2019 IEEE European Symposium on Security and Privacy (EuroS&P)

Discovering Correlations: A Formal Definition of Causal Dependency Among Heterogeneous Events

Charles Xosanavongsa
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA, Thales Six GTS France
charles.xosanavongsa@supelec.fr

Eric Totel
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
eric.totel@centralesupelec.fr

Olivier Bettan
Thales Six GTS France
olivier.bettan@thalesgroup.com

Abstract*—In order to supervise the security of a large infrastructure, the administrator deploys multiple sensors and intrusion detection systems on several critical places in the system. It is easier to explain and detect attacks if more events are logged. Starting from a suspicious event (appearing as a log entry), the administrator can start his investigation by manually building the set of previous events that are linked to this event of interest. Accordingly, the administrator attempts to identify links among the logged events in order to retrieve those that correspond to the traces of the attacker's actions in the supervised system; previous work is aimed at building these connections. In practice, however, this type of link is not trivial to define and discover. Hence, there is a real necessity to describe and define formally the semantics of these links in literature. In this paper, a clear definition of this relationship, called contextual event causal dependency, is introduced and proposed. The work presented in this paper aims at defining a formal model that would ideally unify previous work on causal dependencies among heterogeneous events. We define a relationship among events that enables the discovery of all events, which can be considered as the cause (in the past) or the effect (in the future) of an event of interest(e.g., an indicator of compromise, produced by an attacker action). This model is gradually introduced and defined by merging two previously defined causality models from the distributed system and operating system research areas (i.e., Lamport's and d'Ausbourg's). Our model takes into consideration heterogeneous events that emanate from different abstraction layers (e.g., network, system, and application) with the main objective of formally defining a causal relationship among logged events. Thereafter, we show how existing implementations separately allow the computation of parts of the model. Finally, we describe the implementation and assessment of the model according to real attacks on distributed environments and its accuracy to extract all causally linked events related to a given attack event trace.

***Index Terms*—alert and event correlation, multi-step attack discovery, formal model, causal dependencies, distributed systems, forensic**

I. Introduction

Competitive and complex enterprise environments are prone to industrial spying, sabotage, intrusion, and data theft. The alarming number of vulnerabilities that has been reported over the last year [2] highlights the reality of the threats of targeted attacks and their consequences relative to the financial aspects and reputation of the enterprise. To achieve their goals, it is generally necessary for attackers to perform several consecutive actions. Such attacks are known as multi-step attacks and can potentially remain undetected because each step can typically be considered normal until the ultimate intrusion objective is realized; at that point, hopefully, it can be detected. A typical attacker usually succeeds in gaining a foothold inside the target system using social engineering techniques, such as phishing email, watering hole, or Trojan software. Thereafter, he maintains his foothold through the deployment of command and control channels, explores the network, and attempts to elevate his privileges by exploiting vulnerabilities or stealing passwords. The final stage, i.e., the attack objective, often consists of sensitive data leakage or sabotage.

The current intrusion prevention techniques are not sufficient to repel targeted attacks that use authorized communication channels to reach the targeted machine. Hence, it is necessary to deploy sensors inside the supervised system to be able to thwart any action performed by the attacker. Unfortunately, logged events[1] related to his actions are typically scattered across different machines, log files, and databases of the intrusion detection system (IDS) alerts. Moreover, the various steps of the attack can occur over a large time window. The alerts produced by any deployed IDS are symptomatic of malicious activities. However, an IDS frequently relies on the analysis of a single type of data source. A network-based IDS (NIDS) relies on network packet analysis, whereas host-based IDS (HIDS) solutions may supervise a given application (e.g., the system calls it produces) or the system's access control policy, such as OSSEC [6]. Unfortunately, an alert rarely explains the context of the detected attack; hence, the correctness of an alert must be investigated. Indeed, even if an alert is a consequence of an attack step, it only indicates a small portion of the overall attack, which is also composed of footprints that are not sufficiently suspicious to trigger an alert and might even appear benign and legitimate in the system when initially considered. Accordingly, analysts have to verify and contextualize an alert by manual techniques to determine whether or not it is related to a step of an attack scenario; they manually attempt to correlate IDS alerts and events logged by various sensors, and contextual information.

The discovery of correlations among different sources of information is the key to identify the attack steps [46]. Automatic tools are indispensable in aiding analysts to comprehend an

[1]In this paper, any logged information produced by applications, sensors, and IDSs is an event. When the event is produced by an IDS, we also refer to it as an alert.

DOI 10.1109/EuroSP.2019.00033

attack and ascertain its root causes and impacts by identifying all the compromised assets. Such tools would enable them to discern the attack and implement actions to avert similar scenarios in the future. Every single instance of log may contain valuable information in order to better comprehend an ongoing attack or intrusion. The typical approach to correlate events and alerts is to rely on a base of correlation rules that explicitly describe the logged events and alerts, which are parts of the consequence of attacks inside the supervised system [19], [20]. This technique has been adopted for years in security information and event management (SIEM) tools. However, the formulation of correlation rules can be considerably difficult because it necessitates precise knowledge of the supervised system (i.e., its topology and cartography) and deployed IDSs. Evidently, the aforementioned approach has reached its limits and new methods must be defined to automatically discover the relationships among the attack traces on the supervised system. An action might be observed by sensors in the supervised system. Thus, there is a cause–effect relationship between the action and its observations, i.e., the related logged events. Moreover, the order of the attack's actions is generally meaningful. The actions are also linked and there are cause–effect relationships between them. Consequently, the logged events related to each successful attack steps are also causally dependent. In this paper, a definition of the notion of causal dependency among entries in heterogeneous log files is presented. With this, the objective is to aid security analysts investigate suspicious log entries, such as an IDS alert, by computing all log entries they causally depend on and all entries that are causal consequences of identified malicious events.

Computing causal dependency among events is a difficult task and has been studied for years, especially in distributed systems [43]. To discover causal relationships among actions of distributed processes in these types of systems, several methods have been introduced. Regarding security, certain definitions have also been introduced to reflect object state causal dependencies [11]. This is well-illustrated by methods that track information flows between the subjects and objects of an operating system (OS). Indeed, if an information flow between two objects is observed, it can be defined that their states are causally linked. Recently, it has been observed that a considerable amount of work has been focused on causal dependency inference but not on explicit causal dependency computations. Such an inference is performed by attempting to identify links [26] among events, mining data to discover patterns (such as temporal invariant properties) [8], or by computing a measure of similarity among event attributes. In practice, with these types of approaches, the events are expected to be causally dependent when they are tightly linked.

The main difficulty in finding causal dependencies among heterogeneous events is that it aims to merge different points of view: an application embeds the business logic, whereas an OS only sees requests from user space applications through system calls. Additionally, network packet analysis and alert production, either from NIDS or HIDS, are difficult to reconcile. This illustrates the semantic gaps among different layers of abstraction and among different data sources.

In order to overcome these problems, we propose a formal definition of the causal dependency relationships among events emanating from heterogeneous logs (i.e., logs containing events of different natures). The relationships are deduced from the causal dependency relationship among events that are referred to as *contextual events*. By leveraging the causal dependency among events, a graph of causally dependent events and alerts found in heterogeneous logs that would explain the alert can be built for each identified malicious alert. In Section II, the state of the art in this field of research is described. In Section III, an overview of the problematic and approach (described in a later section) is provided. In Section IV, a new model that features the definition of what causal dependency is among process actions in the system is introduced. It further explicates the extrapolation of the proposed model after these actions have been linked to logged events in order to deduce a causal dependency relationship among heterogeneous events. This section describes that starting from a suspicious event, i.e., an alert triggered by an IDS or an indicator of compromise (IoC), this new model makes it possible to build the set of logged events that cause the IoC and the set of events that results from the IoC. The section also presents how the set of all the events related to a given attack can be subsequently deduced. Section V elaborates how our proposed model can be approximated using existing works. Thereafter, a computation technique for approximating the model without relying on any instrumentation is proposed in Section VI. The assessment of the model and how its implementation would enable the discovery of logged events in a small set of attacks is demonstrated in Section VII. This leads to further discussions in Section VIII. Finally, the conclusions are summarized in Section IX.

II. State of the Art

Various event correlation methods have been developed to aid security analysts investigate alerts. Several approaches have been proposed to enable analysts to explicitly express a dreaded scenario according to an explicit event correlation rule using an attack description language [44], [13], [20], [10]. These rules are then interpreted by a correlation engine that would analyze the stream of events and obtain those described in the correlation rules. In practice, event correlation rules are considerably difficult to formulate because security operators have to combine the perspective of the attacker (by building possible attack scenarios) and that of the defender (i.e., by having a precise knowledge of the system to defend). The combination of these points of view enables them to project the attack steps onto the set of observable events that the sensors can produce; accordingly, they can explicitly write the sequence of events that represents the attack. A recent work [19] proposes to simplify the process of event correlation writing by dissociating the attack scenario specification from the knowledge of the supervised system and leveraging a knowledge database [37]. This database contains organized information on the supervised system, i.e., its topology, cartography, vulnerabilities, and deployed sensors. Using a similar database, other approaches propose to automatically generate attack graphs [23], which represent all possible attack paths in the supervised system. These graphs rely on the knowledge of the vulnerabilities from which explicit alert correlation rules that describe multi-step attack scenarios can be deduced. Nevertheless, the chance that unknown attack steps can be missed by IDSs remains. Despite this limitation, reference [28] proposes a correlation engine that is able to detect incomplete attack scenarios. Unfortunately, a majority of approaches only correlate alerts, particularly NIDS alerts. However, such events are not sufficient to conduct an attack analysis, and attackers attempt to be as stealthy as possible to avoid triggering alerts. Moreover, because of the difficulty in writing correlation rules, the development of automatic event correlation tools becomes necessary.

Information flow tracking (IFT) techniques are also employed to compute explicit causal dependencies. Applied to the OS layer, information flows are deduced from system calls to build information flow graphs [25], [31], where nodes represent kernel objects, and directed edges represent information flow from one node to another. Because every application requires the use of system call interface to use kernel services, it is a good observation point to log the behavior of applications. It has been widely used for decades, e.g., in the work of Forrest [16]. The provenance research field [36], [17] is also closely related to causality analysis as it aims to answer the two following questions. Where does a given object come from? What are the objects that it influences? Applied to the OS, provenance can be represented as a branch of the IFT research field. Using backward and forward tracking, all these approaches make it possible to locate potential entry points and identify potential consequences of the attack. These methods can be applied using commercial off-the-shelf kernel logging frameworks, such as *auditd* [4] or *event tracing windows* [3], which allow the logging of system calls. Some works propose the alternative instrumentation of the kernel with an optimized design that includes security considerations [34], [7], [40]; other approaches couple the IFT with tag propagation policies [22] [18]. Tag propagation-based policies can be coupled to access control policies or attack detection policies; these may make it possible to achieve a finer-grained causality tracking. However, the cost of tag propagation and tracking is high because the number of tags that follows constantly increases.

To our knowledge, few approaches offer finding causality links among *heterogeneous events*. Because multi-step attack scenarios potentially involve several applications, machines, and network communications, we argue that different types of logs have to be taken into consideration to better comprehend an incident. In [41], the authors propose to model the discovery of multi-step attack scenarios as a community discovery problem inspired from the social network domain to infer causality among heterogeneous events. The provenance layering approach [38] provides a means to build the value causality links among some object states of a supervised system. These states could be used in our approach to compute a part and an approximation of our model. However, we advance the idea by defining the implication of the causality links among object states on the logged event causality link definition; this aspect is not provided by the provenance work.

As a conclusion, methods for discovering correlations between events and alerts generated by an attacker's activity in several heterogeneous log remain necessary. The solution we propose is to formally define a causal dependency relationship between logged events and alerts. As described in Section IV, this model provides a unified understanding of the causality relationships that can be defined between active entities, passive entities, and logged events. Once this model is defined, it is shown in Section V that previous works can be used conjointly to provide sufficient information and build an approximation of the model. Thereafter, in Section VI, it is shown that it is possible to build an approximation of the model that does not rely on any system instrumentation; moreover, it permits the obtainment of causal dependency relationships among heterogeneous logged events using only existing and available logging facilities to record events.

III. Illustration of the Problematic and Proposed Model

In this section, the concept of causality analysis in heterogeneous and distributed event logs via a motivating attack scenario example is introduced. In a later section, this example is used to further illustrate how each type of logged event is treated in our model.

The web server architecture is composed of two machines, which host the Apache server and a MySQL database. The administrator deploys several sensors to collect data and perform intrusion detection on the servers. Specifically, the servers are equipped with the Linux audit framework *auditd* [4]. The Apache and MySQL logging modules are activated for the web server and database server, respectively. Moreover, the database server is equipped with Bro NIDS [1]; auditd and *netfilter* [5] are particularly configured to record system calls of interest and established connections, respectively.

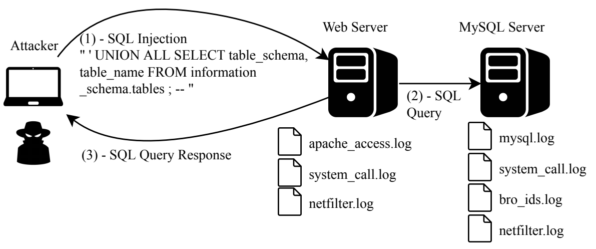


Fig. 1. SQL injection attack scenario on a vulnerable web server.

The attack scenario, illustrated in Fig. 1, is described as follows. After a discovery step, a malicious user performs an SQL injection via the POST parameters on an HTTP request in order to obtain the database table scheme. The analyst is alerted by a Bro alert, which describes an SQL injection attempt. Because this behavior is considerably suspicious, the analyst decides to investigate the alert.

Because the SQL query is initiated by the web server, the analyst immediately checks the events logged by the Apache server. However, because the Apache web server is not configured to record POST parameters, the analyst is unable to identify the web request related to the SQL query; hence, he cannot determine the requests issued by the attacker. Finding no other clues in the Apache logs, he decides to perform an analysis of the recorded system call logs produced by auditd in both machines by manually backtracking the sequence of system calls that led to the SQL query observed over the network. With such an analysis, he eventually retrieves the socket from which the SQL query originated. The information contained in this network entry point allows the analyst to identify the attacker's IP address. Finally, the investigation of the Apache and system call logs reveals the rest of the traces related to the attacker's IP. This simple example illustrates the fact that the attacker's footprints are scattered across different types and formats of event logs. Evidently, the network, applications, and system events emanating from different machines must be investigated to understand the full picture of the attack.

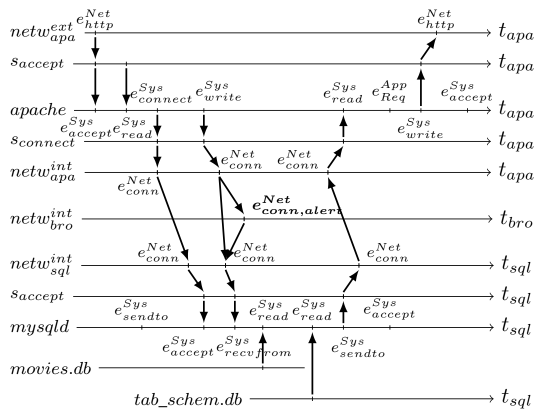


Fig. 2. Visualization of logged events, alerts, and information flows on a space–time diagram.

If the analyst could automate this task, then he would have to specify that he intends to retrieve the different logged events that capture the attacker's activity. These events are scattered in various heterogeneous logs. For simplicity, he can draw a space-time diagram, such as that illustrated in Fig. 2, which captures the semantics of different logs of interest. In this figure, the timelines of all active and passive entities that are part of the attack can be observed. Each logged event is interpreted as representing the relationship among these entities and is placed on the timeline of the entity it describes. The different types of logged events are also indicated using the following: (1) e^{Net} for log entries deduced from network packet flow analysis, e.g., $e^{Net}_{conn,alert}$, which represents an alert raised by the Bro NIDS; (2) e^{Sys} for system call log entries, e.g., e^{Sys}_{accept}, which represents the invocation of an accept() system call; (3) e^{App} for application log entries, e.g., e^{App}_{Req}, which represents the Apache application logged event for the HTTP request. In most cases, the logged events pertain to relationships between two entities. In reading this space diagram by backtracking starting from the alert or the IoC, it can be deduced that several process activities can explain the attacker's steps. In the following sections, we formally present these relationships as causal dependencies among log entries and define a model that permits the generation of causality links among logged events.

In this paper, all concepts are illustrated in Section VI using the example used in this section. For clarity, only the principal events of interest that allow the comprehension of the segment of the attack scenario related to the raised alert are presented.

IV. Defining a Causal Dependency Relationship among Logged Events

The purpose of this section is to explicate the concept of causal dependency among logged events. Starting from the definitions of contextual actions and the causal dependency that links them, we gradually define the notions that are necessary to link the contextual actions to their corresponding logged events and finally be able to describe causal dependencies among logged events.

The notion of causal dependency is not new; it has been the subject of numerous works [43] particularly in the field of distributed computations. However, these works focus on the dependence of distributed application process actions[2], i.e., actions deduced from application logs and message exchanges among them. These dependence computations frequently rely on logical clocks [15], [35]. Consequently, the relationship captured among different actions is one that is temporal (e.g., the happened-before relationship defined by Lamport [27]). However, in the security field, certain extended relationships have been defined according to the fact that information flows exist among system entities; thus, their states are interdependent. This is particularly the case of the dependency relationship defined by d'Ausbourg [11]. In the two following subsections, our attempt to emphasize the particularities of these two types of models is presented. Thereafter, the merging of these two approaches in order to exploit them and define a new causality dependency notion, called *contextual action causal dependency* relationship, is explained.

A. Defining Causal Dependency among Contextual Actions

1) Lamport's Happened-Before Relationship among Events: In [27], Lamport emphasizes that it is impossible to capture a total ordering of actions in a distributed computation. Indeed, most actions that arise in distributed systems cannot be ordered because we cannot rely on a global clock. In practice, the relationship defined by Lamport, called *happened-before*, is a partial order relationship on the set of

[2]An action is different from a logged event. An action can occur without being logged.

actions that are performed by concurrently executed processes. More precisely, a distributed computation is considered as a collection of processes that can communicate through message exchanges. Each process produces a sequence of totally ordered events that are process actions, such as function calls and sending–receiving messages.

The happened-before relationship, denoted by "$\prec$" on the set of actions of a distributed computation, is a partial ordering relationship defined as follows. Given two logged application actions, a and b, $a \prec b$ is true

1) if a and b are actions produced by the same process, and a comes before b;
2) or if a is the sending of a message m by a process, and b is the receipt of the same message by another process;
3) or if $\exists\ c\ /\ a \prec c$ and $c \prec b$.

Two distinct actions, a and b, are regarded as concurrent (denoted $a||b$) if $a \not\prec b$ and $b \not\prec a$. The advantage of the happened-before relationship is that it does not rely on a global clock to order the actions; this is particularly important in the context of distributed systems where having local clocks synchronized is particularly difficult.

In his paper, Lamport states that $a \prec b$ means that it is possible that action a causally affects action b; evidently, this is not always true. In practice, given an action b for all actions a so that $a \prec b$, it is typically assumed that b is causally dependent on a; this is an over-approximation of the set of actions that actually causally influence b. This is because Lamport's relationship is one that is temporal and does not take into consideration the context value dependency in which the actions are produced. Moreover, in this model, only application level actions that are produced by concurrent processes are covered. Consequently, not all system-level actions or network actions can be taken into account by the model; thus, causal dependencies among heterogeneous event logs cannot be explicitly computed.

2) D'Ausbourg's Causal Dependency Relationship among Object States: In [11], d'Ausbourg describes the relationship of causal dependency, denoted by "$\rightarrow$," among the states of the system. A state (o, t) is the value of an object o at a given time t. Formally, by stating that the state (o, t') *causally depends* on the state (o, t), i.e., $(o, t) \rightarrow (o, t')$, means that the value of o at time t, denoted (o, t), is used to generate the value of o at time t', denoted (o, t'). Actions performed on the system can imply the evolution of a state, producing a new state that is considered as *causally dependent* on the previous state of the same object. In fact, states are causally dependent if and only if an information flow occurs among these states. The concept of an object is considerably flexible; it can be a variable in a program execution or a system process, file, socket, etc.

As opposed to Lamport's model, the claim that all actions of a process are causally dependent across time is false in the d'Ausbourg's model. The following is an illustrative example. Consider a variable a as an object. The code "$(action\_1, 1)\ a := 1;\ (action\_2, 2)\ a := a + 1;\ (action\_3, 3)\ a := 0$" yields three states of a, with $(a, 1) = 1$, $(a, 2) = 2$, and $(a, 3) = 0$, respectively. As a result, $(a, 1) \rightarrow (a, 2)$, and $(a, 2) \not\rightarrow (a, 3)$ because the value of a at line 3 is independent of the value of a at line 2. In the Lamport model, we would state that $action\_1 \prec action\_2$ and $action\_2 \prec action\_3$, which involve the causal dependency between $action\_2$ and $action\_3$, simply because these actions are consecutive. This is of course meaningless if the context values (in this case, the value of variable a) of different actions are to be taken into consideration.

The d'Ausbourg causal dependency relationship is applied to object states and not on events produced by applications, systems, network captures, etc. Thus, it is extremely difficult to use this model because the states are not easily captured by applications or system executions, where only logged events are the actions and information captured by supervision mechanisms. Accordingly, we must define a model that takes into account both the action causal dependencies in time (similar to Lamport's model) and the contextual states in which these actions are produced (similar to d'Ausbourg's model). The purpose of the *contextual action causal dependency* model that is presented in this section is to capture the advantages of both Lamport and d'Ausbourg models. To this end, we have to consider not only the actions performed by processes, but also the context of the process at the moment the action is executed. This leads to the definition of what we refer to as a *contextual action*.

3) Contextual Action Definition: A contextual action is composed of (1) the action performed by an object, and (2) the value of the context of the object at the time at which the action is performed. In order to resolve the problem of dependency among heterogeneous objects, the objects between active objects (processes or network that produce actions) and passive objects (information containers, such as files, sockets, memory, and pipes that do not produce any action) must be distinguished. An active object is supposed to produce actions that can be linked to a context of the object. For a passive object, only its context can be observed and no action is produced. In order to have a unique notation for both types of objects, an action a is that performed by a process if the object is active and $\emptyset$ if an object is passive (i.e., no action is produced by a passive object). Note that if we consider the state of an active object at a given time without running an action, $\emptyset$ is the precise notation used to indicate that no action is performed in this context. The set of actions produced by an active or passive object, o, is defined as the set $ObjectActions(o)$.

Definition 1: $ObjectActions(o) = \{a_i\} \cup \{\emptyset\}$ with a_i are the actions that can be performed by the object, o, and the absence of action, $\emptyset$.

As an example, a process p has to call a function in a library to invoke a system call to request a kernel service. We then consider that all library function invocations are actions of $ObjectActions(p)$. We now formally introduce the concept of contextual action:

Definition 2: A *contextual action* is a couple $(a, (o, t))$ where $a \in ObjectActions(o)$, and (o, t) is the state of object o at time t.

Lamport supposes that for two actions, a and b are produced by a given process such that $a \prec b$, b is causally dependent on a. As previously noted, it is desired to realize a more precise model that can define that at a given moment, the causality relationship between a and b inside a given object evolution can be broken if the state of the object is independent from its previous state. In practice, numerous server processes keep no memory in their sequenced request executions. For instance, a network file server process can access any file without the knowledge of the previous access. This means that a given process execution can be divided into temporal intervals, where executions are partially or completely "independent" from each other. In our model, such an interval in the execution of a process is called a *session*. The concept of session is not new. In particular, in [29], actions that delimit sessions inside long-running processes have been thoroughly studied to reduce the number of false causal dependencies among actions. The definition of the notion of session is as follows.

Definition 3: Given an object o, a session $Session_n(o)$ is a sequence of contextual actions $(a_i, (o, t_i))$, where $a_i \in ObjectActions(o)$ and $Session_n(o) = \{(a_i, (o, t_i)) \mathbin{/} (o, t_i) \rightarrow (o, t_{i+1})$ and $(o, t_{end_{n-1}}) \not\rightarrow (o, t_{start_n})$ and $(o, t_{end_n}) \not\rightarrow (o, t_{start_{n+1}})\}$; t_{start_n} is the time of the first contextual action of $Session_n(o)$, and t_{end_n} is the time of the last contextual action in $Session_n(o)$.

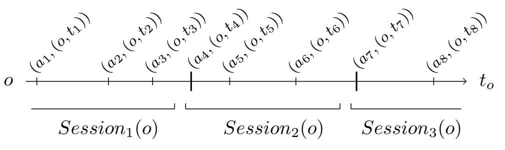


Fig. 3. Sequence of contextual actions and sessions.

An execution of an object o is the union of all $Sessions(o)$, as shown in Fig. 3. The figure also illustrates how different sessions of an object are built on its timeline. In this example, the actions, a_4 and a_7, start new sessions. Thus, (o, t_4) and (o, t_7) are independent of their temporally previous states. In practice, such actions can be identified with expert knowledge or the use of underlying mechanisms of applications. The concept of sessions applies to any object type, e.g., a file that is emptied or an Apache process that has no memory between two execution requests. An action that starts a new session can actually be performed by an application or the OS. This can be illustrated for passive objects with a shared memory that can be cleared by the system or another process and whose state becomes independent from its previous states.

4) Definition of Contextual Action Causal Dependency Relationship: The concept of contextual action takes into consideration the actions performed by the objects and their states involved in these actions. It allows us to exploit both models and define a more precise dependency relationship among heterogeneous actions. This new dependency relationship is called *contextual action causal dependency*, denoted "$\mapsto$," and is defined on the set of all contextual actions produced by all objects in the system.

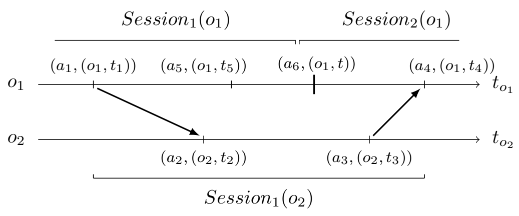


Fig. 4. Contextual action causal dependency in different sessions.

Definition 4: Given two contextual actions, $(a_1, (o_1, t_1))$ and $(a_2, (o_2, t_2))$, the latter being causally dependent on the former, written as $(a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2))$, is true

1) if o_1 and o_2 are the same object o, $\exists\ n$ so that $(a_1, (o, t_1)) \in Session_n(o)$, $(a_2, (o, t_2)) \in Session_n(o)$, and $t_1 < t_2$;
2) or if $o_1 \neq o_2$, $(o_1, t_1) \rightarrow (o_2, t_2)$, i.e., they are causally dependent in the sense of d'Ausbourg, indicating that there is an information flow from the state (o_1, t_1) to the state (o_2, t_2);
3) or if $o_1 \neq o_2$, action a_1 corresponds to the sending of a message, m, and the action a_2 corresponds to the reception of m, implying that $a_1 \prec a_2$ using case (2) of Lamport's happened-before relationship;
4) or $\exists\ (c, (o, t))$ so that $(a_1, (o_1, t_1)) \mapsto (c, (o, t))$ and $(c, (o, t)) \mapsto (a_2, (o_2, t_2))$.

In practice, for a given object, o, two contextual actions, $(a_1, (o, t_1))$ and $(a_2, (o, t_2))$, are causally dependent if their states are causally dependent in the sense of d'Ausbourg causality definition, i.e., if $(o, t_1) \rightarrow (o, t_2)$. This implies that if $(a_1, (o, t_1))$ and $(a_2, (o, t_2))$ are part of the same session, then they are causally dependent. However, if $(a_1, (o, t_1))$ and $(a_2, (o, t_2))$ are not in the same session, then they can either be causally dependent or independent. This is clearly different from the Lamport causality definition, where all actions performed by object o are considered as causally dependent even if they belong to different sessions.

Figure 4 illustrates the use of our model. Given two objects, o_1 and o_2, we have the following relationships among the different contextual actions: $(a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2)) \mapsto (a_3, (o_2, t_3)) \mapsto (a_4, (o_1, t_4))$; $(a_1, (o_1, t_1)) \mapsto (a_5, (o_1, t_5))$; $(a_6, (o_1, t)) \mapsto (a_4, (o_1, t_4))$. Because the relationship "$\mapsto$" is transitive, we have $(a_1, (o_1, t_1)) \mapsto (a_4, (o_1, t_4))$ even if these two contextual actions belong to two different sessions. Moreover, as action a_6 starts a new session, it is implied that $(a_5, (o_1, t_5)) \not\mapsto (a_6, (o_1, t))$. Note that objects o_1 and o_2 have their own clocks, i.e., t_{o_1} and t_{o_2}, respectively; they are not necessary to synchronize our model.

B. From Contextual Actions to Contextual Events and Timestamped Logged Events

The previous subsection describes the notions of contextual actions and the causal dependencies among them. In practice, the actions can be recorded or unrecorded as logged events produced either by an application or a sensor (e.g., at the OS level or network level). The objective of our model is to define a causal relationship among these events.

1) Contextual Event Definition: As in [14], an event is defined as "an identifiable action that happens on a device and is recorded in a log entry." Note that an action might not be recorded in a log entry (e.g., it is not observed by any sensor); consequently, an action can be missed by an analyst. Several sensors can be deployed in a supervised system; thus, an action might be observed by different sensors and be recorded as several log entries in heterogeneous logs.

Given the set of the system's logged events, denoted $\mathbb{E}$, each logged event is produced at a given time (e.g., its timestamp) by observing a given contextual action performed by an object. This leads to the definition of a *contextual event*.

Definition 5: A *contextual event* is a triplet (e, o, t_e), where $e \in \mathbb{E}$; o represents the observed object, and t_e is the timestamp of event e.

According to definition 2, action a of a given contextual action, $(a, (o, t_a))$, might represent a real action or the lack of action; hence, a might not be observable. We thus extend the previous definition by introducing the contextual event, $(\emptyset, o, t_a)$, corresponding to the lack of observation of a at time t_a.

We can now introduce a function, *Obs*, which maps a contextual action into a set of contextual events corresponding to the observations of this single contextual action. The function *Obs* can be defined as follows.

Definition 6: Given an action $a \in ObjectActions(o)$ occurring at time t_a, the observation of a *contextual action* is $Obs\big((a,(o,t_a))\big) = \{(e_i, o, t_{e_i})\} \cup \{(\emptyset, o, t_a)\}$, where $e_i \in \mathbb{E}$ and is an observation of a; $(\emptyset, o, t_a)$ corresponds to the lack of an observation of a and thus to the lack of an event.

It must be noted that in definition 6, the timestamp of event t_{e_i} might be different from time t_a at which the related action is actually executed. We have no clue whether $t_a < t_{e_i}$ or not because the action can be recorded before it occurs, during, or after its execution. Moreover, the type of an action and its observations are the same as that of the observed object (i.e., for active objects, process or network type).

2) Definition of the Contextual Event Causal Dependency Relationship: Recall that the goal of this model is to define the causal dependency relationship among events. To accomplish this, it is first necessary to define the *contextual event causal dependency* relationship, denoted as "$\rightharpoonup$."

Definition 7: Given two contextual events (e_1, o_1, t_{e_i}) and (e_2, o_2, t_{e_j}), the latter is causally dependent on the former, written as $(e_1, o_1, t_{e_i}) \rightharpoonup (e_2, o_2, t_{e_j})$, if and only if there exist two contextual actions, $(a_1,(o_1,t_1))$ and $(a_2,(o_2,t_2))$, such that $(a_1,(o_1,t_1)) \mapsto (a_2,(o_2,t_2))$ and $(e_1,o_1,t_{e_i}) \in Obs\big((a_1,(o_1,t_1))\big)$ and $(e_2,o_2,t_{e_j}) \in Obs\big((a_2,(o_2,t_2))\big)$.

3) Definition of Event Causal Dependency: At this point, the core result of this model is obtained, i.e., the definition of what we call the *event causal dependency* relationship, denoted "$\triangleright$."

Definition 8: Given two events, e_1 and e_2, the latter is causally dependent on the former, written as $e_1 \triangleright e_2$, if and only if $(e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2})$, where o_1 and o_2 are the observed objects, and t_1 and t_2 are the timestamps of events, respectively.

The relationships "$\mapsto$," "$\rightharpoonup$," and "$\triangleright$" define partial orders on the set of contextual actions, the set of contextual events, and the set of events, respectively.

C. Cause and Dependence Graphs for Events

The relationships defined in this section are transitive. This property allows us to build the *cause graph* and *dependence graph* of a given event, e, of interest. The cause and dependence graphs represent all events that contribute to and depend on a given event, respectively.

Definition 9: The *cause graph* is defined as $cause(e) = \{e'/ e' \triangleright e\}$

Definition 10: The *dependence graph* is defined as $dep(e) = \{e'/e \triangleright e'\}$

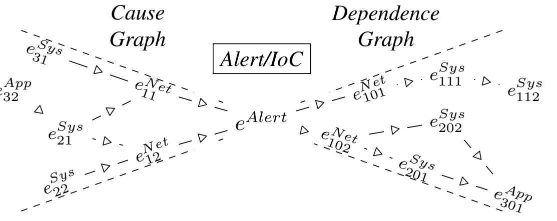


Fig. 5. Cause and dependence graphs of an event of interest.

Figure 5 illustrates the cause and dependence graphs of a given event of interest, i.e., an alert recorded by an IDS.

The following sections present how the contextual event causal dependency model can be computed based on logged events.

V. Model Implementation

In this section, we present how existing work permits the computation of the parts of our contextual event causal dependency model. The computation of causal dependencies among events has been especially studied in the operating system research field and distributed system field. All these methods and implementations allow the observation and recording of a subset of actions performed by active objects in the supervised system. Each implementation permits observations at a given level of the system and thus provides partial information of what really transpires in the system. For example, a system call is recorded at a time that is different from the library call occurrence time in the application. Accordingly, it is important to note that the existing work only enables an approximation of the correct model. However, the most advanced implementation of our model would involve merging all technologies that are described in the following subsections.

A. Computing an Approximation of Causal Dependencies among Actions using Information Flow Monitoring

Numerous existing works propose the observation and recording of information flows performed by kernel level objects. Usually, in all these works, an active object can be a process, a thread, or an execution unit as that in [29]; moreover, a passive object might be a file, socket, pipe, memory, or finer-grained data unit as that in [30]. An event is logged each time an information flow is observed between two object states; however, the object states are not known. Thus, all these information flow monitoring implementations permit the approximation of the contextual action causal dependency model.

1) Information Flow Monitoring inside Kernel Space: Although information flows can be deduced from system call monitoring [17], [31], considerable work has been performed to instrument the Linux kernel by leveraging the Linux Security Module framework and netfilter modules to enable a finer-grained observation of information flows produced by system call invocations [39], [42], [40]. In particular, RfBlare [18] proposes a solution to handle information flow monitoring evasions that leverage race conditions among system calls. In [7], the authors propose the insertion of new dedicated hooks to record information flows into the provenance model.

Some implementations [21], [42], [7], [40] also consider message-passing among the different nodes of the supervised system using the IP option field of packets to transmit additional information, such as the identifier of the sender. Hence, the packet-related part of these methods permits the capture of an approximation of our model using case (3) (cf. definition 4) at the OS level, i.e., inside the netfilter module of the kernel.

2) Application Behavior Analysis—Enabling Finer-Grained Causal Dependency Computation: Different from the kernel abstraction level, memory I/O operations can be observed at the application level. Application instrumentation methodologies at the source code level or binary level can be leveraged to enable the observation of more actions. An example to achieve this is the *libc* instrumented with hooks to enable the recording of more actions (e.g., the action of writing to a memory cell) [29], [24]. Consequently, this approach is able to compute more precisely the contextual action causal dependencies in the application. Moreover, program behavior analysis techniques can be leveraged to discover the notion of sessions as defined in our model. In [29], [34], binaries are instrumented to make them invoke specially crafted system calls that delimit execution units, which can be considered as sessions, of long-running processes. The insight is that these applications mainly rely on the external event loop processing paradigm, and each iteration of the loop is causally independent of the previous and subsequent iterations. A training phase is then conducted using dynamic analysis to determine inter-unit dependencies through memory accesses. The notion of data units has also been proposed in [30], [47] to enable finer-grained dependency observation pertaining to passive object states. However, data unit discovery requires the instrumentation of the application's source code.

Unfortunately, these works are constrained by the fact that execution and data units are extremely low-level to be practical and do not perform high-level tasks well. With the insight that these tasks are typically represented by data structures in the source code, the MPI [33] proposes a framework to annotate the source code and perform program analysis to identify execution partitioning, i.e., sessions.

As instrumentation is not always possible or desirable, and it is frequently difficult to maintain; other techniques [26], [32] infer program behavior by dynamically identifying implicit sessions and their related system calls.

B. Object State Snapshots

The existing work presented in the previous subsection only records events by observing active object actions. They do not provide a means to record object states at any given time and only focus on information flow monitoring (e.g., this is attributable to the known semantics of system calls). Hence, only the contextual event causal dependency part of our model can be computed. To completely compute our model, from contextual action causal dependency to contextual event causal dependency, the supervision system has to be able to capture objects states at any moment during the observed system execution. Such memory captures are performed in debugging and replaying tools, where an action can be stopped and its context captured, i.e., the value of the object state. Such memory capture mechanisms can be embedded in processors, virtual machine hypervisors, or emulators, such as QEMU, as that in [9]. In [12], the authors define a state-aware system as an *eidetic system*. As opposed to the whole system, their implementation is embedded in the Linux kernel and allows the recording of process states and their evolution in order to replay subsets of processes. These principles can also apply at the application level. In [45], the authors instrument the Chrome web browser to enable the logging and replay of the user action contexts, i.e., the document object model that contains web page objects and Javascript source code. By enabling the finer-grained logging of action contexts, these methodologies also allow the approximation of the contextual action causal dependency model.

C. Message Passing Systems

Every work in this distributed system research field implements various means to deduce causal dependencies among distributed process actions. One of the first approaches to obtain partial ordering of actions in distributed systems was introduced by [15]. Numerous articles have been published to propose various means of capturing temporal causal dependencies [35], [43]. Evidently, any of these technical solutions is relevant for computing contextual action dependencies of different processes.

D. Implementation Conclusion

We can thus leverage previous work to obtain an approximation of our model that could ultimately lead to the computation of causal dependencies among logged events. As far as we know, no system provides the sum of all notions required by our model. The purpose of the next section is to propose another type of model approximation without any instrumentation or OS modification, message passing technology, or application. The entire approximated model is deduced from the heterogeneous logs of the distributed system.

VI. Approximation of Contextual Action Causal Dependency Model—A Lightweight Approach

In this section, an implementation of our contextual event causal dependency model using only existing logging facilities to record events is proposed; thus, our implementation does not rely on any instrumentation of the supervised system. Herein, it is shown that the use of existing logs generated by various sensors, such as auditd, netfilter, application logging systems, and Bro NIDS, already yield a good approximation of the model.

We describe how each type of log is treated depending on its data source to deduce causality relationships among heterogeneous logged events. Section VI-B describes how contextual events issued from the analysis of application-related sources are deduced, i.e., from application logs and application level HIDS alerts. Section VI-C is related to the analysis of system call logs and HIDS alerts. Finally, Section VI-D is related to the events deduced from network traces or NIDS alerts.

A. From Contextual Events to Contextual Actions

In Section IV, three different notions are introduced: contextual actions, contextual events, and events. These objects are assembled in a top-down approach, from the top layer of contextual actions to the bottom layer of events. In order to produce such an approach, inspection mechanisms must be implemented (Section V). Actually, the lightweight implementation that is proposed in this section involves taking the perspective of an analyst who only has events and alerts. This approach does not permit the generation of real contextual actions from the system observation (e.g., there is no system implementation that enables the observation of object states and computation of causal dependencies among them). Thus, the approach we propose involves building layers from events to contextual actions in a bottom-up manner. As stated in Section IV, it is known that two contextual events are causally dependent if and only if two corresponding contextual actions are causally dependent as well. We virtually produce (1) approximations of contextual events from heterogeneous logged events and (2) an approximation of contextual actions from contextual events. For each contextual event, there exists a single contextual action (the event is produced by observing this action). A contextual action is surmised from a set of events that are the results of observations of several sensors of this single action. This contextual action is an approximation of the real action: (1) the time at which it occurs is supposed to be in the time interval of the timestamps of contextual events, and (2) the action is the set of events that is produced by the observation of the action of several sensors. To build this set, it is necessary to preprocess contextual events in order to aggregate them into a single set. Aggregation is a common part of a correlation process; it consists of merging several events or alerts into a single meta-alert [46].

B. Application Data Source

1) Contextual Events from Application Level Logged Events: A process, p, running an application produces a sequence of contextual application events, $\{(e_i^{App}, p, t_i)\}$, where the event e_i^{App} describes an action executed by the observed process, p, at time t_i. Note that the observed object has to be identified to enable the computation of our model. For instance, these information can be retrieved from the event semantics or from an external knowledge base. Depending on the deployed supervision system, these contextual events can be analyzed by an application level HIDS to detect suspicious program behavior, e.g., an abnormal sequence of events. The HIDS analysis is performed in the context of the supervised application; any alert raised can be defined as part of the application context. An application level alert is then recorded as an application contextual event $(e_{alert}^{App}, p, t_{alert})$ with $e_{alert}^{App} \in Obs\big((e_{alert}^{App}, (p, t_{alert}))\big)$; $(e_{alert}^{App}, (p, t_{alert}))$ is the approximation of the contextual action that triggered the alert.

Example 1: e_{Req}^{App} = [04/Nov/2018:21:50:54 +0000] 1566 10.0.2.15 80 10.0.2.2 56582 "POST /bWAPP/sqli_6.php HTTP/1.1" 200 6799 "http://10.0.2.15:80/bWAPP/sqli_6.php" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0"

Let the processing of application events in our attack scenario example be illustrated using the Apache application events. The application event, e_{Req}^{App}, illustrated with Example 1, corresponds to the Apache log entry that records the fact that the request has been treated. The second attribute, i.e., 1566, represents the *process identifier (PID)* of the process that treated the request. This information allows us to identify precisely the process that recorded the logged event. Thus, a contextual event can be computed by identifying the corresponding Apache active object using the PID information and place it on its timeline using the timestamp, $t_{e_{Req}^{App}}$. Note that this unique application event is actually a fusion of two events: the POST request and its result; however, we only have one timestamp for the two. Thus, an approximation that the two events happened at the same time (i.e., the timestamp) is necessary. Moreover, these two events could also be considered as the reception of a message: the POST request and sending of request result.

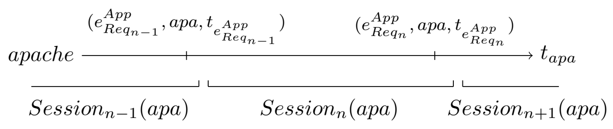

$(e_{Req_{n-1}}^{App}, apa, t_{e_{Req_{n-1}}^{App}})$
$(e_{Req_{n}}^{App}, apa, t_{e_{Req_{n}}^{App}})$
$apache$
t_{apa}
$Session_{n-1}(apa)$
$Session_{n}(apa)$
$Session_{n+1}(apa)$

Fig. 6. Apache application contextual event computed from access.log.

2) Sessions in Application Contextual Events: Figure 6 shows the placement of the contextual application event on the Apache object timeline. Considering that Apache is a web server with no memory, each request processing is independent of the others. In other words, each request would be executed in an independent session, as defined in Section IV-A3. Thus, a session can be deduced for each contextual event corresponding to a request. Note that the Apache application could be further instrumented for the accurate determination of sessions at the application level. With two logged events shown in the example, three sessions are illustrated. How to identify the sessions more precisely using system calls is illustrated in Subsection VI-C.

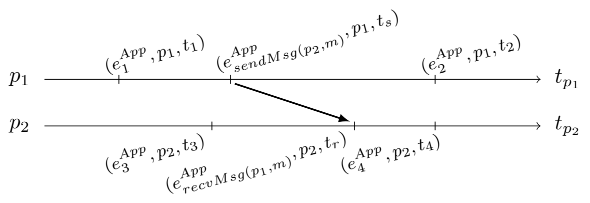


Fig. 7. Message exchange between two applications.

3) Message Exchanges in Application Contextual Events: In our attack scenario example, messages are exchanged among different processes at different nodes; for instance, the $apache$ and $mysqld$ processes exchange messages. With further instrumentation of the applications, these messages could be recorded in the application level logged events. Figure 7 illustrates the use of application level contextual events (when they are available) to compute causal dependencies between the two process active objects. Note that the clocks of the two processes, t_{p_1} and t_{p_2}, are not necessarily synchronized. Process p_1 records in its log a contextual event, $(e_{sendMsg(p_2,m)}^{App}, p_1, t_s)$, indicating that it sends a message m to p_2 at time t_s with the $sendMsg$ function. Process p_2 records $(e_{recvMsg(p_1,m)}^{App}, p_2, t_r)$, which indicates that it receives message m from p_1 at time t_r with the $recvMsg$ function. The foregoing implies that both contextual events are causally dependent: $(e_{sendMsg(p_2,m)}^{App}, p_1, t_s) \rightharpoonup (e_{recvMsg(p_1,m)}^{App}, p_2, t_r)$. Accordingly, we also have $e_{sendMsg(p_2,m)}^{App} \triangleright e_{recvMsg(p_1,m)}^{App}$. If it is supposed that all contextual events of an object are part of the same session, then we can deduce the following as partial ordering of events using the *event causality* relationship: $e_1^{App} \triangleright e_{sendMsg(p_2,m)}^{App} \triangleright e_2^{App}$; $e_3^{App} \triangleright e_{recvMsg(p_1,m)}^{App} \triangleright e_4^{App}$; $e_1^{App} \triangleright e_{sendMsg(p_2,m)}^{App} \triangleright e_{recvMsg(p_1,m)}^{App} \triangleright e_4^{App}$. In practice, if the processes are communicating on a single node, then we would be able to build this causal dependency using the OS level system call traces. However, if the two processes are communicating through the network via message exchanges, causality relationships can be deduced from their application level logs.

C. System Call Data Source

1) Contextual Events from System Call Level Logged Events: System call invocations can be recorded inside the kernel or by a dedicated module in the kernel space that uses hooks to intercept system calls and produce a trace for each process. To avoid the instrumentation of the kernel, an already existing tool, auditd, is utilized to record system call events. In this trace, contextual events are recorded in the form (e^{Sys}, p, t); this means that each recorded event indicates which system call is invoked by process p at a given timestamp. The event e^{Sys} is a system call that is executed in the context of process p at time t. Note that this time is defined as the timestamp of the contextual event, which is an approximation, as previously detailed. In that sense, the event e^{Sys} is not produced by process p but is part of the set of contextual events of p, i.e., the system call is executed in the context of the process state.

Example 2: $e_{accept4}^{Sys}$ =

```
type=SYSCALL msg=audit(1541366508.539:47875): arch=c000003e syscall=288
success=yes exit=10 a0=3 a1=7ffce59a1100 a2=7ffce59a10e0 a3=80000 items=0
ppid=1106 pid=1566 auid=4294967295 uid=33 gid=33 euid=33 suid=33
fsuid=33 egid=33 sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="apache2"
exe="/usr/sbin/apache2" key=(null)
type=SOCKADDR msg=audit(1541366508.539:47875): saddr=0200DD060A0002
020000000000000000 (saddr= (AF_INET) 10.0.2.2 : 56582)
type=PROCTITLE msg=audit(1541366508.539:47875): proctitle=2F7573722F7362
696E2F61706163686532002D6B007374617274 (proctitle=/usr/sbin/apache2 -k start)
```

Among the invoked system calls, some produce information flows (e.g., read(), write(), send(), and recv()) and others do not (e.g., wait(), mprotect(), and futex()). Each time an information flow is produced, two objects are involved: an active object (e.g., a process or the network) and a passive object (e.g., shared memory, sockets, files, and pipes). Recall that only active objects can produce events; accordingly, the information flow implies a causal dependency between two contextual events: (1) if the information flows from process p to the passive object, o, (e.g., system calls from the write() family), then $(e^{Sys}, p, t) \rightharpoonup (\emptyset, o, t)$; (2) if the information flows from the passive object, o, to process p (e.g., system calls from the read() family), then $(\emptyset, o, t) \rightharpoonup (e^{Sys}, p, t)$; (3) if the information flows from a parent process, p_{parent}, to a child process, p_{child} (e.g., system calls from the fork() family), then $(e^{Sys}, p_{parent}, t) \rightharpoonup (\emptyset, p_{child}, t)$.

Processes communicate using the interprocess communication (IPC) mechanism, which always involves passive objects, such as pipes, sockets, message queues, or shared memory. The access to these passive objects may or may not involve system calls (e.g., shared memory and memory mapped files). In the latter case, communications cannot be intercepted by the kernel because they are produced at the hardware level. Consequently, the log files available in the system do not exhibit all information flows. Thus, we can only compute a part of the causal dependencies among contextual events. Example 2 illustrates an accept() system call that produces two contextual events: one for the Apache process and another for the created socket.

Following the illustration of application event processing in our attack scenario example, we now illustrate system call event processing. As can be observed in Fig. 8, system call contextual events are not sufficient to link the Apache and MySQL hosts. For the purpose of making them readable, the

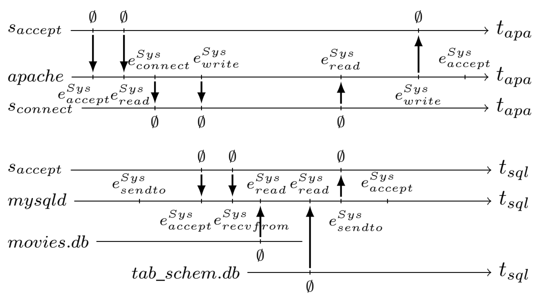


Fig. 8. System call contextual events computed from audit logs.

contextual events are not written in figures, but are replaced by their corresponding logged events. In Fig. 8, it can be seen that system call events already allow the linking of several objects that are involved in the attack scenario. A system call event contains the required information to causally link active objects, i.e., a process identified by its PID and a passive object. Because system call events are produced on a single node, they can be temporally ordered and easily placed on the timelines of their related objects. Note that the Apache and MySQL host clocks might not be synchronized. Thus, the system call contextual events from the two nodes cannot be totally ordered; moreover, at this point, they cannot be partially ordered.

Similar to the application level HIDS, a system call level HIDS analysis is also performed in the context of a process, and any alert raised is also considered to be part of the application context. Accordingly, a system call level alert is recorded as a contextual event, $(e^{Sys}_{alert}, p, t_{alert})$, where the alert e^{Sys}_{alert} is raised in the context of p at time t_{alert} with $e^{Sys}_{alert} \in Obs\big((a,(p,t))\big)$ and $(a,(p,t))$ being the contextual action that triggered the alert. Because auditd can be configured to observe directories or files of interest using dedicated rules, it can also be used as a HIDS. Such an alert contains the same information as a classic system call event and can also be easily placed on the timeline of the related object.

2) Sessions in System Call Contextual Events: As mentioned in the application data source subsection, system calls can be used to determine sessions. In the case of Apache, it is known that request handling delimits sessions. At the kernel level, a request is read from the network that used the accept() system call. Thus, it is considered that accept() system calls start new sessions for the *apache* process.

D. Network Data Source

1) Contextual Events from Network Level Logged Events: Network events[3], specifically packet flows, can be recorded by network sniffing tools. Even though communications over the network are produced by processes, network traces do not have any information concerning the processes and cannot be directly related to their contexts. Therefore, in this trace, contextual events are recorded in the form $(e^{Net}_{conn}, netw, t)$, which means that a network event, e^{Net}_{conn}, i.e., a raw or analyzed packet flow, related to the connection, $conn$, is observed in the context of the network interface, $netw$, at time t. In that sense, $netw$ is modeled as an active object that can observe, analyze, and record packet flows. Moreover, a network active object, $netw$, is considered as stateless; that is, considering the context of $netw$, a contextual event is independent of other contextual events. Note that a given network interface belongs to a unique host, and a host can have multiple network interfaces. A network event, e^{Net}_{conn}, observed on a given network interface, $netw$, always involves communication between two objects, i.e., a network socket that uses $netw$ to read from or write to the network. The network interface of the source or destination of the message is (1) $(\emptyset, socket^{src}_{conn}, t) \rightharpoonup (e^{Net}_{conn}, netw, t)$ if the information flows from the passive object (the socket) to the active object (the network interface, $netw$), or (2) $(e^{Net}_{conn}, netw, t) \rightharpoonup (\emptyset, socket^{dst}_{conn}, t)$ if the information flows from the active object (the network interface, $netw$) to the passive object (the socket).

Packet flows, as previous data sources, can also be analyzed by a NIDS, such as Bro NIDS, which generates event logs from the protocol dissection. A NIDS alert is modeled as any other network event, i.e., a contextual event in the form of $(e^{Net}_{conn,alert}, netw, t_{alert})$ with $e^{Net}_{conn,alert} \in Obs\big((a,(netw,t))\big)$; $(a,(netw,t))$ being the contextual action that triggered the alert. Because Bro can generate several events from the same connection, it well illustrates the fact that an action can be observed as several logged events, i.e., the set $Obs\big((a,(netw,t))\big)$ of its related contextual action, $(a,(netw,t))$, can contain several contextual events: $Obs\big((a,(netw,t))\big) = \big\{(e^{Net}_{i\,conn}, netw, t_{e_i})\big\}$. Moreover, these logged events can even be aggregated using event fusion techniques. Note that a NIDS typically observes network activities using a network tap; thus, it relies on its own network interface to observe network activities.

Example 3: $e^{Net}_{conn,alert}$ = 2018-11-04T21:50:55.001600Z CgGkAp4P6ThQzD2Wg 192.168.1.2 48218 192.168.1.3 3306 tcp MySQL::Sqli SELECT * FROM movies WHERE title LIKE '%%' UNION ALL SELECT table_schema,table_name, null, null, null, null, null from information_schema.tables;– %' SQLi Attempt : Suspect syntax detected. ['Notice::ACTION_LOG']

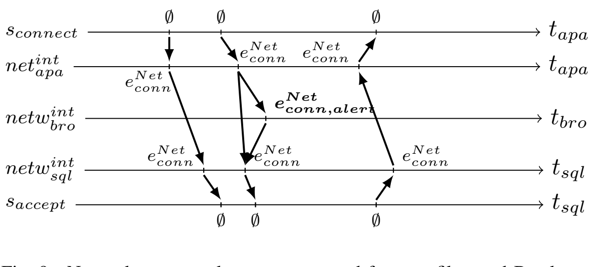


Fig. 9. Network contextual events computed from netfilter and Bro logs.

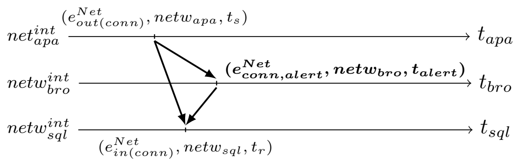
Figure 9 illustrates the use of events deduced from network

[3]We refer to log entries deduced from network traces as network events.

traces in our motivating example. Only the internal network side, i.e., $netw_{apa}^{int}$, $netw_{sql}^{int}$, and $netw_{bro}^{int}$ network interface active objects, has been shown in this example. The event, $e_{conn,alert}^{Net}$, illustrated by Example 3, represents a Bro alert triggered by an SQL injection detection rule. It has been deduced from the network packet capture by dissecting the MySQL application protocol level; it represents the MySQL query crafted by the attacker and requested by the Apache web server.

2) Network Socket Object Identification: Because sockets are mechanisms of the IPC, they are involved in the system call layer of our model. Thus, sockets are the means to bridge contextual network events and contextual kernel events. Concerning Linux and its socket handling, only the pair $\{remote_{ip}, remote_p\}$ is available in the system calls of the network family (i.e., socket(), connect(), and accept()). In our model, a socket is identified by the quadruplet $\{host_{ip}, host_p, remote_{ip}, remote_p\}$. Thus, system calls alone are not sufficient to identify a socket object in our model. To completely identify the socket objects, we leverage netfilter, the embedded Linux firewall, to record any established connection. Netfilter events allow us to easily link a socket to its corresponding connection by matching $\{remote_{ip}, remote_p\}$ with the connection information: (1) for incoming connections, i.e., sockets created by the accept() system call family, $\{remote_{ip}, remote_p\} = \{src_{ip}, src_p\}$; (2) for outgoing connections, i.e., sockets created by the connect() system call family, $\{remote_{ip}, remote_p\} = \{dst_{ip}, dst_p\}$. This matching allows the acquisition of all information necessary to describe a network socket. Note that this matching does not rely on precise timestamp matching, i.e., it is sufficient that the timestamps are close.

3) Message Exchanges in Network Contextual Events: Network contextual events can typically be considered as message exchange events among several network interfaces. By leveraging network socket object information and connection information, i.e., $\{src_{ip}, src_p, dst_{ip}, dst_p\}$, the nature of the message can easily be identified: whether it corresponds to a sending or a reception.



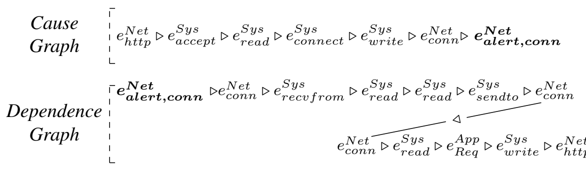
$(e_{out(conn)}^{Net}, netw_{apa}, t_s)$
net_{apa}^{int}
t_{apa}
$(\boldsymbol{e_{conn,alert}^{Net}, netw_{bro}, t_{alert}})$
$netw_{bro}^{int}$
t_{bro}
$netw_{sql}^{int}$
t_{sql}
$(e_{in(conn)}^{Net}, netw_{sql}, t_r)$

Fig. 10. Message exchange among network objects.

The example in Fig. 10 illustrates the use of network level contextual events to compute causal dependencies utilizing the message exchange part of definition 4. Note that the clocks of the three objects (t_{apa}, t_{bro}, and t_{sql}) are not necessarily synchronized because they potentially belong to three different hosts. The network active object, $netw_{apa}$, records in its log a contextual event, $(e_{out(conn)}^{Net}, netw_{apa}, t_s)$, indicating that it sends data (i.e., the SQL query via the connection $conn$ at time t_s). The network active object, $netw_{sql}$, records $(e_{in(conn)}^{Net}, netw_{sql}, t_r)$; this indicates that it receives data from $conn$ at time t_r. The above implies that both contextual events are causally dependent: $(e_{out(conn)}^{Net}, netw_{apa}, t_s) \rightharpoonup (e_{in(conn)}^{Net}, netw_{sql}, t_r)$. Because the Bro NIDS observes and analyzes the network using a network tap, it detects a suspicious behavior and raises an alert. This alert corresponds to the same connection, $conn$; hence, we also have $(e_{out(conn)}^{Net}, netw_{apa}, t_s) \rightharpoonup (e_{in(conn)}^{Net}, netw_{sql}, t_r)$ and $(e_{out(conn)}^{Net}, netw_{apa}, t_s) \rightharpoonup (e_{in(conn)}^{Net}, netw_{sql}, t_r)$.

E. Cause Graph Illustration

Having described the processing of each type of logged events, the heterogeneous contextual event causal dependency model can be computed as shown in Fig. 2. Moreover, with all contextual events set on their related object timelines, we can build the cause and dependency graph for alert $(e_{conn,alert}^{Net}, netw_{bro}, t_{alert})$. As defined in Section IV-C, these graphs contain all contextual events that contribute to or depend on the contextual event $(e_{conn,alert}^{Net}, netw_{bro}, t_{alert})$. Starting from the alert, these graphs are computed using the backward and forward traversal of timelines according to the "$\rightharpoonup$" relationship. Thus, objects and contextual events that do not contribute to the alert according to the "$\rightharpoonup$" relationship are not contained in the graphs.



Cause Graph
$e_{http}^{Net} \triangleright e_{accept}^{Sys} \triangleright e_{read}^{Sys} \triangleright e_{connect}^{Sys} \triangleright e_{write}^{Sys} \triangleright e_{conn}^{Net} \triangleright \boldsymbol{e_{alert,conn}^{Net}}$
Dependence Graph
$\boldsymbol{e_{alert,conn}^{Net}} \triangleright e_{conn}^{Net} \triangleright e_{recvfrom}^{Sys} \triangleright e_{read}^{Sys} \triangleright e_{read}^{Sys} \triangleright e_{sendto}^{Sys} \triangleright e_{conn}^{Net}$
$\triangleleft$
$e_{conn}^{Net} \triangleright e_{read}^{Sys} \triangleright e_{Req}^{App} \triangleright e_{write}^{Sys} \triangleright e_{http}^{Net}$

Fig. 11. Cause and dependence graphs of NIDS alert.

Event causal dependencies can finally be deduced using Definition 8. The result involves the event cause and dependence graphs illustrated in Fig. 11. The administrator would have all the events that causally contribute to or causally depend on the raised NIDS alert according to the causal dependency relationship that we defined. The SQL injection attack scenario on the web and database servers makes it possible to completely illustrate our causal dependency computation methodology among heterogeneous events using an intermediary model (the contextual event causal dependency model).

VII. Approach Assessment

The purpose of this section is to demonstrate that the model described above is useful to detect real attack cases. In particular, the model makes it possible to explain an attack and allows the analyst to easily understand an attacker's activity. This section is organized as follows. Subsection VII-A discusses the services and supervision sensors that are deployed; Subsection VII-B describes the attacks that are performed; Subsection VII-D depicts the model obtained from the event logs and shows the graphs deduced from these logs.

A. Test Environment Description

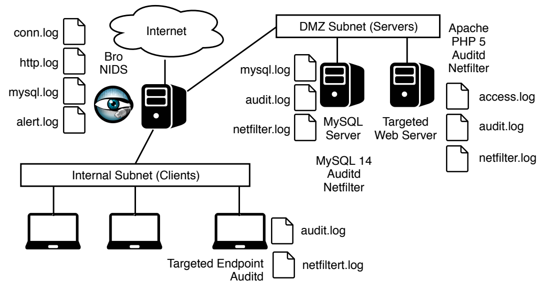


Fig. 12. Network architecture of test environment.

In initial tests, the target machines are three single virtual machines in an OpenStack environment. As shown in Fig. 12, the network architecture is composed of a demilitarized zone (DMZ) and an internal network. Inside the internal network, a end-user machine running on the Ubuntu desktop environment with a Firefox browser is considered. The DMZ is composed of two machines: an Apache server and MySQL server. The Apache server runs a Linux OS 17.10, PHP 5.6 execution environment, and Bash 4.2; and is vulnerable to a ShellShock attack. The MySQL server runs MySQL version 14.14 with the same OS as the Apache server. For supervision, the administrator (1) deploys an auditd daemon on the web server host and MySQL host and configures netfilter to log connections; (2) deploys an auditd daemon on the end-user machine and configures netfilter to log connections; (3) deploys a network supervision machine that runs Bro NIDS. In practice, we can have several virtual machines in the two subnetworks. For the demonstration, the tests are limited to one supervision and three functional machines. Auditd is configured in a particular manner so that (a) it allows monitoring of system calls that produce information flows and (b) produces alerts on specifically known attack signatures. For instance, if a known system executable is launched, an alert is produced.

B. Attack Scenario Description

These experiments are performed using three different attacks. Two of these target the Apache server and one attacks the end-user machine. These three can be described as follows.

– Attack 1: The first attack against the Apache web server primarily involves the exploitation of the ShellShock bash vulnerability (also referenced as CVE-2014-6271). It consists of the execution of bash environment variables when they contain some bash codes. Apache executes a bash script via the cgi-bin interface. The attack includes forging a request that will be executed by bash and contains environment variables with the injected code;
– Attack 2: The second attack again targets the Apache web server; it mainly exploits the command injection vulnerability in a PHP script. The attacker gathers information pertaining to the system and copies the list of users from */etc/passwd*.
– Attack 3: This last attack is performed against the Ubuntu end-user machine. The attacker is assumed to have previously succeeded in obtaining the target's sudo credentials. The victim user downloads a Debian package containing a malicious payload from the Internet. This payload contains a crafted Trojan; once launched, this malware installs a reverse transmission control protocol (TCP) backdoor in interaction with the attacker's machine via the Metasploit framework. The attacker then leverages this connection and previously obtained credentials to further compromise the machine and obtain sensitive data that are sent to a server controlled by the attacker using shell scripts.

C. Generated Logs

1) Log Description: The logs generated on the supervised system, as described in Fig. 12, are as follows. (1) On the network supervision machine, we have Bro Logs (*http.log*, *dns.log*, *conn.log*, and *mysql.log*). (2) On the Apache server machine, we obtain application level logs (*access.log*) generated by the Apache application. Moreover, we have the auditd log (*audit.log*) and netfilter log (*nfconn.log*). (3) On the MySQL server machine and end-user machine, we only generated the auditd log (*audit.log*) and the netfilter log (*nfconn.log*). (4) On the end-user machine, only auditd log (*audit.log*) is generated. Thus, four types of logs (effectively treated as heterogeneous logs) are obtained: network logs (Bro), application level logs (Apache), system call logs (auditd), and netfilter logs.

TABLE I
NUMBER OF EVENTS IN LOGS

	audit.log	access.log	nfconn.log	Bro logs
Attack 1	19 877	5	85	131
Attack 2	8888	35	24	78
Attack 3	82 679	0	345	347

2) Logs Generated During Attacks: The three attacks are sequentially performed; they are run on a fresh configuration and generate the logs that are summarized in Table I. This experimentation does not exhibit the scalability of the approach because only the validity of the model on a small set of logged events is demonstrated. The current implementation seems to scale to a large set of events; however, the performance assessment remains as a work in progress because it relies on several technological choices (e.g., type of databases, scaling of log collectors, and optimization of requests for data).

D. Results

In order to apply our approach, we have to populate the graph database with contextual events and their links, which are deduced from the log files; these contextual events are actually nodes in the graph database. As described in our model in Definition 7, a contextual event node can have several incoming and outgoing edges. These edges indicate the $\rightharpoonup$ causal dependency relationship derived from sessions, information flows, and message exchanges.

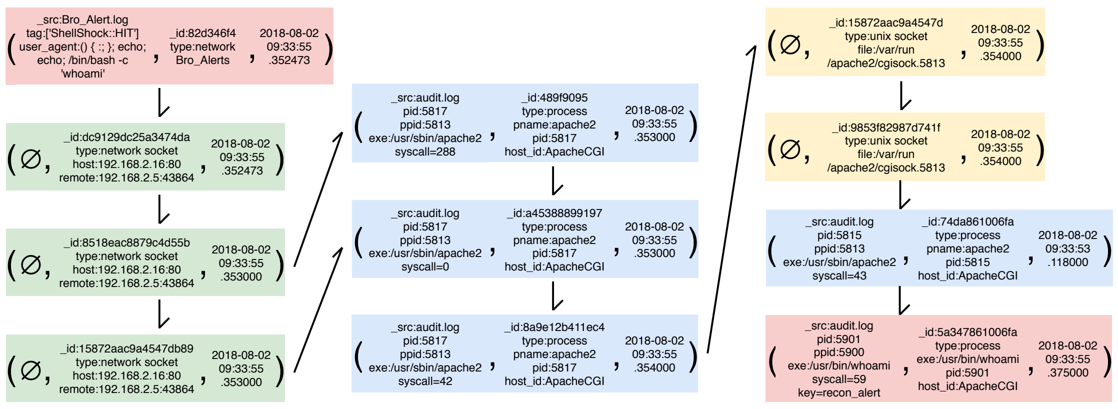

_src:Bro_Alert.log tag:['ShellShock::HIT'] user_agent:() { :; }; echo; echo; /bin/bash -c 'whoami'
_id:82d346f4 type:network Bro_Alerts
2018-08-02 09:33:55 .352473
∅
_id:dc9129dc25a3474da type:network socket host:192.168.2.16:80 remote:192.168.2.5:43864
2018-08-02 09:33:55 .352473
∅
_id:8518eac8879c4d55b type:network socket host:192.168.2.16:80 remote:192.168.2.5:43864
2018-08-02 09:33:55 .353000
∅
_id:15872aac9a4547db89 type:network socket host:192.168.2.16:80 remote:192.168.2.5:43864
2018-08-02 09:33:55 .353000
_src:audit.log pid:5817 ppid:5813 exe:/usr/sbin/apache2 syscall=288
_id:489f9095 type:process pname:apache2 pid:5817 host_id:ApacheCGI
2018-08-02 09:33:55 .353000
_src:audit.log pid:5817 ppid:5813 exe:/usr/sbin/apache2 syscall=0
_id:a45388899197 type:process pname:apache2 pid:5817 host_id:ApacheCGI
2018-08-02 09:33:55 .353000
_src:audit.log pid:5817 ppid:5813 exe:/usr/sbin/apache2 syscall=42
_id:8a9e12b411ec4 type:process pname:apache2 pid:5817 host_id:ApacheCGI
2018-08-02 09:33:55 .354000
∅
_id:15872aac9a4547d type:unix socket file:/var/run /apache2/cgisock.5813
2018-08-02 09:33:55 .354000
∅
_id:9853f82987d741f type:unix socket file:/var/run /apache2/cgisock.5813
2018-08-02 09:33:55 .354000
_src:audit.log pid:5815 ppid:5813 exe:/usr/sbin/apache2 syscall=43
_id:74da861006fa type:process pname:apache2 pid:5815 host_id:ApacheCGI
2018-08-02 09:33:53 .118000
_src:audit.log pid:5901 ppid:5900 exe:/usr/bin/whoami syscall=59 key=recon_alert
_id:5a347861006fa type:process exe:/usr/bin/whoami pid:5901 host_id:ApacheCGI
2018-08-02 09:33:55 .375000

Fig. 13. Contextual event cause graph of `whoami` alert.

TABLE II
NUMBER OF CONTEXTUAL EVENTS, CAUSAL DEPENDENCY LINKS, AND OBJECTS

	Contextual Events	Links	Objects
Attack 1	40 154	70 898	9316
Attack 2	5523	5432	165
Attack 3	166 375	329 947	2645

1) Contextual Event and Causal Dependency Generation: Based on the previously described logs, the model is applied to generate contextual events and their dependencies. Table II lists the number of contextual events and links produced among these nodes in the graph. The graph database contains a reasonable number of nodes and links. For instance, in *Attack 1*, approximately 40 000 contextual events and 50 000 links are generated among these nodes. The analysis of logs permits the presentation of approximately 10 000 objects in the distributed system (network, processes, files, sockets, pipes, and so on).

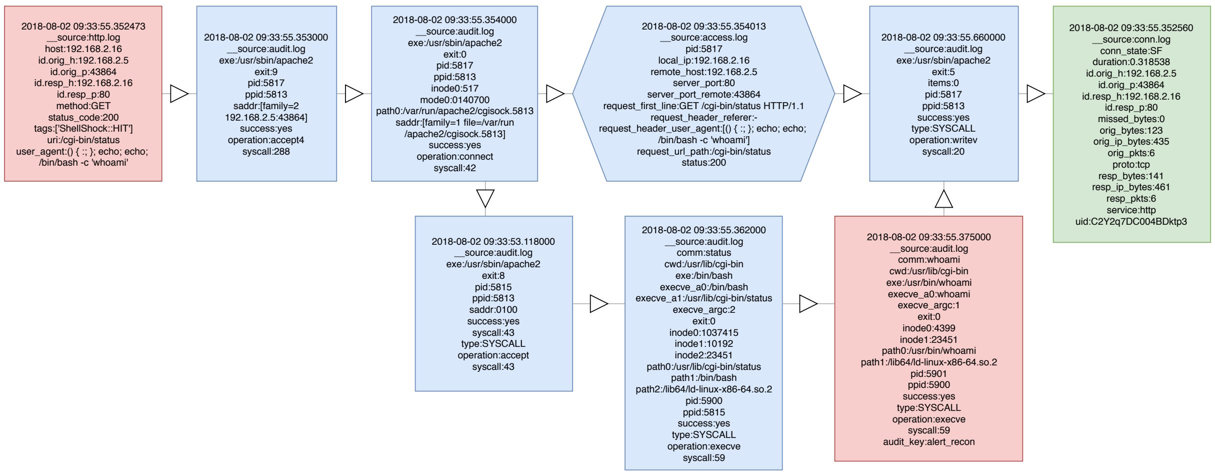
2) Dependency Graph Generation: It may be recalled that if a contextual event that is considered an IoC is given, then the purpose of our approach is to generate the graph of contextual events that are the cause or consequence of this malicious event. Thus, initially, it is necessary for the analyst to identify a malicious contextual event. To illustrate the approach and its subsequent results, this section focuses on the first attack. In practice, the approach proves to be adequate in the context of the three attacks. In *Attack 1*, the first step of the attacker is to determine his privilege. To obtain this information, he executes the `whoami` binary. The analyst recognizes that this execution is probably malicious because of an alert logged in *audit.log* file. Accordingly, he has to analyze the root cause of this event. The cause graph that mainly involves obtaining the graph of contextual events on which this event is causally dependent is first generated; this is shown in Fig. 13. The color representations are as follows: light green boxes are contextual events of socket objects; blue boxes are contextual events of active objects that emanate from system call events; yellow boxes are contextual events of passive objects that originate from system call events, e.g., files, pipes, or unix sockets; red boxes correspond to alerts related to contextual events.

The foregoing indicates that the attacker's action does not depend on anything other than the HTTP request, as shown in Fig. 14. The analyst then investigates the dependence graph of the `whoami` HTTP request. This request is evidently found to lead to the execution of the `whoami` command, whose result would return to the attacker. The analyst can finally comprehend the entire scenario of the first step of the attack. It involves several events issued from heterogeneous logs: Bro log files, auditd log file, and Apache application log file. Accordingly, this provides a basis to write a correlation rule (if necessary) by linking the several events produced by the supervision system during the attack.

VIII. DISCUSSION

Section VII shows several examples in which the proposed model makes it possible to build a dependency graph based on the events related to a given IoC. Certain points pertaining to the proposed approach have to be clarified.

– The analyst begins with a given event. If this event is not a part of a real attack, then the dependence and cause graphs will not contain traces (events) of an attack. Accordingly, the analyst should begin with an abnormal event that emanates from a base of threat intelligence or an alert produced by an IDS. In this case, evidently, the probability that the graph contains all events related to an attack is higher.
– The initial impression on the proposed approach is that it seems to be closely related to forensic examination. This is evidently the case because it provides a formal framework for event investigation. However, in our work, the approach is aimed at being implemented in a SIEM to aid an analyst explain why an IoC exists or to observe an ongoing attack in real time.
– If we include the approach in a SIEM, then we must be able to compute event causal dependencies in real time. This implies that the computation must be saved to an adapted database. The type of database selected



```
2018-08-02 09:33:55.352473
__source:http.log
host:192.168.2.16
id.orig_h:192.168.2.5
id.orig_p:43864
id.resp_h:192.168.2.16
id.resp_p:80
method:GET
status_code:200
tags:['ShellShock::HIT']
uri:/cgi-bin/status
user_agent:() { :; }; echo; echo;
/bin/bash -c 'whoami'
```

```
2018-08-02 09:33:55.353000
__source:audit.log
exe:/usr/sbin/apache2
exit:9
pid:5817
ppid:5813
saddr:[family=2
192.168.2.5:43864]
success:yes
operation:accept4
syscall:288
```

```
2018-08-02 09:33:55.354000
__source:audit.log
exe:/usr/sbin/apache2
exit:0
pid:5817
ppid:5813
inode0:517
mode0:0140700
path0:/var/run/apache2/cgisock.5813
saddr:[family=1 file=/var/run
/apache2/cgisock.5813]
success:yes
operation:connect
syscall:42
```

```
2018-08-02 09:33:55.354013
__source:access.log
pid:5817
local_ip:192.168.2.16
remote_host:192.168.2.5
server_port:80
server_port_remote:43864
request_first_line:GET /cgi-bin/status HTTP/1.1
request_header_referer:-
request_header_user_agent:[() { :; }; echo; echo;
/bin/bash -c 'whoami']
request_url_path:/cgi-bin/status
status:200
```

```
2018-08-02 09:33:55.660000
__source:audit.log
exe:/usr/sbin/apache2
exit:5
items:0
pid:5817
ppid:5813
success:yes
type:SYSCALL
operation:writev
syscall:20
```

```
2018-08-02 09:33:55.352560
__source:conn.log
conn_state:SF
duration:0.318538
id.orig_h:192.168.2.5
id.orig_p:43864
id.resp_h:192.168.2.16
id.resp_p:80
missed_bytes:0
orig_bytes:123
orig_ip_bytes:435
orig_pkts:6
proto:tcp
resp_bytes:141
resp_ip_bytes:461
resp_pkts:6
service:http
uid:C2Y2q7DC004BDktp3
```

```
2018-08-02 09:33:53.118000
__source:audit.log
exe:/usr/sbin/apache2
exit:8
pid:5815
ppid:5813
saddr:0100
success:yes
syscall:43
type:SYSCALL
operation:accept
syscall:43
```

```
2018-08-02 09:33:55.362000
__source:audit.log
comm:status
cwd:/usr/lib/cgi-bin
exe:/bin/bash
execve_a0:/bin/bash
execve_a1:/usr/lib/cgi-bin/status
execve_argc:2
exit:0
inode0:1037415
inode1:10192
inode2:23451
path0:/usr/lib/cgi-bin/status
path1:/bin/bash
path2:/lib64/ld-linux-x86-64.so.2
pid:5900
ppid:5815
success:yes
type:SYSCALL
operation:execve
syscall:59
```

```
2018-08-02 09:33:55.375000
__source:audit.log
comm:whoami
cwd:/usr/lib/cgi-bin
exe:/usr/bin/whoami
execve_a0:whoami
execve_argc:1
exit:0
inode0:4399
inode1:23451
path0:/usr/bin/whoami
path1:/lib64/ld-linux-x86-64.so.2
pid:5901
ppid:5900
success:yes
type:SYSCALL
operation:execve
syscall:59
audit_key:alert_recon
```

Fig. 14. Attack 1: first step event dependency graph.

is a graph database because it is efficient in recovering graph structures (which is the case here) and is expected to scale to a large number of events.

Another problem for the analyst is explaining the attack in detail, i.e., not only with respect to what are observed, but also in terms of the attacker's actions. It could be reasonable to explain how to deduce an attack description (e.g., an attack tree and attack graph) from an observed event graph. However, to the best of our knowledge, in practice, it is practically impossible to guess the attacker actions based on the observed events. As a result of the work of [19], it is known that attack and observation descriptions can have identical representations (here an attack tree and a correlation tree, respectively). However, even if we have the knowledge of a known vulnerability, we cannot easily and automatically deduce how this vulnerability is to be utilized. Even if the observed events can be guessed based on the attacker actions, the reverse cannot be performed automatically. Accordingly, our work intends to recover the consequences of the attacker's actions on the observed system (i.e., the causally dependent events) rather than the attacker's actions themselves.

IX. Conclusion

The purpose of this study is to propose a unified understanding of the causality relationships that can be defined between active entities, passive entities, and event logs. More precisely, we aim to define the notion of causal dependency between logged events and alerts that are produced by distributed processes and are not causally linked according to the relationship defined by Lamport. Inspired by Lamport's happened-before relationship in the distributed system research field and d'Ausbourg's causal dependency relationship among object states in the OS research field, we start by defining the concept of contextual actions and causal dependency, which links them. Thereafter, the relationships between contextual actions and logged events are gradually introduced to finally define the notion of causal dependencies among logged events. To the best of our knowledge, no system provides the sum of all notions required to compute the contextual action causal dependency model; only parts of these notions are actually available in a single implementation. The particularity of our implementation is that it permits the use of existing logging facilities of the supervised system to approximate the contextual action causal dependency model. The attack examples demonstrate that we can retrieve heterogeneous events that contribute to or depend on an event of interest, i.e., an alert or an IoC.

Acknowledgements

We would like to thank Pierre Wilke, Adam Faci and the anonymous reviewers for their precious time, valuable suggestions and comments. This work was supported by Thales Six GTS France. This work is part of the VESTA project sponsored by the *Direction Générale des Entreprises (DGE)* under agreement No. P141560-3210888.

References

[1] Bro network security monitor. [Online]. Available: https://www.bro.org
[2] Cve. [Online]. Available: https://www.cvedetails.com
[3] Etw. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/performance/etw-events-in-the-common-language-runtime
[4] Linux audit framework. [Online]. Available: https://people.redhat.com/sgrubb/audit/
[5] Netfilter. [Online]. Available: https://www.netfilter.org/
[6] Ossec. [Online]. Available: https://www.ossec.net
[7] A. M. Bates, D. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel." in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2015.
[8] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining temporal invariants from partially ordered logs," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 3, 2012.
[9] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2008.
[10] F. Cuppens and R. Ortalo, "Lambda: A language to model a database for detection of attacks," in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer, 2000.

[11] B. d'Ausbourg, "Implementing secure dependencies over a network by designing a distributed security subsystem," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 1994.

[12] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 14. USENIX Association, 2014.

[13] S. T. Eckmann, G. Vigna, and R. A. Kemmerer, "Statl: An attack language for state-based intrusion detection," *Journal of Computer Security*, vol. 10, no. 1-2, Jul. 2002.

[14] European Commission. (2010) Standard on logging and monitoring. [Online]. Available: https://www.eba.europa.eu/documents/101 80/1449046/Annex+5+Standard+on+Logging+and+Monitoring.pdf

[15] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in *Proceedings of the Australian Computer Science Conference*, University of Queensland, Australia, 1988.

[16] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1996.

[17] A. Gehani and D. Tariq, "Spade: support for provenance auditing in distributed environments," in *Proceedings of the International Middleware Conference*. Springer Berlin Heidelberg, 2012.

[18] L. Georget, M. Jaume, G. Piolle, F. Tronel, and V. V. T. Tong, "Information flow tracking for linux handling concurrent system calls and shared memory," in *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM)*. Springer, 2017.

[19] E. Godefroy, E. Totel, M. Hurfin, and F. Majorczyk, "Generation and assessment of correlation rules to detect complex attack scenarios," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2015.

[20] J. Goubault-Larrecq and J. Olivain, "A smell of orchids," in *International Workshop on Runtime Verification*. Springer, 2008.

[21] C. Hauser, F. Tronel, C. Fidge, and L. Mé, "Intrusion detection in distributed systems, an approach based on taint marking," in *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE, 2013.

[22] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, "Sleuth: real-time attack scenario reconstruction from cots audit data," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2017.

[23] S. Jajodia and S. Noel, "Advanced cyber attack modeling analysis and visualization," 2010.

[24] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand interprocess information flow tracking," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

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

[26] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie *et al.*, "Mci: Modeling-based causality inference in audit logging for attack investigation," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[27] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, Jul. 1978.

[28] D. Lanoë, M. Hurfin, and E. Totel, "A scalable and efficient correlation engine to detect multi-step attacks in distributed systems," in *Proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2018.

[29] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 2013.

[30] ——, "Loggc: garbage collecting audit log," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*. ACM, 2013.

[31] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[32] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[33] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "Mpi: Multiple perspective attack investigation with semantics aware execution partitioning," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2017.

[34] S. Ma, X. Zhang, and D. Xu, "ProTracer: towards practical provenance tracing by alternating between logging and tainting," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 2016.

[35] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Citeseer, 1988.

[36] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers *et al.*, "The open provenance model core specification (v1. 1)," *Future Generation Computer Systems*, vol. 27, no. 6, 2011.

[37] B. Morin, L. Mé, H. Debar, and M. Ducassé, "A logic-based model to support alert correlation in intrusion detection," *Information Fusion*, vol. 10, no. 4, 2009.

[38] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. L. McLean, D. W. Margo, M. I. Seltzer, and R. Smogor, "Layering in provenance systems." in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2009.

[39] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2006.

[40] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the Symposium on Cloud Computing (SoCC)*. ACM, 2017.

[41] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, "Hercule: Attack story reconstruction via community discovery on correlated log graph," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016.

[42] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: collecting high-fidelity whole-system provenance," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2012.

[43] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed Computing*, vol. 7, no. 3, Mar. 1994.

[44] E. Totel, B. Vivinis, and L. Mé, "A language driven intrusion detection system for event and alert correlation," in *Security and Protection in Information Processing Systems*. Springer, 2004.

[45] P. Vadrevu, J. Liu, B. Li, B. Rahbarinia, K. H. Lee, and R. Perdisci, "Enabling Reconstruction of Attacks on Users via Efficient Browsing Snapshots," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 2017.

[46] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer, "A Comprehensive Approach to Intrusion Detection Alert Correlation," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 1, no. 3, 2004.

[47] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.