denoted by a superscript *, representing a sequence of `read` system calls (`read`*) on different files γ and δ.

The directed edges between nodes represent the causality/dependency between syscalls, with the solid and dotted edges representing the explicit and implicit dependencies, respectively. For example, in (a), there are explicit dependences from s_β to w_β and implicit dependencies from r^*_γ and r^*_δ. The implicit dependencies are caused by memory operations that copy values from a crypto key file (γ) to encrypted contents δ that are detected and modeled by MCI.

Fig. 4-(f) illustrates a syscall log collected during the incident by the default Linux Audit tool [17]. Given the syscall log and the models, MCI automatically parses the log and hence derives the corresponding dependencies. Each box in (f) denotes a model instance with the letter annotated on the box representing the model id. Note that we use different background colors for boxes to represent nodes belong to different models. We omit the dependences in the model instances for readability. For readability, we use superscripts to denote event timestamps.

The model instances essentially tell us that the user first opened a gpg file (i.e., *out.gpg*) by model (a), opened and copied a file (i.e., *data2*) without pasting by model (b), and opened, copied, and pasted another file (i.e., *data3*) by model (c). Observe that there are events that belong to multiple models, which allow us to determine causality across models and hence *compose* the whole attack path. For instance, event s^{11}_5 belongs to both models (c) and (d) (i.e., the node in the two boxes in blue and green), suggesting that the contents from *data3* are copied to the previous gpg file. The subscript 5 corresponds to file `viminfo` that is used to indicates the state of editing. Note that model (c) does not have explicit dependencies with other models. Hence, without model (d), causality between model (c) and other models is difficult to reveal. After a few editing operations by model (d), the user finally saved the contents to a new gpg file by model (e). The event s^{11}_5 belonging to models (c) and (d) indicates that the new gpg file contains information from *data3* (confidential data). Note that the matched instance of model (b) does not have any overlapping nodes with other model instances nor explicit dependencies, and hence no causal relations with others. This indicates that *data2* is not involved in the incident. The final causal graph is shown Fig. 2-(d), which is accurate and concise, without any missing or bogus dependencies.

III. PROBLEM DEFINITION

In this section, we introduce a number of formal definitions and the problem statement for MCI.

A. Definitions

Causal Model. Fig. 5 shows definitions for a causal model. Specifically, *SysName* represents syscall names such as *open* and *read*. *Repetition* indicates how many times a term or node repeats. It could be a constant number, a variable such as n or m, or $*$ representing any number of repetition. Variables are needed to to denote repetition constraints across syscall events. *ResourceSymbol* represents a symbol for a resource handler that a system call operates on (e.g., file handler). A *Term* is a sequence of *Node*s that could be annotated with the number of repetitions. A node N is a syscall annotated with a set of parameters denoted by *SymbolicResource*. A symbolic resource can be instantiated to different concrete resources during parsing. Two nodes with the same symbolic resource indicates that they have explicit dependency. An *Edge* denotes dependency/causality between two nodes N_{from} and N_{to}. Finally, a causal model is defined as a 3-tuple $< \overline{T},\ \mathbb{P}(E)_{implicit},\ \mathbb{P}(E)_{explicit} >$ where $\overline{T}$ is a sequence of terms, $\mathbb{P}(E)_{implicit}$ is the set of implicit dependency edges and $\mathbb{P}(E)_{explicit}$ is the set of explicit dependency edges. The definitions of two kinds of edges can be found in Sec. II.

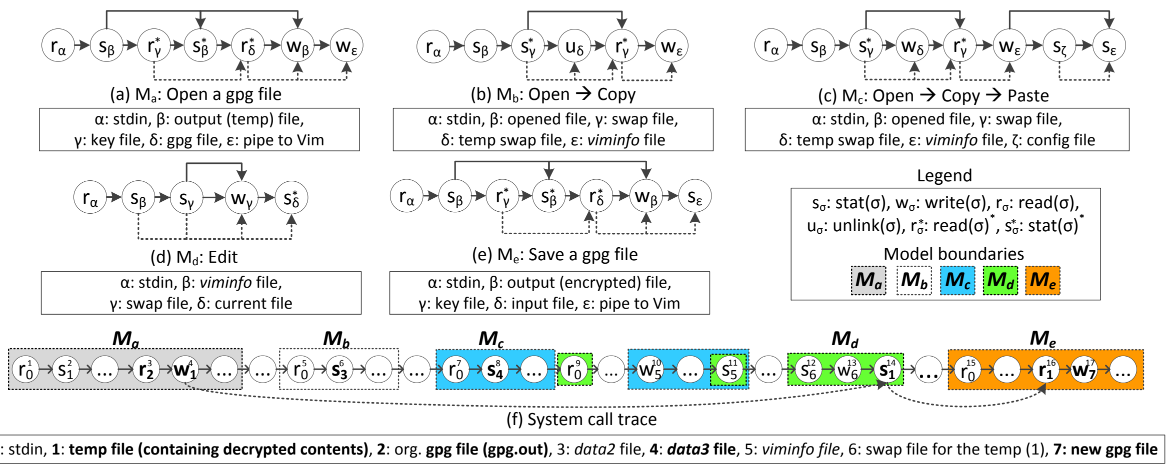


Fig. 4. MCI on the Motivating Example.

SyscallName	$SysName ::=$	open | read | write | ...
Repetition	$R ::=$	$1 \mid 2 \mid 3 \mid ... \mid n \mid m \mid *$
SymbolicResource	$S ::=$	$\{\alpha,\ \beta,\ \gamma, ...\}$
Term	$T ::=$	$N \mid NT \mid (T)^R$
Node	$N ::=$	$SysName_{\mathbb{P}(S)}$
Edge	$E ::=$	$< N_{from}, N_{to} >$
Model	$M ::=$	$< \overline{T},\ \mathbb{P}(E)_{implicit},\ \mathbb{P}(E)_{explicit} >$

Fig. 5. Definition of Causal Model.

For example, the model in Fig. 4 (a) can be represented as follows. First, $\overline{T}$ can be represented by a sequence: $read_\alpha, stat_\beta, read^*_\gamma, stat^*_\beta, read^*_\delta, write_\beta, write_\epsilon$. Implicit dependencies (dotted edges below nodes) are denoted as follows: $\{\langle read^*_\gamma, read^*_\delta\rangle, \langle read^*_\delta, write_\beta\rangle, \langle read^*_\delta, write_\epsilon\rangle\}$. Explicit dependencies (solid edges above nodes) are the following: $\{\langle stat_\beta, stat^*_\beta\rangle, \langle stat_\beta, write_\beta\rangle\}$. Observe the nodes in an explicit edge have the same resource symbol, indicating that they operate on the same resource. In the paper, we will use the more concise graphical representations when possible.

Syscall Trace. As shown in Fig. 6, a system call trace T is a sequence of trace entries $\overline{TE}$ where a trace entry is a system call name annotated with a set of $ConcreteResource$ that represents concrete resource handlers, and a number $\mathbb{N}$ that represent an index of TE in T. Note that it does not contain any dependency information. The first 6 entries in Fig. 4 (f) are represented as $\overline{TE} = (read^1_0, stat^2_1, ..., read^3_2, write^4_1, ...)$. Note that the subscripts represent concrete resource handlers and the superscripts represents indexes.

ConcreteResource	$C ::=$	$\mathbb{N}$
TraceEntry	$TE ::=$	$SysName^{\mathbb{N}}_{\mathbb{P}(C)}$
SyscallTrace	$T ::=$	$\overline{TE}$

Fig. 6. Definition of Syscall Trace.

B. Problem Statement

We aim to infer fine-grained causality from a syscall trace by parsing it with models. This procedure can be formally defined as a function of T and $\mathbb{P}(M)$:

$$T \times \mathbb{P}(M) \rightarrow (TE \rightarrow \mathbb{P}(N \times M))$$

Specifically, given a syscall trace T and a set of models $\mathbb{P}(M)$, the function generates a mapping, in which a trace entry is mapped to a set of nodes N in model M. It is a set because a trace entry can be present in multiple models as shown in the motivation example in Sec. II. With the mapping, the dependencies between trace entries can be derived from the dependencies between the matched nodes in the models. For example, parsing the trace in Fig. 4 (f) using the models in (a)-(d) generates the following mapping. The first 4 events are mapped to model (a): $(read^1_0 \rightarrow < read_\alpha, M_a >), (stat^2_1 \rightarrow < stat_\beta, M_a >), (read^3_2 \rightarrow < read^*_\delta, M_a >), (write^4_1 \rightarrow < write_\beta, M_a >)$. Moreover, $stat^{11}_5$ belongs to two models, resulting in two mappings: $(stat^{11}_5 \rightarrow < stat_\epsilon, M_c >), (stat^{11}_5 \rightarrow < stat_\beta, M_d >)$. It entails the following concrete dependency edges $< read^3_2, write^4_1 >$ (from model edge $< read^*_\delta, write_\beta >$ in (a)) and $< stat^{12}_5, stat^{14}_1 >$ (from model edge $< stat_\beta, stat^*_\delta >$ in (d)). The first edge indicates implicit dependency between the original gpg file (out.gpg) and a temp file containing its decrypted contents, and the second edge implies that the copy and paste action is related to the temp file containing the decrypted contents of the original gpg file (*out.gpg*). Such dependency edges lead to a causal graph as that in Fig. 2-(d).

The mapping may not be total, depending on the comprehensiveness of the models. An important feature of MCI is *model composibility*, meaning that a complex behavior can be composed by multiple models sharing some common nodes. For instance, a complex user behavior in `Vim` such as "open file, edit, copy, edit, paste, save, reopen" can be decomposed to multiple primitive models. As such, the number of models needed for regular workload is limited as shown in Sec. V.

The key challenge of MCI *lies in parsing the trace that does not contain any dependencies with models that contain dependency information*, which entails solving two prominent technical problems discussed next.

C. Technical Challenges: Complexity and Ambiguity

1) Language Complexity: According to our definition, a trace is a string in the trace language that does not contain dependency information, our problem is essentially to parse the string to various model instances. In the following, we use the classic language theory to understand the complexity of our problem. Note that although it seems that we could consider models as graphs and leverage the sub-graph isomorphism theory to understand our problem, there are places that can hardly be formulated in the graph theory. For instance, our trace is not a graph because it does not have implicit dependency information. Furthermore, our model may have constraints among the numbers of event repetitions (e.g., the number of `close` matches with the number of `open` while the number of repetitions may vary). Such constraints can hardly be represented in graphs.

The classical Chomsky hierarchy [9], [10] defines four classes of languages characterized by the expressive power of their defining grammars: *regular*, *context-free*, *context-sensitive*, and *recursively enumerable*. More expressive grammar can describe more complex language but requires higher cost in parsing. We study some of representative causal model types observed in real-world programs. For each type, we show a sample grammar and discuss the complexity of the grammar as well as scalability of the corresponding parser.

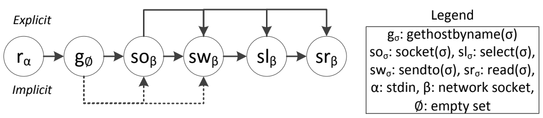


Fig. 7. Regular Model from ping [40].

Regular Model. Fig. 7 shows a model from `ping` [40], representing a behavior "*resolving a network address, sending a packet, and receiving a response.*"

Observe that the explicit dependencies (solid edges) are caused by the socket (β). The implicit dependencies (dotted edges) are introduced because `gethostbyname()` decides whether to execute `socket()` and `sendto()` meaning that they have control dependences. In particular, if `gethostbyname()` returns an error, the program immediately terminates. Also, `sendto()` is dependent on the return value of `gethostname()` (e.g., IP address) as the `ping` program composes and sends Internet Control Message Protocol (ICMP) packets that contain the returned IP address. Such dependencies are not visible at the syscall level. Note that in any model, the first node, which is always an input syscall, has dependencies leading to all other nodes. Recall that a model is acquired from LDX that mutates an input syscall and observes changes at output syscalls (e.g., the first node in Fig. 7 is a syscall that reads an option from the command line that leads to all the other syscalls in the model).

The model in Fig. 7 can be simplified by a regular grammar (e.g., regular expression) which is the simplest one in Chomsky hierarchy. A regular language parser has very good scalability. From our experience, most models (53 out of 56 models in our evaluation) fall into this type.

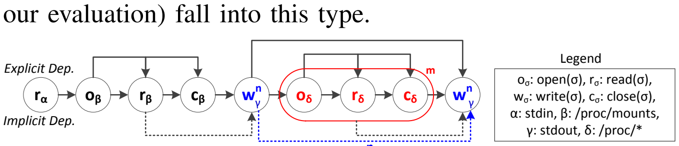


Fig. 8. Context-Free Model from procps [8].

Context-free Model. There are cases that the models need to be context-free. Fig. 8 shows such a model extracted from `procps` [8]. The model represents "*retrieving file system information.*" It first reads a file that contains information about the list of file systems. It then uses an outer loop to emit the information for individual file systems. For each file system, an inner loop is used to collect information about the file system from multiple places (e.g., different disks).

As shown in Fig. 8, three symbols from the 2^{nd} to the 4^{th} (o_β, r_β, c_β) have explicit dependencies due to the file containing the list of file systems (β). The 5^{th} symbol w_γ^n is to emit the header information for each file system, causing the implicit dependency between the 3^{rd} symbol r_β and the 5^{th}. The superscript n denotes that there are n file systems. The 6^{th}, 7^{th}, and 8^{th} symbols (o_δ, r_δ and c_δ) form a term, corresponding to the inner loop that reads m places to collect information for the n file systems. Note that m may not equal to n as multiple files may be accessed in order to collect information for a file system. After that, the 9^{th} symbol w_γ^n emits the collected information for the n file systems. Note that the number of writes in the 5^{th} and the 9^{th} symbols need to be identical (n times). The constraints on the numbers render the model cannot be transformed to an automaton that handles a regular language. It is essentially context-free. The parser for a context-free language requires some push-down mechanism, incurring higher complexity. We have encountered 2 context-free models in our evaluation.

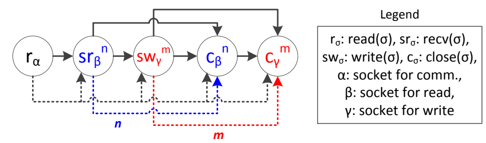


Fig. 9. Context-Sensitive Model from Raft [43].

Context-sensitive Model. In some rare cases, even context-free models are not sufficiently expressive. Fig. 9 shows a model from [49] which is a distributed voting application that implements the Raft consensus protocol [43]. The program can exchange network messages between different number of users to get a consensus. The model describes a voting procedure. Specifically, it receives network messages from n users (n iterations of `read()`), and sends network messages to m users (m iterations of `write()`). Later, it closes the sockets for n users and then m users. The crossing-constraints between m and n ($r_2{}^n$, c^n) and (w^m, c^m) require a context-sensitive language. However, a parser for a context-sensitive language is prohibitively expensive in general (PSPACE complexity [15]). We have not encountered any models more complex than context-sensitive languages. The various language complexities pose a prominent challenge: since syscall events belonging to multiple models interleave and are often distant from each other, we cannot know which model an event belongs to until reaching the end of the model. As such, we do not know which complexity class shall be used to parse individual events. As we will show later, we develop a uniform parsing algorithm for multiple complexity classes that leverages the special characteristics of causal models.

2) Ambiguity: The strings (of syscalls) parsed by multiple models may share common parts (e.g., common prefixes). In the worst case, multiple models may accept the same string, although we have not encountered such cases for models within the same application. As a result during trace parsing, given a syscall, there may be multiple models that it can be attributed to and MCI does not know which model(s) are the right ones. We call it the *ambiguity problem*.

For instance, consider a trace, the ground-truth causality of the trace, and a model shown in Fig. 10-(a), (b), and (c), respectively. Observe that the model has a socket read followed by a file write. The two have implicit dependency but not explicit dependency visible at the syscall level. The three boxes in Fig. 10-(b) denote the three real model instances.

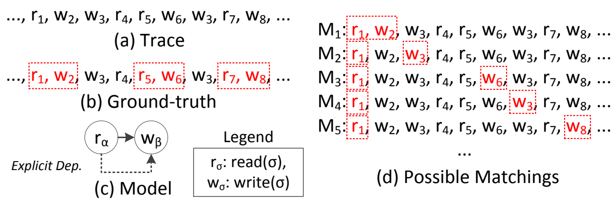


Fig. 10. Ambiguity Problem.

When the model is used to parse the trace, due to the lack of dependencies between the two syscalls in the model, there are many possible matchings as shown in Fig. 10-(d). Note that except M_1, the other matchings are incorrect even though they all appear possible at the syscall level. In practice, such incorrect matchings introduce false causalities which hinder attack investigation. Moreover, ambiguity may cause excessive performance overhead because MCI has to maintain numerous model instances at runtime. The root cause of the problem is that the trace does not have sufficient information. Hence, we develop a method that leverages explicit dependences to mitigate the problem. Details can be found in Sec. IV-B.

IV. SYSTEM DESIGN

MCI consists of two phases: model construction and model parsing. The former is offline and the latter is meant to be deployed for production run.

A. Model Construction

Given an application, the forensic analyst provides a set of regular workloads. The application is executed on the LDX system with the workloads. The dependences detected by LDX, including explicit and implicit dependences, are annotated on the syscall events in the audit logs. The annotated logs are analyzed to extract inter-dependent subsequences, which are further symbolized (i.e., replacing concrete resource handlers with symbolic ones). The sequences of symbolic syscalls with dependences constitute our causal models.

In the following, we use a program snippet in Fig. 11 to illustrate how MCI constructs causal models. It first reads a network message (line 1) and encrypts the received message (line 2). Later, it stores the encrypted message to a local file (line 3) and sends a notification to a GUI component (line 5).

```
1  while( (len = read(socket, buf, 1024)) != -1 ) {
2      ebuf = encrypt(buf);
3      write( file, ebuf, 4096 );
4  }
5  sendmsg( wnd, "Update: " + ebuf ... );
```

Fig. 11. Example Program.

1) Dependencies Identification by LDX*:* The program is executed with a typical workload on LDX [31] to collect a system call log T. To identify dependencies, LDX mutates the value of input syscall `read()` in the slave execution. By contrasting the values of the following syscalls (e.g., the `write()` and `sendmsg()`) in the two executions, LDX identifies all the dependencies between syscalls.

```
1  // SR: means that the system call is a source system call.
2  // CD: means that the system call is causally dependent on the input
3  [SR] 100 = read( 0x11/*file handle*/, "AAAA..."/*Mutation: BBBB...*/, 1024 );
4  [CD] 4096 = write( 0x12/*file handle*/, "Contents to be written", 4096 );
5  [CD] -1 = read( 0x11/*file handle*/, "Returned buffer", 1024 );
6  [CD] 20 = sendmsg( ...., "Content to be written", 4096 );
```

Fig. 12. Causally dependent system calls from LDX.

Fig. 12 shows the output generated by LDX. It includes two `read()`s (lines 3 and 5), one `write()` (line 4) and one `sendmsg()` (line 6) which are causally dependent on the source (i.e., `read()` at line 2). More specifically, the `write()` at line 4 and `sendmsg()` at line 6 are (implicitly) dependent on the source by variables `buf` and `ebuf`, and the `read()`s at lines 2 and 4 are explicitly dependent on the source due to the socket handler `0x11`.

The generated sequence of syscalls includes all the syscalls causally dependent on the source (line 3). We hence leverage them as a sample of the model. Note that LDX also returns dependences between syscalls inside the sequence such as the dependence between lines 3 and 4.

2) Symbolization: The collected sequence of syscalls cannot be directly used as a model due to the concrete arguments. For instance, in Fig. 12, syscalls have concrete values (e.g., handlers `0x11` and `0x12`) which may differ across executions. Hence, we symbolize concretes values in syscalls by replacing with symbols (e.g., α and β). For instance, if two syscalls share the same argument, they are assigned the same symbol.

If the application supports repeated workload, there must be repetitions in the syscalls that need to be modeled (such as n and m in Fig. 5). To do so, MCI duplicates the workload a few times and feeds the new workloads to LDX again. Subsequences that have a constant number of repetitions across workloads are annotated with the constant. Those that have varying numbers of repetitions across workloads are annotated with '*'. If there are correlations between the repetition numbers of multiple subsequences (inside the same model), variables n/m are used to model the number of repetition, such as the previous example Fig. 8 in Sec. III-C1.

```
1  SUCCESS = read( fd1 /* file handle*/, *, * );
2  SUCCESS = write( fd2 /* file handle*/, *, * );
3  FAILURE = read( fd1 /* file handle*/, *, * );
4  SUCCESS = sendmsg( *, *, * );
```

Fig. 13. Symbolized syscalls.

Fig. 13 shows a symbolized log. For example, 0x11 in `read()` in Fig. 12 is replaced by a new symbol fd_1 and 0x12 in `write()` in Fig. 12 is generalized to another symbol fd_2. 0x11 in the second `read()` is replaced by the previously assigned symbol fd_1 as it already appeared before. Moreover, as shown in Fig. 13, all concrete return values are symbolized as either SUCCESS or FAILURE. They are part of the models in our system although our formal definitions did not describe them for brevity. The constructed model is shown in Fig. 14. The formal model construction algorithm is elided due to the space limitations.

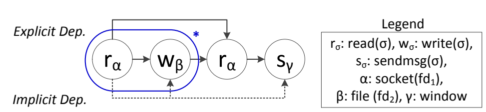


Fig. 14. Constructed model from the example.

B. Trace Parsing with Models

In this section, we describe how MCI parses an audit log with models. As we described in Sec. III-C1, if we simply consider an audit log as a string of the trace language, we need to consider three language classes in the Chomsky hierarchy, namely, regular, context-free, and context-sensitive languages. Recursively enumerable languages are never encountered in our experience. A more expressive language requires more expensive parser. For instance, context-free language can describe almost all causal models we have encountered but context free parsers have a time complexity of n^3 where n is the length of a string (the number of events in audit log in our case), thus they are too expensive to handle real-world logs that can grow in the pace of gigabytes per day [33] (corresponding to millions of events). Context-sensitive parsers have even higher computational complexity. Furthermore, our parser needs to be able to substantially mitigate the ambiguity problem in which MCI does not know which models an event should be attributed to.

Segmented Parsing. Our proposal is *not to consider a trace as a simple string, but rather a sequence of symbols with explicit inter-dependences.* Note that explicit dependences can be directly derived from the trace. The basic idea is hence to leverage explicit dependences to partition the sequence of terms/nodes in a model into *segments*, delimited by terms/nodes that are involved in some explicit dependences. Therefore, all the terms/nodes inside each segment are a string in some regular language. The essence is to leverage explicit dependences to reduce language complexity. During parsing, we first recognize (from the trace) the explicit dependences that match those of the model. These dependences partition the trace into sub-traces. Then automata are used to recognize model segment instances from the sub-traces. Since string parsing is only carried out within small sub-traces instead of the lengthy whole trace, ambiguity can be substantially suppressed. We call the technique *segmented parsing*.

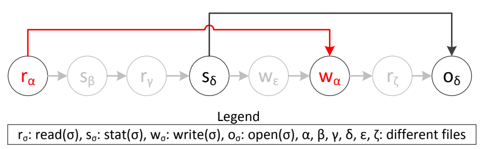


Fig. 15. Example for Segmented Parsing

Next, we use an example to illustrate the basic idea and then explain the algorithm. Fig. 15 shows a sample model. Observe that there are explicit dependences between the 1^{st} and the 6^{th} nodes (r_α and w_α), and between the 4^{th} and the 8^{th} nodes (s_δ and o_δ). The sequence of terms/nodes involved in explicit dependences form the *model skeleton*. In our example, it is r_α-s_δ-w_α-o_δ. The skeleton partitions the model into *sub-models*. A sub-model is a sub-sequence of nodes/terms of the model that are delimited by explicit dependences but themselves do not have any explicit dependences. In Fig. 15, three sub-models are obtained as follows: s_β-r_γ delimited by r_α and s_δ, w_ϵ delimited by s_δ and w_α, and r_ζ delimited by w_α and o_δ.

During parsing, we first find instances of the model skeleton. For each skeleton instance, we try to identify instances of sub-models within the trace ranges determined by the skeleton instance. Any mismatch in any sub-model indicates this is not a correct model instance and the corresponding data structures are discarded. In our example, we first locate the possible positions of r_α, s_δ, w_α, o_δ in the trace, and then look for the instances of s_β-r_γ in between the positions of r_α and s_δ, and so on. Partitioning a model to a skeleton and a set of sub-models is straightforward. Details are hence elided. Given a trace, to facilitate segmented parsing, we extract a number of *trace indexes*, each containing all the nodes related to the same system object (e.g., a file) and the position of the nodes in the raw trace. These indexes allow our parser to quickly locate skeleton instances in the trace. Fig. 16 shows an example of index extraction from a trace. Observe that all the nodes in an index have explicit dependences.

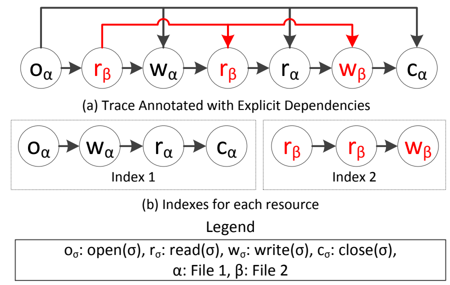


Fig. 16. Trace Preprocessing

Algorithm 1 Locating Skeletons

Input: trace T, indexes I, model skeleton S
Output: a set of skeleton instances P, each consisting of a mapping that maps a symbolic resource to a concrete one, and a sequence of positions

1 **procedure** LOCATESKELETON(T, I, S)
2 **for all** node $N_\alpha \in S$ **do**
3 **if** $P \equiv \{\}$ **then**
4 $P \leftarrow \{\langle\{\alpha \rightarrow h\}, i\rangle \mid for\ all\ T[i] = N_h\}$
5 **else**
6 **for all** $\langle map, seq\rangle \in P$ **do**
7 **Let** the last position in seq **be** i
8 **if** $map[\alpha] \neq nil$ **then**
9 $pos \leftarrow$ findbeyond(N,i,$I[map[\alpha]]$)
10 **if** $pos \neq -1$ **then**
11 $seq \leftarrow seq \cdot pos$
12 **else**
13 P.remove($\langle map, seq\rangle$)
14 **end if**
15 **else** // scan all indexes to find N_h syscalls that are beyond i
16 ... // and instantiate α to h.
17 **end if**
18 **end for**
19 **end if**
20 **end for**
21 **return** P
22 **end procedure**

Algorithms. The parsing procedure consists of three major steps. The first one is to preprocess trace to extract indexes, which has been intuitively explained before. The second step is to locate skeleton instances in the trace and the third is to parse sub-models. In the following, we explain the algorithmic details of steps two and three.

The algorithm of locating skeleton instances is shown in Alg. 1. It takes the trace T, the indexes I that can be accessed by the concrete resource id (e.g., file handler), and a model skeleton S, and identifies all the possible instances of the skeleton. The result is stored in P. Each instance is a pair $\langle map, seq\rangle$ with map projecting each symbolic resource (e.g., α and β) in the skeleton to some concrete handler and seq storing the trace positions of the individual nodes in the skeleton. To simplify our discussion, we assume the skeleton does not have repetitive nodes or terms. The algorithm can be easily extended to handle such cases.

The main procedure iterates over each node N_α in the skeleton (line 2) with N the syscall and α the symbolic resource. For the first node (indicated by an empty result set P), the algorithm considers each syscall of the same type N, in the form of N_h at location i in the trace, may start an instance of the skeleton, and hence instantiates α to the concrete handler h and records its position i (lines 3 and 4). If N_α is not the first node, the algorithm iterates over all the skeleton candidates in P in the inner loop (lines 6-18) to check if it can find a matching of the node for these candidates. If not, the skeleton candidate is invalid and hence discarded. Specifically, for each skeleton candidate denoted as $\langle map, seq\rangle$, line 7 identifies the trace position of latest node i. This is needed as the algorithm looks for the match of N_α in trace entries beyond position i. The condition at line 8 separates the processing to two cases with the true branch denoting the case that α has been instantiated before, that is, a node of the same symbolic resource was matched before (e.g., w_α in Fig. 15), the else branch otherwise (e.g., s_δ in Fig. 15). In the first case (lines 9-11), the algorithm looks up the index of the concrete handler associated with α, i.e., $I[map[\alpha]]$, to find a concrete syscall N beyond position i (line 9). If such a syscall is found, we consider the algorithm has found a match and the new position pos is appended to seq (line 11). Otherwise, the skeleton candidate is not valid and removed (line 13). Here, we have another simplification for ease of explanation. Line 9 may return multiple positions in practice while in the algorithm we assume it only returns one. The extension is straightforward.

In the else branch, the node has a new symbolic resource, the algorithm has to go through all indexes to find all instances of N and instantiate the symbolic resource accordingly. This may lead to the expansion of the candidate set P. Details are elided. To reduce search space, we use time window and other syscall arguments to limit scopes.

Algorithm 2 Model Parsing

Input: trace T, skeleton instances P, sub-models S
Output: the concrete syscall entries that correspond to the sub-models in the temporal order

```
1  procedure ParseSubModels(T, P, S)
2    for all ⟨map, seq⟩ ∈ P do
3      for i from 0 to |S| − 1 do
4        instance[i] ← parse(T[seq[i], seq[i + 1]], S[i])
5      end for
6      if all instance[0 − (|S| − 1)] are not nil then
7        if none of the concrete syscalls in instance[0(−|S| − 1)] share the same resource id then
8          output instance[0 − (|S| − 1)]
9        end if
10     end if
11   end for
12 end procedure
```

Given a set of skeleton instances for a model M, Alg. 2 parses the sub-models of M. In particular, the outer loop (lines 2-11) iterates over all the skeleton candidates identified in the previous step. If matches can be found for all sub-models regarding a skeleton instance, the matches are emitted. Otherwise, it is not a legitimate instance and discarded. Specifically, the inner loop in lines 3 and 4 iterates over individual sub-models in order. In the i^{th} iteration, it uses automata to parse sub-model $S[i]$ in the trace range identified by the i^{th} segment identified by the skeleton candidate, which is from $seq[i]$ to $seq[i+1]$ (line 4). Automata based parsing is standard and elided. After such parsing, line 6 checks if we have found matches for all sub-models. If so, line 7 further checks that none of the concrete syscall entries that are matched with some node in a sub-model do not share the same resource (and hence have explicit dependences). This is because the model specifies that there are not explicit dependences between the corresponding nodes. Line 8 outputs the parsing results.

Handling Threaded Programs. Threading does not pose additional challenges to MCI in most cases because syscalls from different threads have different process ids so that models can be constructed independently for separate threads. Explicit dependences across threads can be easily captured by analyzing audit logs. Some programs such as `Apache` and `Firefox` use in-memory data structures (e.g., work queues) to communicate across threads, causing implicit dependences. However, it is highly complex to model and parse behaviors across threads due to non-deterministic thread interleavings. We observe that these data structures are usually protected by synchronizations, which are visible at the syscall level, and the synchronizations should follow the nature of the data structures, such as first-in-first-out for queues. Hence, MCI constructs models for individual threads including the dispatching thread and worker threads. The models include the synchronization behaviors. It then leverages the FIFO pattern to match nodes across threads. It works nicely for most of the programs we consider except `transmission`, whose synchronization is not visible at the system level (Sec. V).

V. Evaluation

In this section, we evaluate MCI with a set of real-world programs in order to answer the following research questions.

RQ 1. How many models are required to infer causality for these programs in production runs (Sec. V-A1), and how much efforts are required to construct models? (Sec. V-A2)

RQ 2. How effective is MCI for system wide causality inference including multiple long-running programs and various activities? (Sec. V-B)

RQ 3. How effective is MCI for realistic attack investigation? (Sec. V-C)

RQ 4. Is MCI scalable on large workloads for long-running programs? (Sec. V-C3)

Experiment Setup. We evaluate our approach on 17 real-world programs. Table II shows the programs and models we constructed. Note that 15 out of the 17 programs (except `zip` and `Vim`) are network related which is a popular channel for cyber-attacks. For each program, we construct models offline. We use typical workloads briefly described in the second column of Table II. Specifically, if there are available test inputs for a program, we use them as the typical workloads. Otherwise, we construct inputs by inspecting program manuals and identifying options and commands that can trigger different functionalities, such as for `proftpd`, `CUPS`, and `zip`.

A. Model Construction

Table II shows the constructed models for each program. Columns 1 and 2 show programs and model description. Column *Size* shows the number of nodes in each model. The numbers in/out parentheses are for the same behaviors with/without `HTTPS`. The next two columns show the number of explicit and implicit dependencies in each model. The last column (Lang.) shows the language class of each model (Regular (Reg.), Context-free (C.F.), or Context-Sensitive (C.S.)).

TABLE II. DETAILS ON MODEL CONSTRUCTION

Program	Model Description	Size[1]	D_{exp}[2]	D_{imp}[3]	Lang.[4]
Firefox	Tab Open/Switch/Close	7/9/5	2/2/1	3/4/3	Reg.
	Load a URI	12	2	4	Reg.
	Download (Save)	15	3	5	Reg.
	Click a link	9	2	3	Reg.
Apache	HTTP(S) resp.	17 (21)[5]	3 (4)[5]	8 (11)[5]	Reg.
	CGI resp.	26 (33)[5]	4 (5)[5]	11 (14)[5]	Reg.
Lighttpd	HTTP(S) resp.	8 (11)[5]	2 (3)[5]	4 (6)[5]	Reg.
	CGI resp.	16 (19)[5]	3 (4)[5]	7 (9)[5]	Reg.
nginx	HTTP(S) resp.	14 (17)[5]	3 (4)[5]	6 (9)[5]	Reg.
	CGI resp.	21 (24)[5]	4 (5)[5]	8 (11)[5]	Reg.
CUPS	Add printers	6	1	3	Reg.
	Remove printers	5	1	3	Reg.
	Modify printers	6	1	3	Reg.
	Print a doc.	7	2	4	Reg.
vim	Open	8	1	5	Reg.
	Edit	10	1	4	Reg.
	Save	13	2	4	Reg.
	Save As	15	3	6	Reg.
	Copy and Paste	14	3	6	Reg.
	Copy	11	1	5	Reg.
	Plug-in (gpg)	21	2	6	Reg.
elinks	Browse	11	3	6	Reg.
	Save	6	2	5	Reg.
	Upload	7	2	5	Reg.
alpine	Send emails	10	2	6	Reg.
	Send files	13	3	7	Reg.
	Download emails	9	2	6	Reg.
	Download files	11	2	5	Reg.
	Open a link	8	2	4	Reg.
zip	Compress file(s)	16	8	5	C.F.
	Use encryption	6	4	3	Reg.
transmission	Download	17	4	8	Reg.
	Add a torrent file	6	3	3	Reg.
	Add a magnet	12	3	7	Reg.
proftpd/ lftp/yafc	Login	5/4/6	1/1/2	4/3/4	Reg.
	Create directory	4/4/4	2/2/2	3/3/3	Reg.
	Delete directory	3/4/4	1/2/2	3/3/3	Reg.
	List directory	3/3/3	1/1/1	3/3/3	Reg.
	Upload	7/8/18	2/2/3	5/5/9	Reg.
	Download	6/7/16	2/2/4	5/6/9	Reg.
wget	Download (HTTP(S))	7 (15)[5]	2 (4)[5]	5 (8)[5]	Reg.
ping	Option -f	6	2	5	Reg.
	Option -r	5	2	5	Reg.
procps	Get file system info.	6	3	4	C.F.
raft [49]	Voting	5	2	6	C.S.
	Leader Election	7	2	7	Reg.
Average	-	10.2	2.4	5.4	-

[1]: # of nodes in a model. [2]: # of explicit dependencies (edges) in a model.
[3]: # of implicit dependencies (edges) in a model. [4]: Language Class of a model.
[5]: for HTTPS.

We have the following observations from the results. First, the size of model is relatively small (on average 10.2 nodes) and there are on average 2.4 explicit dependencies (more than 4 nodes) for each model. The strong presence of explicit dependencies allows MCI to perform segmented parsing effectively. Second, we observe three language complexity classes and most models fall into the regular class. It supports our design choice of integrating regular parsers (i.e., automata) with explicit dependency tracking.

1) # of Models Required: The constructed models listed in Table II are sufficient to infer causality for logs from realistic scenarios described in Sec. V-C including the motivation example in Sec. II. The number of models for each program ranges from 3 to 12 which is fairly small and not difficult to obtain in practice. We observe that the primary reason why MCI is effective with a small number of models is model composibility, namely, primitive models can be used to compose complex behaviors. For instance, models for "Edit" and "Save" can compose a new model "Edit and Save".

2) Efforts on Model Construction: To construct models, a program is executed repeatedly on LDX. The number of runs required to construct a model depends on the number of events in the model. Specifically, we first run a program with a workload on LDX to identify all the events causally dependent on the workload. Note that the detected events constitute the bulk of the model. Assume there are n such events (nodes). For each node in the model, MCI mutates the value of the corresponding syscall to determine dependencies on the node inside the model. To figure out the repetition factors of the node (Sec. IV), MCI runs k times for the node, each execution repeats the workload for different times. In total, we run a program $(k*n)+1$ times to construct a model. In our experiments, $k = 10$. On average, the machine time to construct a model, including LDX execution time and model extraction time, takes 4 minutes (253 seconds).

B. System-wide Causality Inference

In this experiment, we apply MCI to infer causality on a system wide syscall trace collected for the system execution of a week, to demonstrate the effectiveness of causality inference for realistic programs with production runs. The trace includes syscall logs from multiple programs including those in Table II. Specifically, we enable Linux Audit and use the programs in Table II with typical workloads for a week. Given the collected trace, we identify all the inputs that appear in the trace (e.g., file reads, command line arguments, user interactions). Then, we build a forward causal graph from each input, i.e., identifying all other syscalls depending on the input, using MCI and compare it with the ground truth by LDX. During the experiment, we record all inputs used for the programs. Then, we re-execute the program with the recorded inputs to reproduce the same execution. To do so, we develop a lightweight record and replay system similar to ODR [5]. LDX is run on top of the replay system to derive the ground truth. Note that due to the limitation of the replay system, the replayed execution may differ from the original execution. Such differences are counted as false-positives/negatives for conservativeness.

TABLE III. RESULTS FOR SYSTEM-WIDE CAUSALITY INFERENCE.

Program	# of events	# of causality	# of matched models	FP	FN
Firefox	2,313 M	11 M	549 K	8.3%	3.2%
Apache	296 M	6.6 M	435 K	0%	0%
Lighttpd	125 M	3.3 M	275 K	0%	0%
nginx	187 M	3.8 M	246 K	0%	0%
proftpd	49 M	2.1 M	179 K	0%	0%
CUPS	25 M	918 K	88 K	0%	0.8%
vim	43 M	4 M	219 K	0%	0%
elinks	38 M	3.6 M	145 K	0%	0%
alpine	116 M	4.7 M	231 K	0%	0.3%
zip	5 M	634 K	36 K	0%	0%
transmission	250 M	6.9 M	479 K	3.8%	5.2%
lftp	11 M	438 K	54 K	0%	0%
yafc	9 M	616 K	43 K	0%	0%
wget	627 K	71 K	5.4 K	0%	0%
ping	2.4 k	1.3 K	241	0%	0%
procps	4 M	1 M	176 K	0%	0%

The collected log consists of syscalls from multiple programs and the size of the log is around 732 GB (without compression) containing 3707 million events. We first separate the log into smaller logs per process.

Table III shows results of the experiment. The second column shows # of events (syscalls) in the log for each program. The third and forth columns represent # of dependencies detected and # of models matched by MCI. For the # of dependencies, we count all those inferred by MCI via matched models and those explicit dependencies across matched models. The last column shows false-positive and false-negative rates.

For most programs, MCI precisely identifies causality with not measurable false-positives and false-negatives. There are a few exceptions: `Firefox`, `CUPS`, `alpine`, and `transmissions`. We manually inspect a subset of these false-positives/negatives and have the following observations. Our Firefox models are intended to describe browser behaviors such as following a hyperlink and opening a tab. However, logs contain a lot of syscalls generated by the page content. Some of them are not much distinguishable from browser-intrinsic behaviors, leading to mismatches. For `CUPS`, we identify new behaviors during the experiment which are variations of the existing models. `Transmission` is a threaded program with memory based synchronizations that are invisible to MCI. Hence, MCI misses some thread interdependences via memory.

Comparison with BEEP. To evaluate the effectiveness of MCI when compared with BEEP, we randomly select 100 system objects (e.g., files or network connections) accessed in the week-long experiment. For each selected system object, we construct a causal graph by BEEP and by MCI, and compare the two. Table IV shows the results. First of all, we observe that MCI has fewer false-positives and false-negatives. Again, we use LDX as the ground truth. Especially, MCI reduces the false-positive rate significantly. We investigate some of the cases that BEEP introduces false-positives, and find that many system objects accessed in a unit are included in the causal graphs while they are not causally related. Also, BEEP causes slightly more false-negatives due to missing inter-unit dependencies. We analyze the cases and find that the missing inter-unit dependencies were due to incomplete instrumentation caused by the difficulty of binary analysis in BEEP. We also manually investigate false-positive and false-negative cases from MCI. It turns out they are mostly caused by concurrent executions in `transmission`.

TABLE IV. COMPARISON WITH BEEP.

	System subjects	System objects	Edges	FP / FN
BEEP	9.23	33.71	74.21	12.8% / 0.3%
MCI	9.18	25.38	62.87	0.1% / 0.1%

Runtime/memory Overhead. We also measure runtime overhead and memory overhead of MCI. Specifically, we report how long MCI takes to parse the audit log collected from the one week experiment which contains 3707 millions events. As we discussed in Sec. IV-B, we preprocess an audit log to extract indexes so that the parser can quickly locate skeleton instances. We measure the runtime performance and memory consumption of the trace preprocessor. It takes *4 hours 47 minutes* to preprocess (index) the entire log. The preprocessor occupies around *2.8 GB of memory* on average. The parser first locates segments of the traces and launches automata within the identified segments. We find that the parser spend more time on parsing within the segments. In particular, the parser takes more time when it parses a wrong segment and eventually fails. Note that we parallelize the parsing within a segment to exploit multi-core processors. To parse the log, it takes *around 4 days (95 hours 43 minutes)*, and the parser consumes around *6.2 GB of memory* on average. We consider such one-time efforts reasonable given the huge log size. We leave performance optimization to our future work.

C. Case Studies

In this section, we present a few case studies to demonstrate the effectiveness of our approach in attack investigation.

1) Phishing email and camouflaged FTP server case: In this case, we use a scenario adapted from attack cases that were created by security professionals in a DARPA program [11], to demonstrate how MCI can effectively infer causality in a real-world security incident that happens across multiple programs including `PINE` and `Firefox`.

Attack Scenario. The user regularly uses `PINE` to send and receive emails. At some point, the user receives a phishing email, and she opens it, finds a hyperlink that looks interesting, and hence clicks the hyperlink. `PINE` automatically spawns the `Firefox` browser and the browser navigates to the given hyperlink. The hyperlink leads her to a web-page that contains an FTP server program. As she thinks the program is useful, she downloads the program. Before she closes the Firefox browser, she navigates a few more websites and downloads other files as well. Specifically, she opened 2 more tabs and downloaded 3 more programs.

After she closed the browser, she checked a few more emails and then opened a terminal to execute the downloaded FTP server program. The FTP server is a camouflaged trojan [3]. It normally behaves as a benign FTP server, serving remote FTP requests properly. However, it contains a backdoor which allows a remote attacker to connect and execute malicious commands on the victim computer. After she ran the trojan FTP server program, it served tens of benign FTP user requests with hundreds of FTP commands. A few hours later, the attacker connects to the machine through the backdoor, and modifies an important file (e.g., financial report). Later, the company identifies that the contents of the important file is changed and then hires a forensic expert to investigate the case to identify the origin of the incident.

Investigation. Given the causal models listed in Table II and a system-wide trace collected from the user's system, the forensic expert uses MCI to infer causal relations from the changed file. By matching models over the trace, MCI successfully identifies causality from the initial phishing email to the attacker's connection in the camouflaged trojan. The investigator further identifies that the important file is touched by the FTP server process. However, the file operation does not belong to any model instance. Interestingly, this indicates that the file is not part of regular behaviors, indicating that the FTP server may be trojaned. The investigator then tries to identify how the FTP server is downloaded and executed in the system. MCI reveals that a `Firefox` process downloaded the FTP server binary via y.y.y.y:80 through "LoadURI" and "Download a file" models. MCI further identifies that the `Firefox` process was launched by a `PINE` process when the user clicked a link from an email stored at */var/mail/.../94368.5222* downloaded from x.x.x.x.

We also investigate the same incident with BEEP, and find out that a causal graph generated by BEEP has a number of false-positives. Specifically, as shown in Fig. 17, the causal graph includes n.n.n.n:53 which is resolving the domain name, several other IP addresses from the `Firefox` process, which are from different tabs. Moreover, the causal graph contains other files downloaded from other tabs (*../file1* and *../file2*), two more sockets for internal messaging system (*unix socket*) and XWindow system (*/tmp/.X11-unix*), as well as some database files for storing browsing history (*/.../places.sqlite*).

In contrast, as MCI leverages accurate models generated by LDX, the graph generated by MCI is more accurate and precise without bogus dependencies. We also note that BEEP requires training and binary instrumentation on the end-user site while MCI has no requirements on the end-user site.

2) Information Theft via InfoZip (Zipsplit): In this case, we use another insider attack to demonstrate the effectiveness of MCI. Specifically, an attacker in this case intentionally uses `Zipsplit` to obstruct the investigation of the case as it reads and writes multiple input and output files where dependences between them are difficult to capture by existing approaches. We show how MCI can accurately identify the information flow through the program.

Attack Scenario. In this case, an insider tries to leak a secret document to a competitor company. However, the attacker's company forces all computer systems to enable audit logging system to monitor any attempts to exfiltrate important information. To avoid being exposed, he decides to use `Zipsplit` before sending out the secret. Specifically, he understands that the `Zipsplit` program can compress n files into m compressed files, and traditional audit logs are able to accurately identify causal relations if an input file is compressed to a *single* output file. Hence, the attacker used `Zipsplit` to compress a secret document, *secret.pdf*, as well as two non-secret files, *1.pdf* and *2.pdf*, and generates four output files, *c1.zip−c4.zip*. In this example, the secret file is compressed and distributed into *c1.zip* and *c2.zip*, whereas *c3.zip* and *c4.zip* only contain non-secrets. Then he attached all output files to an email, but before he sent it to the competitor company, he removed *c3.zip* and *c4.zip* from the email and only sent the other two that contain the secret. After that, he deleted all emails histories and compressed files.

A few days later, the company found suspicious behaviors from the attacker's computer. They identified that the secret document was accessed by `Zipsplit`, and some files that may contain the secret were sent out. However, the attacker claimed that the secret document was mistakenly included in `Zipsplit` and he only sent the zip files that contain non-secrets. At this point, the company started to investigate the attacker's machine to identify the source of outgoing files. Note that the investigator is not able to inspect the compressed files or email history as the attacker already deleted them.

Investigation. A forensic expert utilizes MCI to construct causal models for `Zipsplit` and `PINE`. A related model for `Zipsplit` is presented in Fig. 18, corresponding to the "read n files and compress to an output file" behavior. Note that it is context-free as there are two groups of nodes (from the 4^{th} to the 6^{th} and from the 12^{th} to the 16^{th}) that have the same number of repetition. The first group is for reading the meta information of the n input files and the second group is for reading the contents of the files and write to an output file.

MCI matches the models over the audit log collected from the attacker's machine, and it accurately reveals the causality between the secret document and the outgoing message. Fig. 19-(b) presents a causal graph generated by MCI. It shows that the *c1.zip* and *c2.zip* are derived from *secret.pdf*, and they are sent out via `PINE`. In contrast, Fig. 19-(a) shows a causal graph generated by BEEP but it contains many false-positives as BEEP was not able to identify such removed attachments nor causal relations between inputs and outputs of `Zipsplit`. We manually inspect the program to identify the root cause of false-positives. It turns out that `Zipsplit` first compresses input files into a temporary file, then splits it into multiple output files. Hence, BEEP considers the temporary file is dependent on all input files, and the output files are dependent on the temporary file. In other words, BEEP considers all output files are dependent on all input files. Instead, MCI infers precise causality between each input and output file via implicit dependencies annotated in the model.

3) Long running real world applications: In the last experiment, we evaluate MCI on large scale real world workloads. In particular, we use 2 months of NASA HTTP server access logs obtained from [41] as well as 3 months of our institution's HTTP server access logs (from Nov. 2015 to Jan. 2016).

To obtain audit logs from the HTTP access logs, we first emulate the web server environment by crawling all the contents of the original servers. Then, we create a script which connects and accesses the web server according to the access log so that the audit logging system on our server can regenerate logs for our analysis.

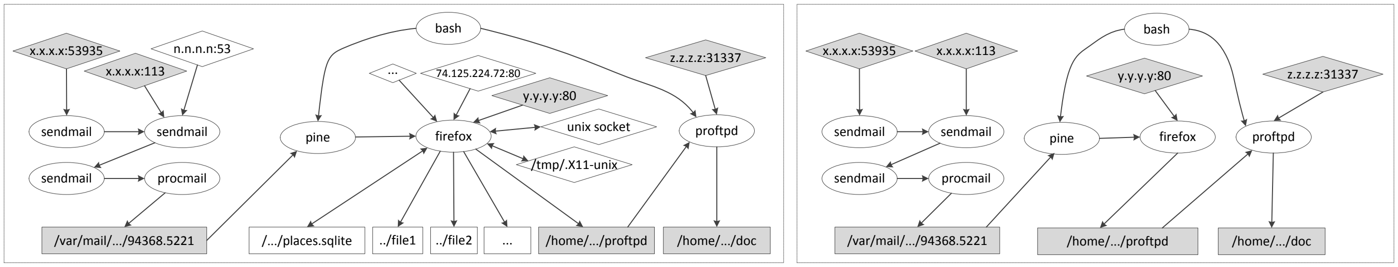
TABLE V. EVALUATION ON LONG RUNNING EXECUTIONS.

Access Log	# of req. (unique)	Elapsed Time	FP / FN
NASA-HTTP [41]	3.4M (36K)	19 hrs 41mins	3.9% / 0.2%
Our institution	5.6M (4.2M)	40 hrs 13mins	1.1% / 0.1%

Table V shows the result. First, our parser takes 19 hours and 40 hours to parse the logs from [41] and our institution, respectively. Considering the size of the logs, we argue that our parser is reasonably scalable. For the accuracy test, we have 3.9% and 1.2% false-positives for the two respective logs. We analyze such cases and find that the NASA-HTTP log includes much more CGI requests than our institution's log. We find that most of the false-positive cases are from those CGI requests (e.g., PHP) that introduce noises. That is, some of the CGI behaviors are similar to the server behaviors and hence confuse our parser. We also have 0.2% and 0.1% false-negative rates. We manually analyze such cases and find out that they are mainly caused by CGI requests and suspicious requests embedding binary payloads, which crash the web-server during the experiment. Overall, the result shows that MCI is scalable to identify causality over large scale logs.

VI. RELATED WORK

Causality Tracking. There exists a line of work in tracking causal dependences for system-level attack analysis [25], [16], [24], [26], [29], [23]. BackTracker [25] and Taser [16] propose backward and forward analysis techniques to identify the entry point of an attack and to understand the damage happened to the target system. Recently, a series of works [32], [37], [36] have proposed to provide accurate and fine-grained attack analysis. Dynamic taint analysis techniques [42], [21], [20] track information flow between taint sources and taint sinks. SME [12] detects information flows between different security levels by running a program multiple times. LDX [31] proposes a dual execution based causality inference technique. When a user executes a process, LDX automatically starts a slave execution by mutating input sources. It identifies causal dependences between input source and outputs by comparing the outputs from the original and slave executions.



(a) Graph generated from BEEP

(b) Graph generated by MCI

Fig. 17. Causal graphs generated from BEEP and MCI for the camouflaged FTP server case.

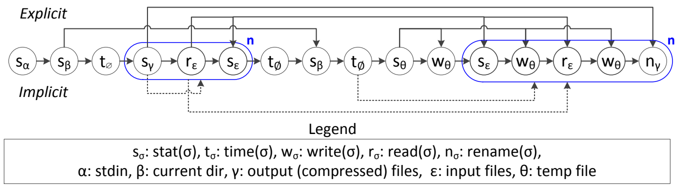
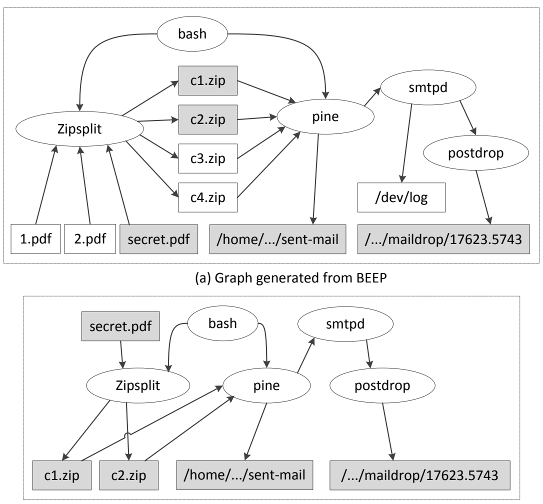


Fig. 18. (Context-free) Model from `Zipsplit`



(a) Graph generated from BEEP

(b) Graph generated by MCI

Fig. 19. Causal graphs for the `Zipsplit` case.

These approaches have limitations, for instance, syscall-based techniques suffer from imprecisions that cause false-positives and false-negatives, unit-based techniques require training or instrumentation on the end-user site, and dynamic taint analysis techniques cause too much runtime overhead. We discussed details of strengths and limitations of those techniques in Section II and compare them with MCI.

Program Behavior Modeling. Constructing program models that represent program's internal structures (e.g., control flow) or behaviors (e.g., system call invocations) have been extensively studied, especially in anomaly detection techniques [46], [28], [14], [48], [50], [47]. Specifically, they train benign program executions to get models which are abstraction of the program behavior. Then, they use various ways such as DFA [28], FSA [46], [14], push-down automaton (PDA) [48], hidden Markov models [50], and machine learning [47], [34]. However, their models are mostly control flow models that do not have dependency information. Having dependences (acquired from LDX) in our models on one hand allows us to use models in attack provenance investigation, on the other hand poses a number of new technical challenges. Due to the difficulty of static binary dependency analysis, generating precise models using static analysis is highly challenging.

VII. DISCUSSION

Kernel-level Attack. We trust audit logs collected at the victim system. Most audit logging systems including Linux Audit and Windows ETW collect and store audit logs at the kernel level, and a kernel-level attack could disable the logging system or tamper with the log. One possible solution is to integrate with LPM-Hifi [6] that provides stronger security guarantees.

Limitations by LDX [31]. In our off-line analysis, we leverage LDX to construct causal models, hence, the limitations in LDX are also inherited by MCI. LDX doubles the resource consumption such as memory, processor and disk storage in order to run a slave execution along with original execution. However, we argue that the limitations only apply to the off-line analysis and do not apply the end-user.

Model Coverage. MCI relies on causal models generated by training with typical workloads. If an audit log includes behaviors that cannot be composed by the models in the provided workloads, MCI may not be able to infer causality precisely and could cause false-positives/negatives. Also, the FPs and FNs caused by missing models may cascade throughout the remaining MCI's parsing process. However, the cascading effect is mostly limited within a unit (e.g., each request in a server program) because MCI nonetheless starts a new model instance when it encounters an input syscall that matches with the model. Moreover, we can detect matching failures due to the incomplete models while MCI is parsing the audit log. For instance, missing models often lead to causal graphs lacking important I/O related system-objects (e.g., files/sockets), hence they are a strong indicator. Then we can enhance the model to resolve the situation by training with more workloads. Furthermore, we can fall back to a conservative strategy to assume unmatched events have inter-dependencies. Although we mitigate the ambiguity problem (Sec. III-C2), as some models may not have enough dependencies to segment traces, ambiguity is still a challenge. We plan to investigate using irrelevant events as delimiters to further partition the trace and suppress ambiguity.

Signal and Exception Handler. Signals and exceptions can be delivered to a predefined handler at anytime, interrupting a normal execution flow. Unfortunately, it is possible that system calls in the handler may affect our parser. However, we observe that in practice our models are robust enough to handle the additional system calls caused by such handlers. This is because system calls invoked in a signal or exception handler are generally distinctive from the system calls in our causal models, hence our parser is able to filter them out. Moreover, in many programs such as `Lighttpd`, handlers functions often do not invoke any system call. In the future, we plan to extend MCI to construct proper models for signal and exception handlers. As such, we can identify handler models from the audit log and extract them before we apply MCI's model parsing process.

VIII. CONCLUSION

We present MCI, a novel causality inference algorithm that directly works on audit logs provided from commodity systems. MCI does not require any special efforts (e.g., training, instrumentation, code annotation) or framework (e.g., enhanced logging, taint tracking) on the end-user. Our off-line analysis precisely infers causality from a given system call log by constructing causal models and identifying the models in a given audit log. We implemented a prototype of MCI and our evaluation results show that MCI is scalable to cope with large scale log from long-running applications. We also demonstrate that MCI can precisely identify causal relations in realistic attack scenarios.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by the United States Air Force and DARPA under contract FA8650-15-C-7562, NSF under awards 1748764, 1409668, and 1320444, ONR under contracts N000141410468 and N000141712947, and Sandia National Lab under award 1701331, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force and DARPA or other sponsors.

REFERENCES

[1] Insider threat spotlight report, 2016. http://crowdresearchpartners.com/wp-content/uploads/2016/09/Insider-Threat-Report-2016.pdf.

[2] Quarterly threat report. https://www.solutionary.com/threat-intelligence/threat-reports/quarterly-threat-reports/sert-threat-report-q4-2016/.

[3] proftpd-1.3.3c-backdoor. https://www.aldeid.com/wiki/Exploits/proftpd-1.3.3c-backdoor, 2011.

[4] Trends from the years's breaches and cyber attacks. https://www.fireeye.com/current-threats/annual-threat-report/mtrends.html, 2017.

[5] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. *SOSP'09*.

[6] A. Bates, D. Tian, K. R. B. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *SEC'15*.

[7] M. Braun. Gnupg vim plugin. https://github.com/jamessan/vim-gnupg/blob/master/plugin/gnupg.vim, 2017.

[8] A. Cahalan. procps. http://procps.sourceforge.net/, 2009.

[9] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, September 1956.

[10] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, June 1959.

[11] DARPA. Transparent computing. https://www.darpa.mil/program/transparent-computing, 2017.

[12] D. Devriese and F. Piessens. Noninterference through secure multi-execution. *SP'10*.

[13] dtrace.org. Dtrace. http://dtrace.org/blogs/, 2017.

[14] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *SP'03*.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

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

[17] S. Grubb. Redhat linux audit. https://people.redhat.com/sgrubb/audit/.

[18] P. Institute. 2016 cost of data breach study. https://app.clickdimensions.com/blob/softchoicecom-anjf0/files/ponemon.pdf.

[19] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. *CCS'13*.

[20] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for effciently accelerating software-based dynamicdata flow tracking on commodity hardware. *NSDI'12*.

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

[22] W. M. Khoo. Taintgrind. https://github. com/wmkhoo/taintgrind, 2017.

[23] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu. Dual execution for on the fly fine grained execution comparison. *ASPLOS'15*.

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

[25] S. T. King and P. M. Chen. Backtracking intrusions. *SOSP'03*.

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

[27] W. Koch. The gnu privacy guard. https://gnupg.org/, 2017.

[28] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5), Sept.

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

[30] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *SIGCOMM'06*.

[31] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu. Ldx: Causality inference by lightweight dual execution. *ASPLOS'16*.

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

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

[34] Z. Li and A. Oprea. Operational security log analytics for enterprise breach detection. In *SecDev'16*.

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

[36] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. *SEC'17*.

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

[38] Microsoft. Event tracing for windows. https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx, 2017.

[39] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: Pipelined symbolic taint analysis. *SEC'15*.

[40] M. Muuss. Ping c program. http://ws.edu.isoc.org/materials/src/ping.c.

[41] NASA. Nasa-http - two months of http logs from the ksc-nasa www server. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html, 1995.

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

[43] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. *ATC'14*.

[44] V. Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, 31(23-24), Dec.

[45] M. Roesch. Snort. https://www.snort.org/, 2016.

[46] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. *SP'01*.

[47] X. Shu, D. Yao, and N. Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. *CCS'15*.

[48] D. Wagner and D. Dean. Intrusion detection via static analysis. *SP'01*.

[49] Willem. C implementation of the raft. https://github.com/willemt/raft.

[50] K. Xu, K. Tian, D. Yao, and B. G. Ryder. A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity. *DSN'16*.