# Heterogeneous Event Causal Dependency Definition for the Detection and Explanation of Multi-Step Attacks

Charles Xosanavongsa

## ▶ To cite this version:

## HAL Id: tel-02947368
## https://theses.hal.science/tel-02947368

Submitted on 24 Sep 2020

# **T**HÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
**C**OMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
*Mathématiques et Sciences et Technologies*
*de l'Information et de la Communication*
Spécialité : *Informatique*
        Par

## **Charles XOSANAVONGSA**

**Heterogeneous Event Causal Dependency Definition for the**
**Detection and Explanation of Multi-Step Attacks**

**Thèse présentée et soutenue à** RENNES, CENTRALESUPÉLEC, le 23 juin 2020
**Unité de recherche : IRISA**
**Thèse N° : 2020CSUP0003**

**Composition du jury :**

Président du jury :        Hervé DEBAR                Professeur, Télécom Sud Paris

Rapporteurs :        Isabelle CHRISMENT        Professeur, Télécom Nancy
                Sébastien MONNET        Professeur, Polytech Annecy-Chambéry

Dir. de thèse :        Éric TOTEL                Professeur, IMT Atlantique

Co-dir. de thèse :        Olivier BETTAN                Head of Theresis Cyber Security Lab, Thales

# ABSTRACT

Knowing that a persistent attacker will eventually succeed in gaining a foothold inside the targeted network despite prevention mechanisms, it is mandatory to perform security monitoring on the system.

The purpose of this thesis is to enable the discovery of multi-step attacks through event analysis. To that end, previous alert correlation work has aimed at building connections among events and between attack steps. In practice, this type of link is not trivial to define and discover, especially when considering heterogeneous events (i.e., events emanating from monitoring systems deployed in different abstraction layers of the monitored system), and the literature lacks a formal definition of these connections. We argue that these connections among heterogeneous events correspond to a causal dependency relationship among events.

Inspired from two causality models from the distributed system and the security research areas, i.e., Lamport's [Lamport, 1978] and d'Ausbourg's [d'Ausbourg, 1994] models, we have thereby proposed a formal definition of this relationship called *event causal dependency*. The relationship enables the discovery of all events, which can be considered as the cause or the consequence of an event of interest (e.g., an alert produced by an attacker action). To the best of our knowledge, our work is the first one to propose a definition of the causal dependency relationship among heterogeneous events. We present how existing work permits the computation of parts of the overall model, and detail our implementation, which exclusively leverages existing monitoring facilities (e.g., auditd, and Zeek network-based intrusion detection system) to produce events. We show that our implementation already yields a good approximation of our model.

# ACKNOWLEDGEMENTS

poussé à donner le meilleur de moi-même. Je tiens également à remercier tout particulièrement Tatiana, Raphael, Soudrudy, et Bounsana pour m'avoir aidé à me relever pendant les moments les plus difficiles de ma thèse. Je vais terminer ces paragraphes en remerciant mes grands parents, à qui je dédie ce manuscrit.

# RÉSUMÉ SUBSTANTIEL EN FRANÇAIS

Du fait de leur environnement compétitif, les entreprises sont sujettes à l'espionnage industriel, le sabotage ou encore le vol de données. Le nombre de rapports de sécurité détaillant les brèches subies par les entreprises ne fait qu'augmenter. Il nous rappelle la réalité de la menace des attaques ciblées et de leurs conséquences sur les finances et la réputation des entreprises. Le constat est le suivant : *malgré tous les moyens de prévention mis en place, un attaquant motivé trouvera toujours le moyen d'infiltrer un réseau informatique.*

**Attaques Multi-Étapes.** Pour atteindre ses objectifs, un attaquant doit généralement effectuer plusieurs actions consécutives. De telles attaques sont connues sous le nom d'attaques multi-étapes et peuvent potentiellement rester longtemps inaperçues. Ceci s'explique notamment par le fait que, en dehors du contexte de la sécurité informatique, certaines étapes de l'attaque peuvent potentiellement être considérées comme étant des ensembles d'actions légitimes. Ceci les rend plus difficile à détecter.

**La Supervision de Sécurité est Indispensable.** Partant du constat que les mécanismes de prévention d'intrusion sont insuffisants, il est indispensable de mettre en place des moyens d'observer le système [Anderson, 1980]. Pour cela, les équipes de supervision de sécurité déploient différents types de sondes leur permettant d'observer le système sous plusieurs angles (réseau, système d'exploitation, application...) et d'enregistrer les observations dans des fichiers de journalisation sous la forme d'*événements*[1]. En particulier, les équipes de supervision déploient des systèmes de détection d'intrusion (appelés *Intrusion Detection Systems* (IDS) en anglais). En pratique, les événements produits par ces sondes, ainsi que les IDS, sont la seule source d'information disponible pour le débogage post mortem ou l'investigation d'attaques numériques. *L'objectif de cette thèse est de permettre la découverte de scénarios d'attaque multi-étapes à travers l'analyse d'événements de sécurité.*

**Détecter des Scénarios d'Attaque grâce à la Corrélation d'Alertes.** Un événement représente l'observation d'une action spécifique ayant été effectuée dans le système supervisé. Les événements doivent donc être analysés pour pouvoir en tirer une vision d'ensemble du système supervisé. Dans le contexte de la supervision de sécurité, un des objectifs principaux de cette analyse est de permettre *la découverte, ainsi que la compréhension, des différentes étapes constituant une attaque.* Ce rôle est joué par la *corrélation d'alertes* [Jakobson & Weissman, 1993].

Les méthodes de corrélation d'alertes cherchent à construire des liens entre les événements produits par les différents types de sondes et d'IDS déployés dans le système supervisé. Dans le contexte de la détection et décou-

---

1 Un « événement » correspond à toute information journalisée par une sonde ou un IDS. Lorsqu'un événement est produit par un IDS, il peut aussi être appelé « alerte ».

verte de scénario d'attaque multi-étapes, la corrélation de différents types d'événements est indispensable pour permettre leur complète compréhension [Valeur et al., 2004]. En pratique, ces liens sont difficiles à définir et découvrir, notamment lorsque l'on considère l'analyse d'événements hétérogènes.

L'approche typique pour corréler les événements et détecter des attaques est d'écrire des règles de corrélation. Un telle règle correspond à une description explicite des ensembles, ou séquences, d'événements correspondant aux observations des actions composant une attaque. Malheureusement, la formulation de règles de corrélation est très difficile car elle se base sur la fusion des points de vue du défenseur et de l'attaquant. Autrement dit, elle nécessite d'avoir une connaissance fine du système supervisé (c.à.d. connaître sa topologie, sa cartographie, ses vulnérabilités, ainsi que ses capacités de supervision), ainsi que de la créativité pour pouvoir imaginer les scénarios d'attaque potentiels.

Notre étude de la corrélation d'alertes nous a amenés à la réflexion suivante : grâce à la corrélation, un analyste de sécurité espère trouver des événements *causalement dépendants*. Autrement dit, *les méthodes de corrélation d'alertes cherchent idéalement à découvrir des liens de dépendance causale entre les événements.*

**Vers la Recherche de Liens de Causalité entre Événements.** Dans la continuité de nos recherches sur la corrélation d'alertes, nous nous sommes naturellement dirigés vers l'étude de la causalité et de sa définition dans le contexte de la supervision sécurité. Nous avons donc étudié la notion de dépendance causale dans un contexte plus général que la corrélation d'alertes. Ceci nous a mené à l'étude des domaines de recherches tels que les systèmes distribués, les flux d'information, ainsi que la provenance. Suite à notre étude de la littérature, nous sommes arrivés au constat suivant : *la littérature manque d'une définition formelle de la relation de dépendance causale entre événements hétérogènes.* Nous pensons que définir, ainsi que pouvoir calculer, ces relations de dépendance causale permettrait de simplifier le procédé d'identification de scénario d'attaque et d'améliorer les capacités d'investigation d'attaques.

En pratique, partant d'un événement d'intérêt $e$, tel qu'un indice de compromission ou une alerte, un analyste de sécurité cherche à identifier tous les événements pouvant être considérés comme les causes ou les conséquences de $e$. Idéalement, cet ensemble d'événements contiendrait toutes les traces laissées par les actions de l'attaquant dans le système supervisé. Notre raisonnement est le suivant : si les relations de dépendance causale entre événements peuvent être définies et calculées, alors la découverte de scénarios d'attaque se traduit en des parcours de graphes de dépendance causale où les nœuds et les flèches correspondent respectivement aux événements et à leurs liens de dépendance causale.

## DÉFINITION DE LA RELATION DE DÉPENDANCE CAUSALE ENTRE ÉVÉNEMENTS

Dans ce manuscrit de thèse, nous proposons une définition claire de la relation de dépendance causale entre événements. L'objectif est de définir un modèle formel permettant, dans l'idéal, d'unifier les travaux existants portant sur la notion de dépendance causale entre événements hétérogènes, c'est-à-dire, des événements issus de différentes couches d'abstraction (réseau, système d'exploitation, application).

Suite à notre étude de la littérature, ainsi qu'à notre quête de définir ce qu'est la relation de dépendance causale entre événements hétérogènes, nous sommes parvenus aux réflexions suivantes :

(a) Un système informatique peut être modélisé comme un ensemble d'objets actifs et passifs décrits par leur état. Les objets actifs ont la possibilité d'effectuer des actions et d'interagir avec les autres objets du système. Les objets passifs, quant à eux, ne peuvent pas effectuer d'actions. Un objet passif peut être influencé par un autre objet actif, qui effectue des actions sur lui, ou par un autre objet passif, par le biais d'un objet actif. Par exemple, les processus d'un système correspondent à des objets actifs, tandis que les fichiers correspondent à des objets passifs. Les processus peuvent interagir entre eux, ainsi qu'avec les objets passifs du système;

(b) Il existe des liens de dépendance causale entre les états des objets, les actions effectuées par les objets, ainsi qu'entre leurs états et leurs actions. Considérons par exemple un processus écrivant dans un fichier. L'état du fichier, autrement dit la valeur de son contenu, dépend causalement de l'état du processus au moment de l'écriture;

(c) Les systèmes de supervision permettent d'observer et d'enregistrer, sous la forme d'événements, ces actions et/ou états. Par exemple, un système de supervision peut être mis en place pour observer et enregistrer les interactions entre les processus et les fichiers à travers les appels systèmes;

(d) Si nous sommes capables de déterminer les relations de dépendance causale entre les actions et les états, nous pouvons transférer cette connaissance aux événements pour pouvoir déterminer des liens de dépendance causale entre événements.

Ces réflexions nous ont amené à proposer et définir trois nouvelles relations de dépendance causale :

1. La *relation de dépendance causale entre actions contextuelles*. Cette relation a pour objectif de modéliser le fonctionnement du système supervisé. Elle se base sur les réflexions (a) et (b) précédemment décrites;

2. La *relation de dépendance causale entre événements contextuels*. La définition du concept d'événement contextuel a pour objectif de modéliser le lien entre actions contextuelles et événements bruts. De manière succincte, une action contextuelle peut être observée, et enregistrée sous

la forme d'événements, par différentes systèmes de supervision. La relation se base sur les réflexions (c) et (d) précédemment décrites;

3. *La relation de dépendance causale entre événements bruts.* Cette relation a pour objectif de permettre la découverte de tous les événements étant causalement liés à un événement d'intérêt.

———————

Les paragraphes suivants présentent plus en détails le modèle que nous avons défini. Dans un premier temps, nous commençons par introduire la notion d'*action contextuelle* ainsi que sa relation de dépendance causale. Nous décrivons ensuite progressivement les liens qui unissent *actions contextuelles* et *événements*. Nous pourrons alors définir la notion de dépendance causale entre événements bruts.

Ces trois relations s'inspirent de deux modèles de causalité précédemment définis dans les domaines des systèmes distribués (modèle de Lamport [Lamport, 1978]) et de la sécurité (modèle de d'Ausbourg [d'Ausbourg, 1994]).

### Relation de Dépendance Causale entre Actions Contextuelles

**Définition d'une Action Contextuelle.** De manière informelle, une action contextuelle est composée de :

1. l'action effectuée par un objet, de manière similaire au modèle de Lamport [Lamport, 1978];

2. la valeur du contexte de l'objet au moment où l'action a été effectuée, autrement dit, l'état de l'objet, au sens du modèle défini par d'Ausbourg [d'Ausbourg, 1994].

Pour résoudre le problème de dépendance entre objets de types différents, nous devons distinguer deux catégories d'objets : les objets *actifs*, qui effectuent des actions (tel que les processus ou les interfaces réseau), et les objets *passifs* (comme les conteneurs d'information tels que les fichiers, sockets, ...), qui ne peuvent pas effectuer d'actions. Un objet actif est supposé effectuer des actions pouvant être liées aux contextes de l'objet. Pour les objets passifs, seul leur contexte peut être observé.

**Definition 1** $\text{ObjectActions}(o)$ *correspond à l'ensemble des actions produites par un objet* $o$*, actif ou passif.* $\text{ObjectActions}(o) = \{a_i\} \cup \{\emptyset\}$ *avec* $\{a_i\}$ *l'ensemble des actions pouvant être effectuées par* $o$*, et* $\emptyset$ *l'absence d'action.*

Par exemple, les appels système invoqués par un processus $p$ pour requêter les services du kernel correspondent à des actions de *ObjectActions*$(p)$.

Nous pouvons maintenant introduire formellement la notion d'action contextuelle :

**Definition 2** *Une action contextuelle est un couple* $(a, (o, t))$*, où* $a$ *correspond à l'action effectuée par l'objet* $o$*, avec* $a \in \text{ObjectActions}(o)$*, et* $(o, t)$ *correspond à l'état de l'objet* $o$ *au temps* $t$*. Dans le cas où* $a \neq \emptyset$*,* $t$ *correspond au temps où l'action a été effectuée.*

**Définition d'une Session.** Étant donné deux actions $a$ et $b$ produites par un processus donné telles que $a \prec b$, « $\prec$ » étant la relation *happened-before* [Lamport, 1978], Lamport précise que $b$ peut être causalement dépendant de $a$. Nous voulons définir un modèle plus précis permettant de casser la relation de causalité entre $a$ et $b$ lors de l'évolution d'un objet, c'est-à-dire, si l'état de cet objet est indépendant de ses états précédents au sens de d'Ausbourg. En pratique, de nombreux services ne gardent pas en mémoire les différentes séquences d'exécution. Ceci implique que l'exécution d'un objet donné peut être divisée en intervalles de temps où les exécutions sont partiellement ou complètement indépendantes l'une de l'autre. Dans notre modèle, un tel intervalle dans l'exécution d'un objet est appelé *session*.

**Definition 3** *Une session* $\mathrm{session}_n(o)$, *d'un objet* $o$, *est une séquence d'actions contextuelles* $(a_i, (o, t_i))$, *où* $a_i \in \mathrm{ObjectActions}(o)$, *telle que :*
$$\mathrm{session}_n(o) = \left\{ (a_i, (o, t_i)) \,/\, (o, t_i) \to (o, t_{i+1}) \land (o, t_{end_{n-1}}) \not\to (o, t_{start_n}) \right.$$
$$\left. \land\, (o, t_{end_n}) \not\to (o, t_{start_{n+1}}) \right\}, \textit{ avec } t_{start_n} \textit{ correspondant au temps où la pre-}$$
*mière action contextuelle de* $\mathrm{session}_n(o)$ *est effectuée,* $t_{end_n}$ *correspondant au temps où la dernière action contextuelle de* $\mathrm{session}_n(o)$ *est effectuée, et « $\to$ » correspondant à la relation de dépendance causale définie par d'Ausbourg [d'Ausbourg, 1994].*

Une exécution d'un objet $o$ est l'union de toutes les $\mathrm{sessions}(o)$. La notion de sessions s'applique à tout type d'objet. Exemples : un fichier peut être vidé de son contenu; pour un processus Apache, deux requêtes consécutives sont indépendantes. En pratique, une action commençant une nouvelle session peut être produite par une application ou encore le système d'exploitation.

La notion de session n'est pas nouvelle. En particulier, les actions délimitant les sessions au sein des processus ayant une longue durée de vie, tels que les services par exemple, ont fait l'objet d'une étude approfondie pour permettre de diminuer le nombre de fausses dépendances causales entre les actions [Lee et al., 2013a].

**Définition de la Relation de Dépendance Causale entre Actions Contextuelles.** Le concept d'action contextuelle prend en compte les actions effectuées par les objets et leurs états au moment où les actions sont effectuées. Ceci nous permet de profiter des modèles de Lamport et de d'Ausbourg pour définir une relation de dépendance causale entre différents types d'actions. Nous avons appelé cette nouvelle relation *dépendance causale entre actions contextuelles*. Cette dernière est dénotée « $\mapsto$ », et est définie sur l'ensemble de toutes les actions contextuelles produites par tous les objets du système.

**Définition 4** *Étant donné deux actions contextuelles* $(a_1, (o_1, t_1))$ *et* $(a_2, (o_2, t_2))$, *l'action contextuelle* $(a_2, (o_2, t_2))$ *est causalement dépendante de l'action contextuelle* $(a_1, (o_1, t_1))$, *dénoté* $(a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2))$, *lorsque :*
1. $o_1$ *et* $o_2$ *correspondent au même objet* $o$, *et il existe* $n$ *tels que :*
   $(a_1, (o, t_1)) \in \mathrm{Session}_n(o)$, $(a_2, (o, t_2)) \in \mathrm{Session}_n(o)$ *et* $t_1 < t_2$;
2. *ou,* $o_1 \neq o_2$, *et* $(o_1, t_1) \to (o_2, t_2)$, *c'est-à-dire que les états des deux objets sont causalement dépendants au sens de d'Ausbourg. Autrement dit, il existe un flux d'information de l'état* $(o_1, t_1)$ *vers l'état* $(o_2, t_2)$;

3. *ou, $o_1 \neq o_2$, et l'action $a_1$ correspond à l'envoi d'un message $m$ et l'action $a_2$ correspond à la réception de $m$, ce qui signifie que $a_1 \prec a_2$ en utilisant la relation happened-before de Lamport;*

4. *ou $\exists (c, (o, t))$ tel que $(a_1, (o_1, t_1)) \mapsto (c, (o, t))$ et $(c, (o, t)) \mapsto (a_2, (o_2, t_2))$.*
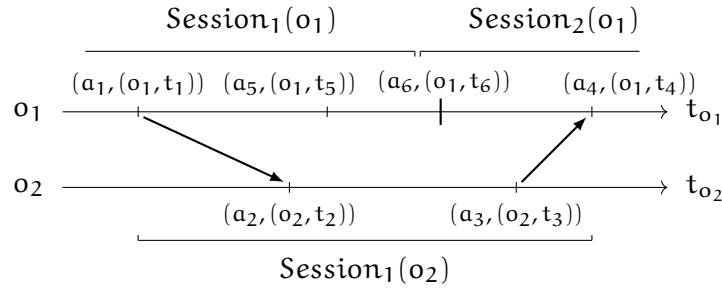


**Figure 1:** Illustration de la relation de dépendance causale entre actions contextuelles issues de sessions différentes.

La figure 1 illustre l'utilisation de notre modèle. La relation « $\mapsto$ » étant transitive, nous avons par exemple $(a_1, (o_1, t_1)) \mapsto (a_4, (o_1, t_4))$ même si ces deux actions contextuelles appartiennent à deux sessions différentes. Il est important de noter que les deux objets possèdent leur propre horloge, $t_{o_1}$ et $t_{o_2}$, et que notre modèle ne nécessite pas qu'elles soient synchronisées.

### Des Actions Contextuelles vers les Événements

Comme défini dans [European Commission, 2010], un événement correspond à « une action identifiable ayant lieu sur un dispositif et étant enregistrée comme une entrée de journal ». En pratique, les actions contextuelles peuvent donc être observées par les sondes déployées dans le système supervisé. Il est important de souligner qu'une action contextuelle peut ne pas être observée et journalisée. C'est le cas lorsqu'aucune des sondes déployées ne permet d'observer ce type d'action contextuelle. Prenons l'exemple d'un système où seul le réseau est supervisé à l'aide d'un NIDS. Les interactions entre les processus et les fichiers ne pourraient pas être observées dans ce cas. Par conséquent, certaines actions contextuelles peuvent être manquées par l'analyste de sécurité. À l'opposé, une action contextuelle peut également être observée par plusieurs sondes de type différent. Les événements correspondant à une même action sont alors répartis sur différents fichiers de journalisation.

**Définition des Événements Contextuels : Observations d'Actions Contextuelles.** Étant donné l'ensemble des événements journalisés du système, dénoté $\mathbb{E}$, chacun des événements est produit à un temps donné, c'est-à-dire au moment de son horodatage, en observant une action contextuelle effectuée par un objet.

**Definition 5** *Un* événement contextuel *est un triplet* $(e, o, t_e)$ *où* $e \in \mathbb{E}$, o *représente l'objet observé et* $t_e$ *est l'horodatage de l'événement* e.

D'après la définition 2, l'action $a$ d'une action contextuelle donnée $(a, (o, t_a))$ peut représenter une action réelle ou l'absence d'action. Par conséquent, $a$,

et $(o, t_a)$ peuvent ne pas être observable. On peut ainsi étendre la définition précédente en introduisant l'événement contextuel $(\emptyset, o, t_a)$ correspondant à l'absence d'observation de $a$ au temps $t_a$. Nous pouvons maintenant introduire la fonction *Obs* qui associe une action contextuelle à un ensemble d'événements contextuels correspondant à l'observation de cette unique action contextuelle.

**Definition 6** *Étant donné une action* $a \in \mathrm{ObjectActions}(o)$ *effectuée au temps* $t_a$, *l'observation d'une action contextuelle correspond à l'ensemble :*
$$\mathrm{Obs}\big((a, (o, t_a))\big) = \{(e_i, o, t_{e_i})\} \cup \{(\emptyset, o, t_a)\}, \text{ où } e_i \in \mathbb{E}, \text{ et } (\emptyset, o, t_a) \text{ correspond}$$
*à l'absence d'observation de* $a$ *et donc à l'absence d'événement.*

**Définition de la Relation de Dépendance Causale entre Événements Contextuels.** Maintenant que nous avons fait le lien entre les actions contextuelles et les événements contextuels, nous pouvons définir la relation de dépendance causale entre événements contextuels dénotée « $\rightharpoonup$ ».

**Definition 7** *Étant donné deux événements contextuels* $(e_1, o_1, t_{e_i})$ *et* $(e_2, o_2, t_{e_j})$, $(e_2, o_2, t_{e_2})$ *est causalement dépendant de* $(e_1, o_1, t_{e_1})$, *dénoté* $(e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2})$, *si et seulement s'il existe deux actions contextuelles* $(a_1, (o_1, t_1))$ *et* $(a_2, (o_2, t_2))$ *telles que* $(e_1, o_1, t_{e_i}) \in \mathrm{Obs}\big((a_1, (o_1, t_1))\big)$, $(e_2, o_2, t_{e_j}) \in \mathrm{Obs}\big((a_2, (o_2, t_2))\big)$ *et* $(a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2))$.

**Définition de la Dépendance Causale entre Événements.** Nous arrivons maintenant au résultat attendu du modèle, c'est-à-dire, la définition de la dépendance causale entre événements bruts dénotée « $\triangleright$ ».

**Definition 8** *Étant donné deux événements* $e_1$ *et* $e_2$, $e_2$ *est causalement dépendant de* $e_1$, *dénoté* $e_1 \triangleright e_2$, *si et seulement si* $(e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2})$, *où* $o_1$ *et* $o_2$ *correspondent respectivement aux objets observés et* $t_1$ *et* $t_2$ *sont les horodatages des événements.*

### Graphes de Cause et de Dépendance pour les Événements

Les relations « $\mapsto$ », « $\rightharpoonup$ » et « $\triangleright$ » définissent respectivement des ordres partiels sur les ensembles des actions contextuelles, des événements contextuels et des événements. Elles sont de plus transitives. Cette propriété nous permet de construire le *graphe de cause*, $\mathrm{cause}(e) = \{e' / e' \triangleright e\}$, ainsi que le *graphe de dépendance*, $\mathrm{dep}(e) = \{e' / e \triangleright e'\}$, d'un événement d'intérêt donné $e$. Ces deux graphes représentent tous les événements qui, respectivement, contribuent ou dépendent de l'événement donné.

La figure 2 illustre les graphes de cause et de dépendance d'une alerte. Ces graphes permettent à un analyste de sécurité d'obtenir tous les événements pouvant être considérés comme les causes ou les conséquences d'un événement d'intérêt. Ainsi, idéalement, les graphes de cause et de dépendance contiennent toutes les informations disponibles, et issues des événements, pour pouvoir investiguer des attaques. Il peut ainsi plus facilement déterminer les origines d'une attaque, ainsi que son impact sur le système supervisé.
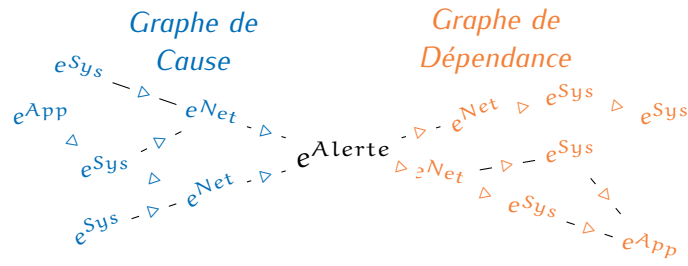
**Figure 2:** Illustration des graphes de cause et de dépendance d'une alerte

### Récapitulatif des Relations de Dépendance Causale Définies

La figure 3 résume les trois relations définies dans notre modèle. Ces relations peuvent être vues comme trois modèles représentant le système supervisé sous différents angles.
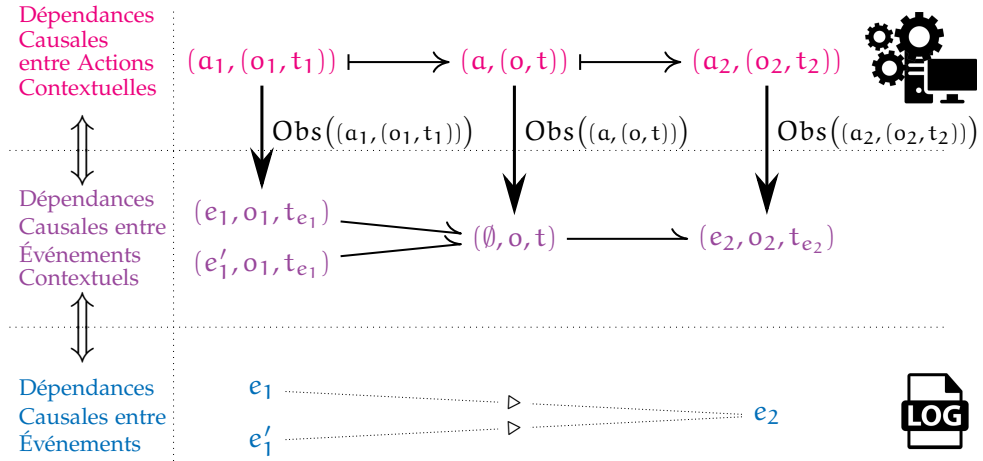


**Figure 3:** Schéma récapitulatif des 3 relations de dépendance causale

La ligne du haut (Dépendances Causales entre Actions Contextuelles) représente les actions, et les états, des objets du système, ainsi que les dépendances causales qui les lient. En d'autres termes, ce modèle décrit comment le système supervisé vit.

La ligne du bas (Dépendances Causales entre Événements) représente les événements produits par les divers types de sondes déployées pour observer le système supervisé, ainsi que les dépendances causales qui les lient. En d'autres termes, ce modèle représente une vue partielle, à travers les événements, du système supervisé.

La ligne centrale (Dépendances Causales entre Événements Contextuels) représente les événements contextuels, ainsi que les dépendances causales qui les lient. La notion d'événement contextuel a été introduit pour faire le pont entre les actions contextuelles et les événements bruts.

## IMPLÉMENTATION DU MODÈLE DE CAUSALITÉ

Les méthodes de calcul de dépendances causales permettent d'observer et journaliser les actions effectuées par les objets actifs du système supervisé. Les travaux existants permettent d'observer le système sous un point de vue donné (réseau, système d'exploitation, applicatif,...). Ils ne donnent donc qu'une information partielle de ce qu'il s'est réellement passé sur le système. Il est donc important de noter que les travaux existants ne permettent de calculer qu'une approximation du modèle.

### Stratégies d'Implémentation

Le modèle que nous avons défini peut être implémenté de diverses manières. Nous pouvons les catégoriser en deux groupes : celles qui adoptent une *stratégie descendante* (c'est-à-dire, partir des actions contextuelles pour aller vers les événements), et celles qui adoptent une *stratégie ascendante*, (c'est-à-dire, partir des événements pour aller vers les actions contextuelles). Les noms de ces stratégies font références à la figure 3.

**Stratégie Descendante.** Cette stratégie est basée sur le calcul et le suivi des relations de dépendance causale entre actions contextuelles. Ces relations sont ensuite utilisées pour calculer les relations de dépendance causale entre événements contextuels, ainsi qu'entre les événements bruts.

Une implémentation adoptant la stratégie descendante pourrait par exemple se baser sur les méthodes de capture et de rejeu des états du système ou des objets tel que dans [Ji et al., 2017].

**Stratégie Ascendante.** Cette stratégie consiste à tirer parti des informations sémantiques contenues dans les événements bruts pour calculer les événements contextuels, ainsi que leur relations de dépendance causale. Plus précisément, cette stratégie permet d'obtenir une approximation du modèle de dépendance causale. La qualité de cette approximation dépend des l'information contenue dans les événements bruts.

### Stratégie Ascendante : Implémentation Basée sur des COTS

Actuellement, notre implémentation du nouveau modèle de causalité adopte une stratégie ascendante.

**Stratégie de Supervision.** Pour cela, nous avons tout d'abord élaboré une stratégie de supervision permettant le calcul des événements contextuels, ainsi que des dépendances causales qui les lient, à partir des événements bruts produits par les sondes déployées. Pour cette implémentation, notre environnement de test est pour le moment exclusivement composé de systèmes d'exploitation Linux. Notre stratégie de supervision a la particularité de se reposer uniquement sur différents types de systèmes de supervision publiquement disponibles («COTS monitoring systems» en anglais). Plus précisément, elle se base sur:
1. la supervision des appels système à l'aide d'*auditd*, l'objectif étant d'observer les flux d'information entre les objets du Kernel Linux;

2. la supervision de *netfilter*, l'objectif étant d'enregistrer les connections établies pour chacune des machines;

3. la supervision du réseau à l'aide de *Zeek NIDS*, également connu sous le nom de *Bro NIDS*;

4. la supervision d'applications telles que le serveur HTTP Apache.

**Calcul du Modèle de Dépendance Causale entre Événements Contextuels.** Ces différents types d'événements permettent de calculer des événements contextuels, ainsi que les dépendances causales qui lies lient. L'architecture de notre implémentation permet le calcul de ce graphe d'événements contextuels et repose sur une base de données graphe pour le stocker. Une interface de visualisation permet à un analyste de sécurité d'investiguer des événements. Cette dernière lui permet notamment de construire les graphes de cause et de dépendance d'un événement d'intérêt donné.

Une évaluation simple de notre approche nous a permis de montrer que notre implémentation nous permet déjà d'identifier et de découvrir certaines étapes d'une attaque multi-étapes.

## CONCLUSION

L'objectif des travaux présentés dans ce manuscrit est de permettre à un analyste de sécurité d'identifier et de retrouver toutes les traces d'une attaque à travers l'analyse des événements. Pour cela, nous avons proposé de définir formellement la notion de dépendance causale entre les objets actifs, les objets passifs, ainsi que les événements hétérogènes produits par les différents types de sondes déployées dans le système supervisé.

Inspiré des relations de dépendance causale de Lamport et de d'Ausbourg, nous avons défini trois nouvelles relations de dépendance causale. Nous avons commencé par définir la notion d'action contextuelle et la relation de dépendance causale associée. Ceci nous a permis d'introduire progressivement la notion de dépendance causale entre événements journalisés. À notre connaissance, nos travaux sont les premiers à proposer une définition formelle de la relation de dépendance causale entre événements hétérogènes. Notre idée est la suivante : si nous pouvons identifier et construire les liens de dépendance causale entre les événements, alors la découverte de scénarios d'attaque se simplifie en une traversée de graphe.

Après avoir présenté le nouveau modèle de dépendance causale que nous proposons, nous avons décrit comment ce modèle se traduisait dans la réalité, dans le contexte de la supervision des réseaux d'entreprise. Plus précisément, nous avons décrit deux types de stratégies pour implémenter notre modèle de causalité : la stratégie descendante et la stratégie ascendante. À notre connaissance, aucun système ne fournit toutes les notions requises pour calculer notre modèle. Seules des parties de ces notions sont disponibles dans une même implémentation donnée.

Notre implémentation adopte la stratégie ascendante. Elle a la particularité de se baser uniquement sur l'analyse d'événements, *hétérogènes*, issus de *COTS* tels que Zeek NIDS et auditd, pour calculer la relation de dépendance causale entre événements contextuels. L'évaluation de notre implémentation montre qu'elle permet d'obtenir une bonne approximation de notre modèle, et de découvrir les différents événements pouvant être considérés comme les causes ou les conséquences d'un événement d'intérêt.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# INTRODUCTION

## 1 COMPUTER SYSTEMS—SOCIETY'S KEYSTONE

The range of application of computer systems goes from everyday life technologies, such as phones, to large information systems of business and state organizations. Regarding information, data has become the new digital gold as more and more aspects of organizations have been digitalized. Regarding industries, such as electric power, water treatment, and supply industries, information technologies provide stakeholders, engineers, and operators with remote access to their industrial plants. These few examples illustrate how critical computer systems are in our Society. They represent a significant means for data theft, industrial spying, and sabotage. The consequences of such attacks range from reputational aspects to economic and human loss. The Cost of a Data Breach Report, conducted each year by the Ponemon Institute, stated that the average total cost of a data breach was 3.92M dollars in 2019. Their study concerned 507 organizations, which suffered from a data breach, in 16 countries, and across 17 industry sectors. Additionally, the Verizon Data Breach Investigations Report reported 2,013 confirmed data breaches for the year 2019. Securing computer systems has become mandatory. In order to achieve this goal, prevention mechanisms such as firewalls, encryption, and access control, have been designed.

**You Will Be Breached...** Unfortunately, the frightening number of data breaches *reported* each year leads us to the following observation: a persistent attacker will *eventually* succeed in gaining a foothold inside the targeted network despite all the prevention mechanisms. 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 achieved. 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. Then, he maintains his foothold through the deployment of command and control channels, explores the network, and attempts to advance his opportunities by exploiting vulnerabilities or stealing passwords. The final stage, i.e., the attack objective, often consists of sensitive data leakage or sabotage [Hutchins et al., 2011]. Hence, detecting any stages of the attack and responding to the intrusion has become a priority.

## 2 THE NEED FOR SECURITY MONITORING

**Towards Situational Awareness.** Our research aims at helping security teams to comprehend an attack, ascertain its root causes, and identify all the compromised assets to ascertain its impacts. To do so, we focused our research on the *security monitoring* research field, i.e., the ability to monitor it, with a computer security point of view, in order to detect and respond to intrusions. Security monitoring is mandatory to enable the observation, recording, and detection of abnormal behaviors that would ideally correspond to attackers' actions. Produced records, also called traces or events, can be analyzed by the security team in order to satisfy their mission. Security monitoring's goal is to enable the three phases of *situational awareness* [Liu et al., 2017]: (1) Perception of the dynamics of relevant elements within the networks; (2) Comprehension of the situation, i.e., "how analysts combine, correlate, and interpret information;" (3) Projection, i.e., "the ability to make predictions based on the knowledge acquired through perception and comprehension." Situational awareness allows security monitoring teams to answers key questions when attacks are launched against their monitored system:

- What has happened to the monitored system? Or, in other words, what has the attacker done?
- What is the impact (damage and system's mission impact assessment)?
- Why did it happen? For instance, the security monitoring team should be able to identify the exploited vulnerability.

Answering these questions allows the security monitoring team to plan the response to the attack.

**Alerting When Suspicious Behaviors are Observed.** In practice, most of the time, the team in charge of security monitoring *exclusively* deploys the security monitoring's spearheads to gain situational awareness, namely, Intrusion Detection Systems (IDS). IDSs trigger alerts when suspicious behaviors are observed. Ideally, these alerts would be 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 monitor a given application or the system's access control policy. Moreover, 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 related attack. The attack may also contain footprints that are not sufficiently suspicious to trigger an alert. Such footprints might even appear benign and legitimate in the system at first. 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. Another issue when solely analyzing a single type of data source is that a threat might be impossible to discover without merging information coming from different origins [Bass, 2000][Abad et al., 2003].

**The Need to Correlate Alerts to other Sources of Information.** The discovery of correlations among different sources of information is the key to identify the steps composing an attack scenario [Valeur et al., 2004]. Therefore, IDS

alerts have to be correlated to contextual information, other alerts, and other types of events produced by complementary monitoring systems. We argue that a holistic security monitoring strategy consequently relies on different types of monitoring systems in order to complement IDSs' coverage and be able to thwart any action performed by attackers. More specifically, these techniques can be leveraged at the different abstraction layers of the monitored system (e.g., the application, operating system, and network abstraction layers) according to the security team's monitoring strategy. Naturally, the events produced by these techniques in these different abstraction layers are heterogeneous. More specifically, their semantics, their formats, and the quantity of information they conveyed vary and complement each other. Ideally, these events would contain all the information needed to perform attack investigation or digital forensics tasks. In fact, every single instance of events produced by monitoring systems may contain valuable information in order to comprehend better an ongoing attack or intrusion.

**Writing Explicit Correlation Rules is Hard.** The typical approach to correlate events and alerts is to rely on a base of correlation rules that explicitly describe the sequences or set of events and alerts that are parts of the consequence of attacks inside the monitored system [Eckmann et al., 2002][Totel et al., 2004][Goubault-Larrecq & Olivain, 2008]. Security Information and Event Management (SIEM) tools have adopted this technique for years in [Nicolett & Kavanagh, 2011]. However, the formulation of correlation rules can be considerably tricky because it requires the merging of the defender's and the attacker's perspectives. More specifically, it respectively requires having a precise knowledge of the monitored system (i.e., its topology and cartography) and deployed IDSs [Godefroy et al., 2015a], as well as the creativity to imagine possible attack scenarios.

**The Necessity for Novel Approaches.** The approach mentioned above has reached its limits, and new methods must be defined to automatically discover the relationships among traces related to the same attack. Such methods and related tools would enable them to discern the attack and implement actions to avert similar scenarios in the future; in other words, detect, respond to, and report on intrusions.

## 3 RETRIEVING ATTACKERS' TRACES—THE SEARCH OF CAUSALITY

Following the lead of alert and event correlation research and the quest for novel approaches, the study of the concept of correlation has naturally brought us to study the concept of causality. We argue that in order to infer the root causes and the overall scenario of an attack, alert and event diagnosis rely on *causal reasoning*.

**From Causal Chains of Actions to Causal Chains of Events.** A multi-step attack tends to follow a logical progression of actions. We argue that this cor-

responds to cause–effect relationships among the attacker's actions. Thus, the detection and discovery of multi-step attacks would ideally highlight the causal chains of actions performed by the attacker. Recorded events that relate to these actions are the direct consequence of these same actions: they correspond to their observation. Consequently, the cause–effect relationships among the attacker's actions can be translated into *causal dependency relationships* among the corresponding events. An event can then be seen as the natural consequence of the previous one.

**Causal Dependency Graph of Events—The Holy Grail.** The concept of causal dependency among events is particularly interesting regarding our research field. Indeed, the goal of comprehending an attack, as well as its root causes and impact, translates to computing the set of events that causally depend on and the set of events that are causal consequences of a given suspicious event related to this attack. In other words, if causal dependency relationships among events can be defined and computed, the discovery of attack scenarios translates to simple graph traversals, where graphs correspond to causal dependency graphs with events as nodes and causal dependency relationships as directed edges.

**The Lack of a Definition of the Causal Dependency Relationship among Events in the Literature.** Previous alert and event correlation work aim at building connections among events and between attack steps. In practice, the relationship among events is not trivial to define and discover, especially when considering heterogeneous events. We noticed that the literature lacks a formal definition of these connections. We argue that the relationship definition our research community is looking for corresponds to a causal dependency relationship among events.

**Causal Dependencies Computation is Hard.** Computing causal dependency among events is a difficult task and has been studied for years, especially in distributed systems [Schwarz & Mattern, 1994]. Several methods have been introduced in order to discover causal relationships among the actions of distributed processes in these types of systems. Regarding security, precise definitions have also been introduced to reflect object state causal dependencies [d'Ausbourg, 1994]. Object state causal dependencies are 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 said that their states are causally linked. Recently, we could observe 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 [Kwon et al., 2018] among events, mining data to discover patterns (such as temporal invariant properties) [Beschastnikh et al., 2011], 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 em-

beds the business logic, whereas an OS only identifies requests from user space applications through system calls. Additionally, network packet analysis and alert production, either from NIDS or HIDS, are challenging to reconcile. These examples illustrate the semantic gaps among different layers of abstraction and different data sources.

# 4 CONTRIBUTIONS

The work presented in this manuscript focuses on the computation of correlations among heterogeneous events to help information security professionals perform attack investigation, and discover their causes and consequences.

Reviewing the literature in order to study our research field in-depth, we came up with a comprehensive state of the art covering the security monitoring research field, i.e., the study of monitoring systems, alert and event correlation and causal dependency computation. Parts of this reviewing work has been leveraged to publish a short paper at the 4$^{th}$ *Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information* (RESSI 2018) [Xosanavongsa et al., 2018].

**Towards a Formal Definition of the Causal Dependency Relationship among Heterogeneous Events**

In order to address the problem of computing correlations among heterogeneous events, we propose a formal definition of the causal dependency relationships among events emanating from heterogeneous monitoring systems (i.e., monitoring systems deployed at different abstraction layers of the monitored system). More specifically, we define three new relationships based on the merging of two previously defined causality models from the distributed system and the security research areas, i.e., Lamport's [Lamport, 1978] and d'Ausbourg's [d'Ausbourg, 1994] models, namely, the *contextual action causal dependency*, the *contextual event causal dependency*, and the *event causal dependency* relationships. These relationships enable 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 alert, or an indicator of compromise, produced by an attacker action).

To the best of our knowledge, our work is the first one to propose a definition of the causal dependency relationship among heterogeneous events. It contributes to the formalization of our research field: the discovery and investigation of multi-step attacks through causality analysis.

The formal definition we propose would ideally unify prior work on causal dependency relationships among heterogeneous events, i.e., events emanating from different abstraction layers (e.g., network, operating system, application). The three relationships we define provide a unified understanding of the causality relationships that can be defined between active entities, passive entities, and events.

This work has been presented at the $4^{\text{th}}$ IEEE European Symposium on Security and Privacy (EuroS&P 2019) [Xosanavongsa et al., 2019a], as well as at the $5^{\text{th}}$ *Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information* (RESSI 2019) [Xosanavongsa et al., 2019b]. These two papers, as well as this manuscript, present how actual work allows for the computation of parts of the overall model. They also present an implementation of our model that exclusively leverages existing monitoring facilities (i.e., auditd, netfilter, application logging systems, and Zeek NIDS) to produce events. We show that our implementation already yields a good approximation of our model.

## 5 OUTLINE

This manuscript is divided into two parts. The first part, consisting of Chapter 1, 2, and 3, introduces the reader to our research context and its state of the art. Chapter 1 aims to define the main terms that will be used throughout the rest of the manuscript, i.e., multi-steps attacks, cyber defense analysts, events, and alerts. Then, it presents the different means to observe the monitored system and produce events. Chapter 2 introduces the reader to the current methodologies to reason about alerts and events. It details the alert correlation process, the different methodologies proposed to detect or discover multi-step attacks, and alert correlation's challenges and limitations. More specifically, alert and event correlation work aims at building connections among events and between attack steps. We argue that these connections ideally correspond to causal dependency relationships. Our research, therefore, led us to the study of causal dependency relationships computation. This is presented in Chapter 3. It focuses on the concept of causal dependency relationships among active entities, passive entities, and events. More specifically, it presents the two models our work is inspired from, namely, Lamport's happened-before relationship [Lamport, 1978] and d'Ausbourg's causal dependency relationship [d'Ausbourg, 1994].

The second part of the manuscript, consisting of Chapter 4, 5 and 6, details our contribution. Based on the observation that the definition of the relationship among events is lacking, we introduce a clear definition of this relationship called *event causal dependency* in Chapter 4. This relationship is gradually introduced, starting from the definition of the *contextual action causal dependency* and the *contextual event causal dependency* relationships. Chapter 5 introduces how existing implementations allow the computation of parts of the model, as well as our implementation of the event causal dependency relationship. An assessment of our implementation is done in Chapter 6. It demonstrates how the proposed model is useful to avert real attack cases in distributed environments.

Finally, the last chapter gives a conclusion and future perspectives.

# Part I

# Context

# 1 | SYSTEM MONITORING AND COMPUTER SECURITY

The first chapter of this manuscript lays down the foundational concepts surrounding security monitoring. Section 1.1 starts by defining intrusions and multi-step attacks, presents who the actors that defend organizations are, and introduces the notions of actions, events, and monitoring. Section 1.2 presents why monitoring is critical in the context of security monitoring. Then, Section 1.3 introduces the reader to the security monitoring spearheads: intrusion detection systems. Finally, Section 1.4 illustrates the different means to observe the monitored system at its different abstraction layers.

## 1.1 BASIC TERMINOLOGY

This section aims to define the main terms that will be used throughout the rest of the manuscript. Following the introduction, it will present in more detail what has been previously introduced: what an intrusion is; the notions of single-step and multi-step attacks; who the actors that defend organizations are; the basic concepts of actions, events, and traces; and monitoring.

### 1.1.1 Defining Multi–Step Attacks

So far, we have mentioned attacks and multi-step attacks without defining them. The following paragraph starts by defining what an attack is as well as its surrounding notions.

#### *What is an Attack?—CIA Triad, Security Policy Violation, and Intrusions*

To define what an attack is, we have to go back to foundational security properties, namely, confidentiality, integrity, and availability. Protecting and securing the information resources of an organization consists in elaborating a security policy that will ensure the requirements of the CIA triad [CISSP et al., 2003]:

CONFIDENTIALITY "The protection of information within systems so that unauthorized people, resources, and processes cannot access that information."

INTEGRITY "The protection of system information or processes from intentional or accidental unauthorized changes."

AVAILABILITY "The assurance that a computer system is accessible by authorized users whenever needed."

Based on the CIA triad, we can now define what an intrusion is:

**Definition 1.1 - *Intrusion:*** *Any action or set of actions that attempt to compromise one of these three properties.*

According to ISO/IEC 27001 standard[1], an attack is defined as an "attempt to destroy, expose, alter, disable, steal or gain unauthorized access to or make unauthorized use of an asset." In fact, we can simply define an attack as an intrusion attempt [Heady et al., 1990].

### *Defining Single-Step Attacks*

As its name suggests, a multi-step attack is made up of several attack steps, which correspond to single-step attacks. The events generated by the different kinds of deployed monitoring systems generally correspond to individual malicious actions [Navarro et al., 2018]. These individual actions are also called single-step attacks. Some attacks, which involve several actions, can be considered as elementary attacks. For example, a distributed denial-of-service attack corresponds to an elementary attack, even though it corresponds to a distributed attack where several machines attack a single one at the same time. Another example of an elementary attack that involves several actions is network scanning, where a single machine probes many others. All the alerts raised by IDSs represent symptoms of potential single-step attacks. Thus, some work is still needed to understand attacks in their entirety.

### *An Attack is Generally Made of Several Attack Steps*

Considering the context of organizations, an attacker has to perform several actions to reach critical assets. Indeed, one single-step attack is very unlikely to be enough to complete an intrusion successfully. Moreover, decomposing an attack in several steps allows an attacker to be more stealthy as they can spread their actions across time. Also, he will likely need to leverage several hosts, using exploits or stolen credentials, for instance, to attain their goal. Actions performed by an attacker are thereby scattered across several hosts and the network. Such sophisticated attacks correspond to multi-step attacks, i.e., sequences of single-step attacks. They have also been denoted *multi-stage* [Chen et al., 2006] [Du et al., 2010], *attack plans* [Qin & Lee, 2004a], *attack strategies* [Huang et al., 1999] or *attack scenarios* [Ning et al., 2002] [Mathew & Upadhyaya, 2009] in the literature.

**Causal Chains of Actions.** To be able to perform a single-step attack, and ultimately achieve a multi-step attack, an attacker might need to perform various actions on the system to set it up in the right state, e.g., performing privilege escalation before uploading and executing malicious pieces of code. Some of these actions might be linked by a cause–consequence relationship, thereby forming a causal chain of actions. However, highlighting these relationships is a hard task. Ideally, the detection and discovery of multi-step attacks would highlight the causal chains of actions performed by the attacker.

---

1 https://www.iso.org/isoiec-27001-information-security.html

**Modeling Multi-Step Attacks.** Several models and frameworks have been proposed to model multi-step attacks. The ATT&CK knowledge database[2] created by the MITRE corporation is an example of such a framework. Single-step attacks are categorized by attack stages, e.g., initial access, lateral movement, or ex-filtration. Moreover, the framework provides the projection of some single-step attacks on logs. This information allows system administrators to be aware of the logging systems to enable and log files to monitor. Other attack models have also been developed to enhance their understanding or perform a better risk assessment. *Attack trees* [Kordy et al., 2014] and *attack graphs* [Sheyner et al., 2002] are examples of such attack models. These graph-based approaches allow quantitative and qualitative analyses of attack scenarios and allow cyber defense analysts to visualize them. As its name suggests, an attack tree models a threat as a tree, where the root node specifies an attacker's primary goal. It is then repeatedly and recursively refined into sub-goals. This refinement is done either disjunctively (i.e., the goal can be achieved in different and alternative ways), or conjunctively (i.e., all the branches, which correspond to attacker's steps, need to be taken to achieve the goal). The attack tree's leafs represent basic actions, i.e., atomic actions that can be easily understood and quantified [Qin & Lee, 2004a].

Attack graphs represent another popular security research field. Kordy et al. defined it clearly in their survey on graph-based attack and defense modeling [Kordy et al., 2014]: "The nodes of an attack graph represent possible states of a system during the attack. The edges correspond to changes of states due to an attacker's actions. An attack graph is generated automatically based on three types of inputs: attack templates (generic representations of attacks including required conditions), a detailed description of the system to be attacked (topology, configurations of components, etc.), and the attacker's profile (his capability, his tools, etc.)."

Such attack models can, in turn, be used as input for alert and event correlation techniques. More details will be presented in the following chapter.

### 1.1.2 Defenders—The Security Monitoring Team

We have seen that organizations' computer systems are threatened in several ways. To protect organizations' computer systems and detect any intrusion attempt, computer systems have to be audited on a regular basis. To do so, organizations have to hire dedicated information security professionals to protect and defend their assets.

Many different roles are needed to elaborate and maintain an organization's security policy. However, regarding our research context and goal, i.e., helping analysts to comprehend an attack, we are mainly interested in security monitoring-related roles. Based on the terminology proposed in the NICE framework developed by the NIST [Newhouse et al., 2017] and the one proposed by the French National Cybersecurity Agency (ANSSI) [ANSSI, 2015], the roles we are interested in are the following:

CYBER DEFENSE ANALYST – "uses data collected from a variety of cyber defense tools (e.g., IDS alerts, firewalls, network traffic logs.) to an-

---

2 https://attack.mitre.org

alyze events that occur within their environments for the purposes of mitigating threats." According to the ANSSI's terminology, cyber defense analysts also contribute to the deployment strategy of the security monitoring systems.

**CYBER DEFENSE FORENSICS ANALYST –** "analyzes digital evidence and investigates computer security incidents to derive useful information in support of system/network vulnerability mitigation." According to the ANSSI's terminology, cyber defense forensics analysts also formulate recommendations for detection capacities.

To perform their job, i.e., investigating attacks, these roles rely on "collected data" and "digital evidence." More specifically, they leverage traces, events, and alerts produced by the deployed security monitoring systems. Section 1.1.3 details these notions.

### 1.1.3 Actions, Events and Heterogeneity

Several standards on logging and monitoring have been written to make the industrial and research converge towards the same definitions [European Commission, 2010] [Chuvakin et al., 2008].

*Defining Action*

According to Oxford's English dictionary, an action is defined as "the fact or process of doing something, typically to achieve an aim." Following this definition, we can consider that any activity performed by a computing system can be described as sequences of actions, with an *action* being the execution of an instruction or a function, for example. Thus, when a user, or an other system, interacts with a given system, the system performs several actions to satisfy the request. Therefore, an attack step is represented by a set of actions performed by the system.

*Defining Event and Log*

Several standards describe what an *event* is, e.g., the MITRE Corporation's Common Event Expression (CEE) whitepaper [Chuvakin et al., 2008] or the European Commission's standard on logging and monitoring [European Commission, 2010]. Based on the definition proposed by the European Commission's standard, we define an event as the following.

**Definition 1.2  - *Event:** The observation and recording, by a monitoring system, of one or several identifiable* actions *happening on a monitored system. An event is recorded in a* log, *a log being a collection of events.*

Actions can be observed and recorded by one or several monitoring systems. Produced records correspond to *events*. Additionally, actions can be performed by a system at different abstraction levels, e.g., network, operating system, or application level. Different types of monitoring systems enable the observation of specific actions at these levels. Consequently, produced events are *heterogeneous*, either regarding their semantics, their formats, or encoding.

*Defining Trace*

Additionally to actions and events, the term *trace* will also be used throughout this manuscript. It englobes all the indications (e.g., events, alerts, or network packets) of the activity of a given subject (e.g., a process, a CPU, or an attacker). Thus, the terms *attacker's traces* refer to the projection of the attacker's actions on the logs, i.e., all the events that correspond to the attacker's actions on the monitored system.

### 1.1.4   Defining Monitoring

Section 1.1.4 presents in more detail the concept of monitoring. After defining it, it starts by introducing the reader to the concept of abstraction layers and presents its relationship with monitoring. It then illustrates the fact that monitoring systems can be either active or passive.

According to the European Commission's standard [European Commission, 2010], monitoring is defined as follows.

**Definition 1.3 - *Monitoring:*** *"The process of pro-actively checking systems for information security incidents, normally by checking log messages or periodically verifying that the system is responding."*

As it is, this definition is clearly oriented towards computer security. Naturally, in monitoring systems have to be deployed to monitor a given system.

*Observing the System at Different Abstraction Layers*

**Computers and Abstraction Layers.** Abstraction is a fundamental concept of computer science. Indeed, to arrange the complexity of computer systems, software engineering and computer science build them as layered architectures, each layer being an abstraction level that hides the complexity and working details of the lower ones. This technique allows studying each abstraction layer separately and building them independently while using interfaces to make them co-operate.

**An Illustration of the Observation of Different Abstraction Layers.** To illustrate the layers of abstraction and the ability to observe their activities, let's consider the simple example of a software application running on a single machine (corresponding to the "Monitored System" in Figure 1.1).

At the application abstraction layer, the application performs high-level tasks that embed the business logic. These tasks can be observed and recorded. For instance, the users' actions can be logged when performing significant tasks. Developers can use log printing statements (denoted "LPS" in Figure 1.1) to record them. Going one layer deeper, all of the processes running on the machine are handled by the OS. To interact with other processes or with resources such as the file system or the networking subsystem, the application sends requests to the OS. Thus, the application layer communicates with the OS abstraction layer through the system call application programming interface (API). System call invocations, which represent requests such as accessing a file, can be observed and recorded by leveraging
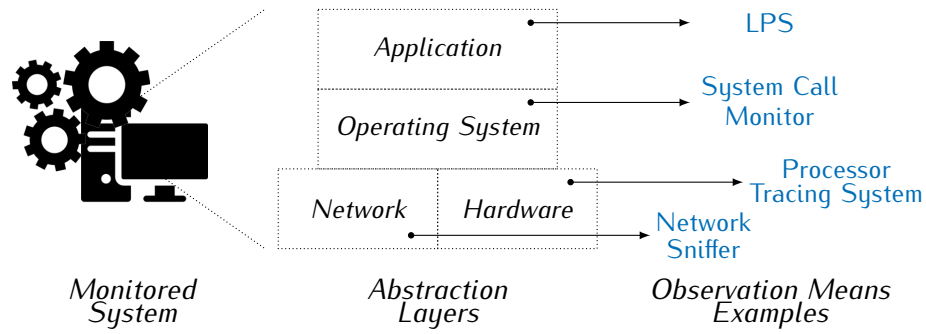
**Figure 1.1:** Illustration of the observation of different abstraction layers.

existing kernel modules (denoted "System Call Monitor" in Figure 1.1). Going one layer deeper, every piece of software is running on the hardware. The OS abstraction layer communicates with the hardware abstraction layer through drivers. Processes can be seen as a sequence of instructions, which can also be considered as actions, executed by the processor. Executed instructions and processor states can be traced using existing hardware facilities (denoted "Processor Tracing System" in Figure 1.1). Finally, going one layer deeper, our computer communicates with other ones by sending and receiving packets. Of course, this can also be observed and recorded using network sniffing technology (denoted "Network Sniffer" in Figure 1.1).

There is thereby a tight bond between observations, i.e., events, and actions occurring in the monitored system. An action performed can be observed and recorded by various monitoring systems at different abstraction layers. Moreover, actions performed at a given abstraction layer might provoke actions in other layers.

**Monitoring Different Abstraction Layers is Critical.** Regarding the context of computer security, an attack can be potentially observed at different abstraction layers. Depending on the problem to solve or the questions to answer, e.g., debugging or attack investigation, several abstraction layers might need to be observed at the same time to understand one or several attack steps. In other words, several sources of information have to be correlated [Valeur et al., 2004].

Merging information coming from different origins can actually enable the discovery of threats that would be impossible to discover otherwise, i.e., by analyzing only a single source of information [Navarro et al., 2018]. For instance, in [Abad et al., 2003], Abad et al. illustrate a case where an attack is not detected by their system call-based anomaly-based IDS alone as the related OS activity is not deemed statistically significant. However, the network activity corresponding to the related sequence of system calls shows an unusual amount of traffic. Combining these two data sources allow the identification of the attack.

*Active and Passive Monitoring Systems*

Additionally to the fact that monitoring systems can observe and record actions at different abstraction layers of the monitored system, they can be classified into two categories: passive and active.

**Passive Monitoring Systems.** Passive monitoring systems produce events either: (1) by observing the system's activity *as it is* and recording it, e.g., a network sniffer that performs full packet capture; or (2) through the execution of *log printing statements* (LPS) when logging systems are enabled. The following paragraphs present in more detail the latter class of passive monitoring systems, namely, logging systems, as well as another class of passive monitoring systems, called *tracing systems*.

**Logging Systems.** Computer systems and networks consist of devices and software components which generally have logging systems. Logging systems generally record system states, significant actions, or any useful information to enable analyses such as system status understanding, root cause determination, debugging when problems arise, and attack investigation.

A logging system produces an event *when the related running process executes a log printing statement* in its code. LPSs are generally placed manually at various critical places in the code to convey meaningful information. To do so, developers often rely on logging frameworks or libraries such as Log4J for Java-based applications and the `logging` library for Python-based applications.

Depending on the system, application, or device to monitor, different types of information can be conveyed into events. A network device will likely record network-related, whereas an OS Kernel will record its internal states.

Logging systems can also be configured according to the needs of the team who analyze logs. For instance, developers can configure, at the application level, the logging system they develop to produce debug log messages. In the context of the previous Python code example, debug messages were not produced as the logger has been configured to the `info` level. Here are a few logging system-related event categories described in Dr. Anton Chuvakin's book "Logging and Log Management" [Chuvakin et al., 2012]:

INFORMATIONAL "Messages of this type are designed to let users and administrators know that something benign has occurred." For instance, a server can log every command an administrator executed;

DEBUG "Debug messages are generally generated from software systems to help software developers troubleshoot and identify problems with running application code;"

WARNING "Warning messages are concerned with situations where things may be missing or needed for a system, but the absence of which will not impact system operation;"

ERROR "Error log messages are used to relay errors that occur at various levels in a computer system. Unfortunately, many error messages only

> give you a starting point as to why they occurred. Further investigation is often required to get at the root cause of the error;"

These categories illustrate the fact that log messages can be leveraged for various purposes.

**Tracing Systems.** Following the definition of the term *trace* (defined in Section 1.1.3), tracing corresponds to the recording of every action performed by a software, e.g., an operating system or a specific application. Tracing can be achieved either using LPSs in the source code of programs or by leveraging specific monitoring systems. Information provided by tracing is much more fine-grained than simple informational logging. However, tracing generates a lot of information in a short period. It is thereby only enabled in special cases such as debugging.

**Active Monitoring Systems.** Contrary to passive monitoring systems, active monitoring systems analyze the activity they monitor to possibly produce higher semantics events. For example, activity analysis might be pattern detection, metrics computation, or metrics visualization. Intrusion detection systems and event monitoring systems fall into this category of monitoring systems.

––––––––––––

Section 1.1 introduced the reader to basic terms that will be used throughout this manuscript, from attack-related terms such as multi-step attacks to defender-related terms such as events and monitoring. The following section presents why monitoring is critical, especially regarding the security monitoring context.

## 1.2 WHY IS MONITORING CRITICAL?—ENABLING SITUATIONAL AWARENESS

The previous paragraph illustrates the fact that monitoring techniques can be leveraged at the different abstraction layers of the system and for its various components. They thereby enable the observation and a better understanding of what is happening on the monitored system.

In practice, logs, and the data they contain, are often the only information available for attack investigation, digital forensics or, postmortem debugging of production systems failures.

The following paragraphs illustrate why monitoring is considered as critical. More specifically, it is considered as: the main debugging method; an important means to perform intrusion detection, attack detection, and investigation; and a key component of compliance programs regarding organizations' security processes.

### 1.2.1 The Main Debugging Method

As the documentation of Log4J[3] highlights, debuggers might not always be available nor applicable, especially for multi-threaded or distributed applications. Thus, inserting log printing statements (LPSs) into code is generally considered as the main debugging method. That is why the questions of where to log, e.g., the placement of LPS in the source code, and what to log, e.g., the information conveyed in the printing statement, are critical.

Despite appearances, logging is not simple. As the authors of "Learning to Log" [Zhu et al., 2015] emphasize: (1) logging too little might prevent the recording of key runtime information for postmortem analysis; (2) logging too much might produce "trivial and/or useless logs that eventually mask the truly important information, thus making it difficult to locate the real issue." Placing LPSs also means more code to write and maintain. Of course, they impact the system performance by consuming additional resources, e.g., CPU and I/O.

Most research work focuses on automating what to log [Yuan et al., 2012] [Yuan et al., 2012]. Fewer approaches propose methodologies for automatic log printing statements placement [Zhao et al., 2017]. These families of methodologies allow system developers to better capture their intent, the information, and events of interest.

### 1.2.2 Detecting Intrusions

From a security perspective, monitoring systems make up for prevention mechanisms. In fact, a persistent attacker will eventually succeed in gaining a foothold inside the targeted network despite all prevention mechanisms. The security monitoring team has thereby to deploy various monitoring systems at critical places to develop situational awareness, as defined in the introduction of this manuscript, of the system to protect. Such deployment would ideally enable cyber defense analysts to detect, investigate, and respond to incidents.

To detect potential security incidents, the monitored system has to be proactively analyzed. Therefore, security monitoring teams have to deploy specific active monitoring systems, as defined in Section 1.1.4, called *Intrusion Detection Systems* (IDSs). This class of monitoring systems is presented in more detail in Section 1.3.

In practice, enabling situational awareness is hard. The security monitoring methodologies currently applied are not efficient enough to perform effective detection and investigation of attacks. The launching of the Transparent Computing program, by the Defense Advanced Research Projects Agency (DARPA), in 2014, illustrates well this issue [Defense Advanced Research Projects Agency, 2014]. Considering the observation that "modern computing systems act as black boxes in that they accept inputs and generate outputs but provide little to no visibility of their internal workings," the program aims to respond to the lack of visibility in monitored systems and to counter the rise of complex multi-step attacks. More specifically, it

---

3 https://logging.apache.org/

aims to build enterprise-scale monitoring systems that highlight and record interactions and causal dependencies among system components to perform root cause analysis and damage assessment in the context of an attack. The fact that this program is still ongoing in 2019 shows that security monitoring remains an active research field in computer security.

### 1.2.3 Attack Investigation in the Context of Security Monitoring

Attack investigation often relies on log analysis to retrieve the attackers' traces and understand their context. For instance, security monitoring teams generally enable logging on their firewall and their web proxy to record connections and HTTP requests. However, other logging systems are often overlooked and ignored in the enterprise environment, especially regarding security-related log data [Australian Cyber Security Center, 2019].

Several projects and recommendations have been launched and proposed to counter this tendency. The following paragraphs present the event collection guidelines proposed by different Cybersecurity-related organizations, namely, OWASP, ANSSI, ACSC, JPCERT, and EUCERT.

**OWASP Security Logging Framework.** In 2014, the Open Web Application Security Project's (OWASP) Security Logging Framework was born out of the need for web applications to have the capacity to produce security-related events [OWASP, 2014]. The idea behind the project was to enhance events' context to be able to answer the questions: who, what, where, and when? Answering these questions allows a cyber defense analyst to get all the information needed to understand an attack, as well as its context, and respond to the incident. More specifically, given a suspicious event, they allow him to identify which user performed the related suspicious action (who and what), on which machine the action was performed (where), and the time it was performed (when).

**ANSSI's Security Recommendations for Logging Systems Implementation.** In 2013, the ANSSI released its recommendations for logging systems implementation [ANSSI, 2013]. It emphasizes on the fact that any logging system is indispensable for an information system's security, and is complementary to prevention systems. More specifically, ANSSI's recommendations do not detail what to log for specific devices, operating systems, or software. Instead, it focuses on the best practices for security-oriented logging. For instance, logging systems' clocks have to be synchronized, storage size for logs have to be estimated, logs have to be automatically exported to dedicated servers, used protocol for logs exportation should be based on TCP and cryptographic mechanisms. The fact that the French National Cybersecurity Agency did these recommendations clearly illustrates the importance of logging regarding Cybersecurity.

**ACSC's Logging Configuration Guidelines for Windows.** Based on the observation that the lack of visibility of activity occurring on workstations and servers is a common problem in organizations, the Australian Cyber Secu-

rity Center (ACSC) released its guidelines for Windows event logging in organizations in 2017 [Australian Cyber Security Center, 2019]. The report highlights the fact that this lack of visibility prevents investigation teams from performing effective investigations and responses to Cybersecurity incidents: already deployed host-based IDSs and prevention systems are not enough to attain these goals. According to the report, this gap could be filled by enabling the production of specific Windows event logs. The guidelines they developed have been designed to enable the detection and investigation of suspicious and malicious activity while keeping in mind the balance needed between the collection of significant events and the management of generated data. Events collected by following these guidelines are complementary to the alerts triggered by HIDSs and can, in turn, be analyzed by HIDSs to trigger alerts. Moreover, they allow investigation teams, i.e., cyber defense analysts, to understand the overall context of incidents better. The guidelines are general enough to cover all the stages of a multi-step attack, as modeled by Lockheedmartin's Cyber Kill Chain [Hutchins et al., 2011].

**JPCERT's Guidelines for Lateral Movement Detection.** Complementary to the general logging configuration guidelines made by the ACSC, the Japan Computer Emergency Response Team Coordination Center (JPCERT) focused its report on the detection of attacker's lateral movement through Windows event logs analysis [Japan Computer Emergency Response Team Coordination Center, 2017]. As modeled by Lockheedmartin's Cyber Kill Chain, lateral movement corresponds to one of the seven stages of a multi-step attack and is described as following: after gaining a foothold inside the targeted network, the attacker leverages different tools and methodologies to access other machines or sensitive resources, e.g., credentials. These can, in turn, be used to perform privilege escalation, steal more valuable resources, or compromise additional systems. Many of the actions performed during this stage can be captured and recorded in the Windows event logs. Studying the security incidents involving lateral movement, the JPCERT noticed some patterns in the attack methods, especially the tools used by the attackers in the lateral movement stage, e.g., "net", "ipconfig", "mimikatz". Tools were identified through the investigation of the traces left by their usage on the server and clients. Then, the JPCERT looked for the right logging configuration to obtain sufficient evident information in the Windows event logs. The report produced by the JPCERT is more detailed than the general logging configuration guidelines provided by the ACSC. Indeed, it contains the names of classic tools used by attackers, as well as various means to retrieve or detect their usage in the monitored system.

**EUCERT's Guidelines for Lateral Movement Detection.** Similarly to the JPCERT, the European CERT also made a report on the detection of attackers' lateral movement in infrastructures based on Windows Vista/7/8 [European Union Computer Emergency Response Team, 2017]. However, their report focuses on the exploitation of the Microsoft authentication protocol, and Kerberos protocol using stolen credentials.

These few project examples illustrate how important logging is regarding security. Of course, these guidelines are not perfect. They are limited: by

the expert knowledge of cyber defense analysts, i.e., they have to specifically know how to look for attack traces; by known attack cases that have already been observed; to one type of OS, i.e., Windows in these examples. Even though all these CERT reported on the lack of logging systems use to gain visibility, they do not seem to converge towards a standard. In fact, these projects have the same approach: they identify the minimal event information needed to enable the detection of specific attacks. In this thesis, we follow a similar approach: we identify the minimal event information needed to compute causal dependencies among events.

### 1.2.4 Digital Forensics

Log analysis can also be performed in a digital forensic investigation context where the goal is to help investigators reconstruct the timeline of events that happened on the crime scene. In this context, this discipline is called *Event Reconstruction* [Chabot et al., 2015], i.e., "the process of identifying the underlying conditions and reconstructing the sequence of events that led to a security incident." Here again, the heterogeneous nature of events and their quantity make the task of event reconstruction complex. Recent research efforts focus on automating some parts of the investigation process. As an example, Chabot et al. propose tools and approaches for the extraction, management, and reasoning on events by leveraging an ontology to capture event semantics and enhance their analysis [Chabot et al., 2015].

### 1.2.5 Logging and Compliance Program

To raise organizations' security standards, different kinds of compliance programs have been designed to evaluate their security processes, e.g., PCI Data Security Standard (DSS)[4] and ISO 27001. Organizations have to comply with given programs according to their business and corresponding regulations.

Similarly to the projects presented in the previous paragraph, compliance programs logging and event monitoring requirements highlight the importance of these monitoring systems regarding security. For example, the PCI DSS program requires the tracking and monitoring of all access to network resources, cardholder data, and user activities to enable effective forensics.

---

Previous sections defined the concept of monitoring and emphasized its importance regarding computer security. The following sections present in more detail *how* monitoring is performed to detect attacks or to observe different abstraction layers of computer systems. Section 1.3 introduces the reader to security monitoring's spearheads, i.e., intrusion detection systems.

---

4 https://www.pcisecuritystandards.org

## 1.3 INTRUSION DETECTION SYSTEMS

To observe and detect intrusion-related actions, a new category of methodologies and techniques has been created – Intrusion Detection. A software that implements them is called an *Intrusion Detection System*. An IDS either attempts to automatically detect attackers that try to break into the system, as well as attackers that already have a foothold inside the system or authorized users that are misusing system resources.

An IDS is a security monitoring system that raises alerts when noticing something suspicious. Analyzing these alerts, cyber defense analysts can then take countermeasures to bring back the system into a normal state. These countermeasures can be done either manually, or automatically with the use of another class of monitoring systems called: *Intrusion Prevention Systems* (IPSs). We will only focus on IDSs in this manuscript.

The intrusion detection discipline began in 1980 with Anderson's seminal work "Computer Security Threat Monitoring and Surveillance" [Anderson, 1980]. Diverse intrusion detection methodologies and techniques have been proposed since then. Different taxonomy propositions have been made to get a better view of the field [Debar et al., 1999] [Axelsson, 2000].

### 1.3.1  IDS Taxonomy According to the Analyzed Data Source.

According to the activity that is monitored, IDSs are generally classified into two categories, namely, Network-based IDS (NIDS) and Host-based IDS (HIDS). As its name suggests, a NIDS will monitor network-related activity by sniffing the traffic. A HIDS will monitor system or application behavior by watching its activities, or the event logs its activities produce. Following the diversity of event semantics, the variety of data sources that can be monitored makes IDSs heterogeneous. A few examples are presented in Section 1.4 for the most common abstraction layers.

### 1.3.2  IDS Taxonomy According to the Methodology Used.

Going further in the taxonomy of IDSs, intrusion detection methodologies can be divided into two categories: misuse-based and anomaly-based.

#### Misuse-based IDS

Misuse-based techniques characterize malicious activities *using specifications, i.e., patterns or sets of detection rules* [Debar et al., 1999]. Misuse detection is the most prevalent methodology used in commercial intrusion detection products, especially the signature-based detection techniques [Sommer & Paxson, 2010]. This popularity is mainly due to the fact that misuse-based techniques are expert knowledge-based approaches. These approaches are trusted by cyber defense analysts as they can easily understand why an alert was triggered, especially in the case of signature-based detection, where the triggered alerts contain the information related to the attack. The major limitations of misuse-based approaches rely on the fact that they cannot detect unknown threats. As threats are constantly evolving, these approaches are

prone to detection evasion. Thus, cyber defense analysts have to periodically update the knowledge base to keep up with evolving threats.

### *Anomaly–based IDS*

Anomaly-based techniques *consists in building a reference model* to detect deviations [Denning, 1987]. Compared to misuse detection, anomaly detection is expected to detect yet unknown threats. Anomaly detection techniques proceed in two stages: 1) design of a set of models of reference representing the legitimate behavior; 2) automatic detection of deviant behaviors according to the rules. Contrary to misuse-based approaches, anomaly-based IDS can likely detect unknown threats, e.g., new attack scenarios or new vulnerability exploits. However, the interpretation of the triggered alert is not straightforward as it does not refer to a specific malicious behavior. The only information known to the cyber defense analyst is that it is deviant compared to the legitimate behavior model.

**Building Reference Models.** Reference model building techniques for anomaly-based IDSs can be categorized into the following three families:

SPECIFICATION OF THE REFERENCE MODEL - Specifications are generally developed by performing in-depth analyses of attacks.

POLICY AS A REFERENCE MODEL - Policy-based techniques *define formal security policies to enforce to detect violations*, which are considered as intrusion attempts. Policy-based IDSs (PBIDS) fall into the category of anomaly-based IDSs. More specifically, security policies correspond to the reference models. The limitation of PBIDSs lies in the expressive power of their policy definition formalism. Blare is an example of PBIDS that enforces its security policy on the information flows of the operating system abstraction layer [Zimmermann et al., 2003].

LEARNING MODELS - Learning techniques, either supervised or unsupervised, *model legitimate, malicious behaviors, both at the same time, to classify an event as legitimate or malicious*. A learning model is trained on a labeled dataset containing legitimate and malicious data, in the case of supervised learning, or on an unlabeled dataset containing either legitimate or malicious data in the case of unsupervised learning. The learned model serves as the reference model for anomaly detection. For example, the reference model can be learned using methods such as automata or invariant modeling (presented in Section 1.4.5), or unsupervised learning, where the legitimate behavior is automatically induced from benign data, e.g., DeepLog [Du et al., 2017].

---

The previous section presented intrusion detection systems, i.e., the monitoring systems dedicated to attack detection, as well as their taxonomy. The next section presents monitoring systems in a more general context. It illustrates how monitoring can be implemented in different abstraction layers, namely, the application, OS, network, and hardware layers.

## 1.4 MONITORING ACTIVITY AT THE DIFFERENT ABSTRACTION LAYERS

We have previously mentioned that monitoring can apply to various sources of data, i.e., network, operating system, or application-related data. This section illustrates how monitoring can be applied in these different abstraction layers.

### 1.4.1 Application Layer

Considering the application abstraction layer, a typical example of active monitoring is the ability to load a module in the application to extract the desired information [Almgren & Lindqvist, 2001]. ModSecurity[5] is a good example of such a monitoring module. It is an open-source Web Application Firewall for web servers. It has the capability to thoroughly inspect HTTP traffic in real-time and perform user-defined actions, i.e., actions defined by the cyber defense analysts, thanks to its event-based programming language. Such actions might be access control, web application hardening with the restriction of selected HTTP features, or logging for security purposes like forensics analysis.

Another means to monitor an application is to trace it using tracing techniques. For example, considering a Python program, the `trace` module will record every statement and function executed, including their parameters and caller-callee relationships[6]. In the same fashion, the Linux OS kernel provides the system call *ptrace()* to enable a *tracer* process to observe and control the execution of a *tracee* process. The tracer process gets access to the tracee's memory and registers. Microsoft provides the Windows Driver Kit[7] to help developers perform software tracing. The kit relies on the Event Tracing for Windows facility to generate traces.

### 1.4.2 Operating System and Kernel Layer

Whenever a program requests a service from the operating system's kernel, it has to use the system call interface. Such services can, for instance, be accessing the file system, accessing network facilities or asking for the creation and execution of a new process. Thus, the system call interface is a noticeable observation point to monitor process behavior. Commercial off-the-shelf (COTS) kernel monitoring frameworks, such as *auditd*[8] for Linux, and logman, that leverages the *Event Tracing Windows* (ETW) for Windows[9], easily enable the observation and recording of system calls. Considering the increasing interest in leveraging system call monitoring and logging for security purposes and the high performance overhead induced by this type of

---

5 https://modsecurity.org
6 https://docs.python.org/3/library/trace.html
7 https://docs.microsoft.com/en-us/windows-hardware/drivers
8 https://people.redhat.com/sgrubb/audit
9 https://docs.microsoft.com/en-us/windows-hardware/test/wpt/event-tracing-for-windows

observation mechanism, recent work focuses on the improvement of auditd architecture to lower its run-time and storage overheads [Ma et al., 2018].

System calls have been widely used to perform anomaly-based intrusion detection since the seminal work of Forrest in 1996 [Forrest et al., 1996]. This family of work models the legitimate behaviors of processes using sequences of system calls. The vast majority of approaches focuses on the study of the Linux kernel. Very few approaches propose to adapt these techniques to the Windows kernel. An example of such adaptation was proposed in [Creech, 2014].

System calls have also been used as an information flow tracking means to perform intrusion detection. These approaches are part of the policy-based IDS family. They consider any violation of the information flow security policy, defined by a cyber defense analyst, as an intrusion attempt. Implementations of such techniques are generally done on the Linux kernel by instrumenting it. They either leverage the Linux Security Module framework, such as Blare [Zimmermann et al., 2003] or BlueBox [Chari & Cheng, 2003], or add their own modules to the kernel.

Apart from the system call interface, many activities can be monitored at the operating system layer. For instance, OSSEC[10] and Samhain[11] have the capabilities to perform file integrity checking, log file analysis, and rootkit detection, among others.

### 1.4.3 Network Layer

Computer communications constitute a major field of monitoring systems. The monitored activity corresponds to packet exchanges inside the computer network. Considering the context of a company, these exchanges can happen inside an internal network or between the internal and external networks. A network security policy that explicitly allows or restricts exchanges is enforced using firewalls. Unfortunately, as we have already mentioned, it is now well-considered that prevention mechanisms will eventually fail. As computer systems to protect are likely to be connected to the Internet, network-oriented security monitoring is thereby mandatory.

So far, approaches that have been presented fall into the HIDS category. As threats can also come from the network, security monitoring also applies to the network layer. This discipline is called Network Security Monitoring (NSM), i.e., "the collection, analysis, and escalation of indications and warnings to detect and respond to intrusions" [Bejtlich, 2013]. In other words, NSM aims to find intruders on the monitored network and react before they attain their goal and damage the company. Examples of open-source NSM platforms are *Security Onion*[12] and *RockNSM*[13].

Regarding the network context, the different level of observation can be illustrated using Figure 1.2. For example, alerts only give a few information on punctual suspicious activities, according to the NIDS rule set. Netflow logs contain all connections, and some statistics such as the size of packet ex-

---

10 https://www.ossec.net
11 https://www.la-samhna.de
12 https://securityonion.net/
13 http://rocknsm.io/

Size

Packets' Payloads (Full Packet Capture)
×

Application Logs (Zeek)
×

Connections Stats (Netflow)
×

Alerts (NIDS)
×

Information Quantity
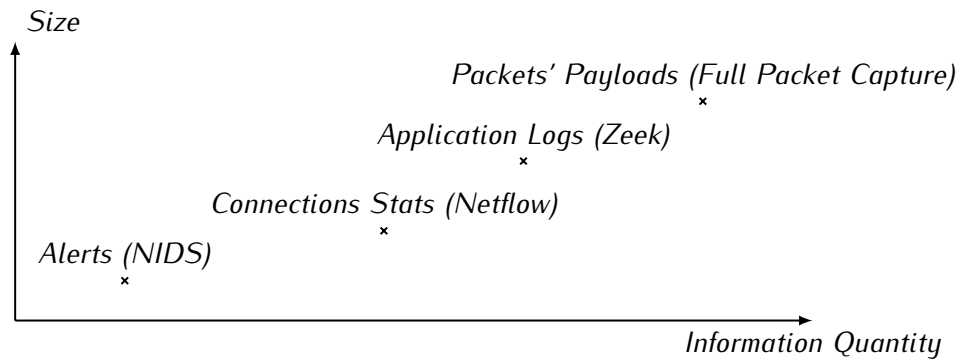
**Figure 1.2:** Detail levels of information acquisition according to the NSM tool used.

changes, among hosts of the monitored network. Application logs, obtained from protocol dissection, contain more detailed information about the states of the applications and their requests and responses. Finally, a full packet capture allows a cyber defense analyst to perform a complete network forensics analysis by deep diving into packets payload, e.g., extracting transferred files, or simply by allowing him to use all the analysis tools he wants. All the data presented in this figure are derived from the analysis of the packet streams in the monitored system. This analysis can be done either online, using monitoring tools, or offline, while replaying a full packet capture. Of course, the more the information is complete, e.g., full packet capture, the more storage capacity is needed.

Here again, these different levels of observation can be seen as layers of abstraction. According to the needs of investigation and the monitoring capacities, different types of analyses can be performed. In the best case, the whole traffic can be recorded using tools such as tcpdump. This recording process is called a full packet capture and allows cyber defense analysts to perform fine-grained network forensics using Wireshark[14] for instance.

NSM relies on many tools, especially on NIDS such as Snort[15] or Suricata[16]. According to the IDS taxonomy previously defined (Section 1.3), these tools are misuse-based IDSs. They perform deep packet inspection to retrieve patterns defined in their detection rule set. One of the most important tool of NSM is Zeek[17], which was formerly known as the Bro NIDS. Contrary to Snort and Suricata, whose primary goal is to highlight suspicious behaviors by triggering alerts when a signature matches, Zeek is about enabling the observation of network activity to have a better understanding of it. To do so, Zeek performs protocol dissection to generate logs out of the activity that is happening on the monitored network. Protocol dissection allows the retrieval of application level data. This enables network and cyber defense analysts to have a semantically high-level view of the activity occurring on their monitored network, e.g., which service is requested and what is requested.

---

14 https://www.wireshark.org
15 https://www.snort.org
16 https://suricata-ids.org
17 https://www.zeek.org

We have seen that NSM tools rely on the analysis of protocols and packet payload. Unfortunately, encrypted communication prevents these tools from analyzing the content of communications, leading NSM to be less effective. In this context, deep packet inspection techniques cannot be performed. Moreover, only source and destination IP addresses and ports and information such as the payload size will be available to the cyber defense analyst.

### 1.4.4 Hardware or Assembly Instruction Layer

Monitoring can also be performed at the hardware level. Intel Processor Trace (IntelPT)[18] is an example of such technology. IntelPT leverages dedicated hardware facilities to perform tracing of the processor, i.e., the recording of all executed instructions. It supports exact control flow tracing at the instruction level. Regarding computer security, data generated by IntelPT can be used to perform Control Flow Integrity [Abadi et al., 2009] or to create control flow-based behavioral models to perform anomaly detection [Chen et al., 2018]. Dynamic binary instrumentation, such as the `Pin tools` developed by Intel[19], can also be considered as a tracing technique. More specifically, this technique allows code insertion into a program at run-time, without the need to recompile the application. For example, such capability can be used to inspect each instruction executed by an application at the userspace level, collect run-time information, track function calls (i.e., control flow), track data flow, or perform program capture and replay.

———————————

At this point, we have seen that events can be generated at the different layers by various types of monitoring systems. The produced events can, in turn, be monitored and analyzed by active monitoring systems.

### 1.4.5 Event Monitoring—Log Analysis

As we have already mentioned, analyzing log data generated by a system allows monitoring teams to get different types of insight regarding the system itself, its status, its properties, or its security state. Event monitoring can apply to any abstraction layer where logs are produced. Various levels of complexity can be found among event monitors. Some simply scan log files, searching for known text patterns. For example, at the OS layer, a simple system call event monitor can be configured to raise alerts when confidential files are accessed. Others have the capability to correlate events. This is particularly useful when several events need to be considered to characterize an attack. To reason about them, events are parsed into structured data. For instance, events can be used to build graph structures or feature vectors that can, in turn, be used as inputs for other algorithms. The following paragraphs illustrate examples of the different ways of analyzing logs to give an idea of the possibilities of event monitoring and log analysis.

———————————

18 https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
19 https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation
-tool

*System Understanding*

**Discovering Infrastructure's Critical Services.** Regarding network, network devices' logging systems can produce logs such as Netflow[20]. Netflow logs can be analyzed and used to gain insight on the running infrastructure through the computation of a topological graph. For example, they can be used to discover the infrastructure's critical services and their dependency relationships inside a network [Zand et al., 2015]. This knowledge allows monitoring teams to prioritize their prohibitive and defensive actions.

**Understanding Control-Flow and Data-Flow.** Considering application logging systems, we have seen so far that log messages are the consequence of executed log printing statements in the code. Thus, log messages are generated according to the program's *control-flow*, i.e., execution path, as well as its *data-flow*. This insight has been leveraged in several works where the main idea is to extract knowledge out of the logs using mining methods.

For instance, SALSA [Tan et al., 2008] leverages log messages produced by a distributed system to derive approximated automaton views of the entire distributed system execution. Their goal is to reconstruct the distributed system's control-flow and data-flow for graph-based visualization and a better understanding of failures.

**Detecting Anomalies.** Some approaches leverage the knowledge extracted from logs to build a reference behavior model and perform anomaly detection, i.e., they monitor the stream of events while performing model checking to detect any deviation.

Similarly to SALSA, presented in the previous paragraph, the authors of "CloudSeer" [Yu et al., 2016] use totally ordered logs, also referred to as interleaved logs, to build a finite-state automaton (FSA) for each task of the distributed application. The built FSA captures temporal dependencies between log messages to model the task workflow. During the first offline modeling stage, each identified task, e.g., creating a virtual machine in an OpenStack[21] environment, is executed several times to generate several traces of the same task. These traces are then used to build the FSAs corresponding to tasks. Computed FSAs are then used in an online checking stage to identify whether the newly produced traces satisfy the FSA specification. Every deviation is considered as an anomaly. Another noticeable approach is DeepLog [Du et al., 2017], a state of the art anomaly detection method that uses interleaved logs. The authors' intuition is that logs produced by the execution of a program are really close to a natural language: log messages can be seen as "elements of a sequence that follows certain patterns and grammar rules." To do so, they leverage a deep neural network model based on long short-term memory [Hochreiter & Schmidhuber, 1997]. Contrary to CloudSeer, their method allows the capture of inter-task dependencies. Moreover, its anomaly detection models can be updated in an online manner.

---

20 https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow
21 https://www.openstack.org

*Attack Detection*

As we have seen before, attack investigation often relies on log analysis to understand what happened. Depending on the nature of the log analyzed, the analysis enables different attack detection capabilities. For instance, OrchIDS [Goubault-Larrecq & Olivain, 2008] and GnG [Totel et al., 2004] are event correlation engines that rely on attack description languages to correlate events explicitly. Cyber defense analysts can express their knowledge of attacks in these languages to write detection rules.

**Detecting Anomalies—Security Monitoring Perspective.** All the log analysis approaches presented in the previous "Detecting Anomalies" paragraph are based on interleaved logs. Their drawback relies on the fact that the authors consider that the different clocks are loosely synchronized across hosts, i.e., they assume the existence of a global clock in the distributed system and consider log traces as temporally totally ordered. However, other methods consider the fact that the existence of a global clock cannot be assumed in a distributed system. They leverage relationships such as Lamport's "happened-before" [Lamport, 1978] to build a partial order of the log messages issued from distributed logs. Similar data mining approaches can be performed on partial orders. For instance, temporal invariant mining algorithms have been proposed in [Beschastnikh et al., 2011].

In [Totel et al., 2016] and [Lanoë et al., 2019], the authors propose a log-based and anomaly-based IDS for distributed applications. However, contrary to DeepLog and CloudSeer, their methodology relies on the partial ordering of logs. Here again, any deviation from the learned behavior reference model is considered as the observation of an attack. The methodology developed relies on the log analysis of several legitimate executions of the distributed application to capture as many legitimate behaviors as possible. For each execution logs, the partial order is computed and used to derive an FSA model as well as likely temporal invariant, similarly to [Beschastnikh et al., 2011]. A first global behavior model is then built by merging all the automata and invariant properties. Finally, the behavior reference model is obtained by leveraging generalization algorithms, i.e., KTail algorithms family [Beschastnikh et al., 2014], on the global automaton. This model can then be used by the detection part of the methodology for global automaton checking and temporal invariant checking. Security-wise, these kinds of anomaly detection methodologies extract properties from the logs and use these properties to build the reference model. Any execution log which does not comply with this model is then considered as the observation of an intrusion.

**Leveraging a Single Type of Log.** Some approaches focus on the analysis of a single type of log to detect specific actions. For instance, Lamprakis et al. propose an approach to detect the communication of compromised hosts that try to reach their Command and Control (C&C) server through HTTP protocol in [Lamprakis et al., 2017]. To do so, the authors leverage the HTTP web proxy logs to build web request graphs, where a node corresponds to an HTTP request and its response and edges correspond to dependencies

between requests, a target node being the consequence of a source node. The web request graphs reconstruction allows filtering out regular web browsing, thus highlighting single nodes that would ideally correspond to malware requests.

**Leveraging Several Types of Log.** Other approaches propose to leverage several types of logs to detect attacks. One of the seminal work in this precise research field is the work of Abad et al. [Abad et al., 2003]. The authors highlight the fact that attacks can be reflected in different logs and are not obvious to detect when only a single log is analyzed. They argue the need to correlate heterogeneous logs to increase intrusion detection systems' accuracy. More specifically, several approaches propose to correlate network-related and system-related logs to improve intrusion detection [Abad et al., 2004] [Li et al., 2004] [Yurcik et al., 2006]. On the other hand, Beehive [Yen et al., 2013] correlates different types of network-related logs, mainly HTTP web proxy and DHCP logs, to detect suspicious hosts that might be compromised. DHCP logs allow the association of IP addresses to their corresponding hosts. A daily feature vector is then computed for each host based on the HTTP requests it made during the day. Based on the fact that hosts form homogeneous groups inside enterprises, a clustering-based outlier detection is then performed to identify suspicious hosts, i.e., the ones that behave differently from the group. These detections are considered as incidents that can be further analyzed to determine whether it corresponds to an attack or a policy violation made by an employee.

**Detecting Multi-Step Attacks with Heterogeneous Logs Analysis.** Very few approaches leverage heterogeneous logs to detect multi-step attack scenarios automatically. In HERCULE [Pei et al., 2016], the authors model multi-step attack scenario detection as a community discovery problem. Inspired from social networks where people with similar interests form communities, events are considered as individuals that are likely to be close to each other when generated by the same activity, i.e., events generated by attacker's actions are likely to be part to the same community. Starting from a given finite set of heterogeneous events, the approach consists in building a weighted correlated event graph, where nodes, edges, and weights respectively correspond to individual events, attribute-based correlations, and distance metrics. The built-weighted graph is then given as an input to a community discovery algorithm. In a final step, communities containing at least one suspicious event are tagged as malicious. The authors claim that detected communities contain the various steps that make up the attacks.

## 1.5 SUMMARY

Chapter 1 introduces the reader to the basic terminology that will be used throughout the rest of the manuscript, as well as the key concepts of security monitoring.

Briefly, Section 1.1 introduces the reader to the concepts of: *intrusions*, which correspond to sets of actions that attempt to compromise the confidentiality, integrity, or availability of a system; *multi-step attacks*, which refer to attack scenarios where attackers have to perform several steps in order to attain their goals; *cyber defense analysts*, i.e., the organizations' defenders, who are in charge of deploying and configuring the security monitoring systems at various abstraction levels in order to enable situational awareness; *events*, i.e., monitoring systems' outputs, which correspond to one of the major means for cyber defense analysts to perform attack detection and investigation; and, finally, *abstraction layers*, e.g., network, OS, or application layers, which correspond to the different possible observation points for monitoring.

Section 1.2 presents why monitoring is critical, especially in the computer security context. Based on the observation that prevention mechanisms are not sufficient, i.e., persistent attackers will eventually break into the system, security monitoring is mandatory to enable the observation, detection, recording, and collection of the actions performed by the attackers.

The two remaining sections of this chapter illustrate different types of monitoring systems. Section 1.3 focuses on the monitoring system class dedicated to attack detection, namely, intrusion detection systems. Briefly, IDSs can be categorized into the network-based category, i.e., NIDS, or the host-based category, i.e., HIDS, depending on the data source they analyze. Additionally, IDSs can be categorized into misuse-based and anomaly-based IDSs, depending on their analysis methodologies. Section 1.4 presents how monitoring can be performed in different abstraction layers, namely, the application, OS, network, and hardware layers.

**The Need for Alert and Event Correlation.** While performing attack investigation, cyber defense analysts' goal is to identify the attacker's traces and highlight their *links* in order to deduce the global strategy of the attack, and its consequences. Retrieving the attacker's traces is a difficult task as they are scattered across several machines, across heterogeneous and diverse log files, and potentially scattered across time. The increasing complexity of computer systems also makes it hard to identify relevant security information. Moreover, IDSs and observation means only enable the detection of single-step attacks. Therefore, discovering and detecting sophisticated attacks, i.e., multi-step attacks, is a hard problem. It involves finding correlation links among heterogeneous events. Automatic or semi-automatic tools are needed in order to detect multi-step attacks by linking the different observations, i.e., the ones related to single-step attacks, that are part of the same multi-step attack. This discipline forms a dedicated security research field called *Alert and Event Correlation*. The next chapter focuses on this precise research field.

# 2 | ALERT CORRELATION

The last chapter introduced the reader to the basic concepts of security monitoring. We have seen that these systems enable the observation of various types of actions performed at different abstraction layers of the monitored system. Ideally, these monitoring systems would record malicious actions happening inside the monitored system. However, recorded events need to be further analyzed to *discover and understand the different steps that make up attacks*. This is the role of *Alert Correlation*. Alert correlation is tightly linked to the security monitoring industry. Indeed, in the enterprise context, all the events generated by monitoring systems are generally collected and analyzed by a SIEM, which is in charge of the correlation process [Nicolett & Kavanagh, 2011].

This chapter introduces the reader to the alert correlation research field. Section 2.1 presents why alert correlation is critical regarding security monitoring: alert correlation makes up for IDSs' limitations and aggregate events to enable the discovery of multi-step attacks. Section 2.2 presents in more detail the alert correlation process and defines its surrounding concepts. More specifically, the different types of correlation (e.g., correlating alerts to vulnerability knowledge) used by alert correlation techniques will be presented. Section 2.3 focuses on the attack scenario identification component of the alert correlation process. Finally, Section 2.4 presents alert correlation's challenges and limitations. Furthermore, it discusses the relationship between alert correlation and causal dependency relationships among events. More specifically, we observed that the definition of the causal dependency relationship among events has not been clearly defined in the literature. We argue that addressing this issue would help the research community to formalize the alert correlation research field and enable the discovery of new types of approaches.

## 2.1 ADDRESSING INTRUSION DETECTION SYSTEMS' LIMITATIONS

The alert correlation research field began in the early 2000s, when researchers and security monitoring vendors realized the IDSs' limitations [Debar & Wespi, 2001] [Cuppens & Miege, 2002]. Alert correlation aims to address the major limitations of IDSs presented in the following paragraphs, namely, alert flooding, false positive and false negatives, presented in Section 2.1.1, and the aggregation of alerts and/or events corresponding to the same attack, presented in Section 2.1.2.

### 2.1.1 Flooding, False Positives and False Negatives

The first limitation of IDSs is called *alert flooding*: too many alerts are produced by IDSs, and cyber defense analysts cannot keep up the pace. In practice, a lot of alerts are related to the same problem or the same occurrence of an attack, but the lack of *context* prevents them from being easily logically grouped. Moreover, alerts only describe symptom-related or problem-related information. They generally do not contain explicit information about the root causes of a problem. On top of this semantic issue, many of the produced alerts correspond to false positives [Axelsson, 1999], meaning that an alert was triggered even though the attack attempt was unsuccessful or that the triggering action was a benign behavior irregularity. For example, false positives can be the consequences of an off-the-shelf detection rule set that does not suit the monitored system. False negative, i.e., the fact that a successful attack is not detected, is still an issue as well. In order to address these limitations, the alert correlation discipline emerged. Thus, alert correlation's first objectives have been:

1. Alert verification, i.e., false positive checking, to filter out irrelevant alerts;
2. Merging of alerts corresponding to the same attack occurrence and;
3. Alert prioritization, also called "alert triaging," to focus on most crucial ones [Porras et al., 2002].

These objectives comprehensively tackle the issue of "alert fatigue", also called "information overload problem," where cyber defense analysts miss alerts relating to actual attacks as: (1) true positives generally have a low priority; (2) cyber defense analysts are drowned in the noise of false positives [Hassan et al., 2019].

### 2.1.2 Aggregating Events Corresponding to the Same Attack

As systems become more and more complex, the diversity of data sources also increases. The amount of information to process motivates another objective: the need to merge information from different data sources to translate alerts into more understandable and exploitable information [Dain & Cunningham, 2002]. Such capabilities would ideally enable cyber defense analysts to gain analysis time and perform more efficient responses. The alert correlation process thereby integrates information from other data sources than IDS alerts, such as events generated by other monitoring systems, for instance. Alert correlation thereby includes event correlation [Dain & Cunningham, 2001]. Here again, the amount of events generated is important, and events often lack the information regarding the context in which they happen. As an example, a legitimate event such as the recording of an successful admin login might be malicious in a specific context. Another objective is thereby added to alert correlation's: increasing attack investigation insight by identifying and aggregating alerts and events, which have a low detail level, corresponding to the same attack. Meta-alerts [Valeur et al., 2004], also called hyper-alerts [Ning et al., 2002], are produced when aggregated alerts correspond to symptoms of actual attacks. Doing so, the overall number of produced alerts can be reduced, and their details can be

enhanced, thus elevating their semantic level and making them more exploitable and understandable for cyber defense analysts.

––––––––––

Section 2.1 presented IDSs' limitations and introduced how alert correlation tries to tackle them. Alert correlation aims to help cyber defense analysts to find "needles in the haystack." More specifically, its goal is to identify significant events and their links by leveraging aggregation techniques, i.e., alert correlation identifies the "threads among needles in the haystack." Following sections present in more detail the process of alert correlation.

## 2.2 ALERT CORRELATION DEFINITION

In their seminal work *Alarm Correlation* [Jakobson & Weissman, 1993], Jacobson and Weissmann defined network alarm correlation as a "conceptual interpretation of multiple alarms such that new meanings are assigned to them." Their goal was to emphasize the fact that such techniques could enable network administrators to improve network surveillance and fault management. In [Gardner & Harle, 1996], Gardner and Harle defined it as the "interpretation of multiple alarms so as to increase the semantic information content associated with a reduced set of messages." Although this work focused on telecommunication network alarms, these definitions still hold in today's security monitoring field.

### *Correlation Meaning*

So far, we have introduced alert correlation without defining what *Correlation* means. Let's first define what it is.

**Definition 2.1  - *Correlation:*** *"An action to carry back relations with each other", taken from the report Alert Correlation: Review of the State of the Art [Pouget & Dacier, 2003].*

In facts, alert correlation can be performed in several ways: among events; among observed or known states of the monitored system; among system information such as its topology and its vulnerabilities in the case of networks; among alerts; and, of course between all of these categories at the same time. The precise definition of correlation thereby depends on the technique used for a given alert correlation process.

The two following paragraphs propose two ways of seeing the alert correlation process. The first one describes a functional point of view of the alert correlation process (Section 2.2.1). The second one describes the different types of information alerts are correlated to in the literature (Section 2.2.2).

### 2.2.1  The Alert Correlation Process

This section introduces the reader to its functional architecture. The different approaches proposed [Valeur et al., 2004] [Sadoddin & Ghorbani, 2006] [Salah et al., 2013] mainly converged to the same representation.
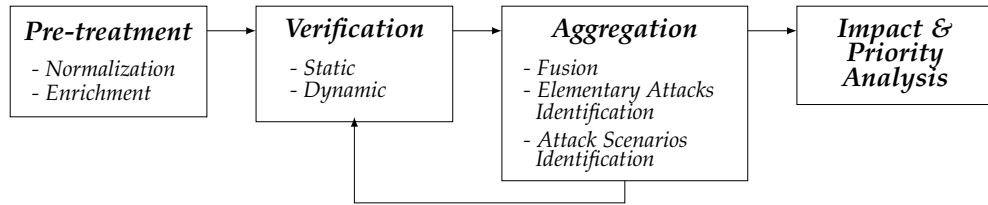
**Figure 2.1:** Alert correlation functional architecture.

More specifically, four families of components can be highlighted, namely, *pre-treatment*, *verification*, *aggregation*, and *impact and priority analysis* [Godefroy, 2016]. Figure 2.1 illustrates the global functional architecture of alert correlation. These components will be presented in the following paragraphs.

The dotted arrow from *aggregation* to *verification*, in Figure 2.1, corresponds to a feedback loop. Salah et al. have proposed this functional architecture for meta-alerts to be enriched and verified again [Salah et al., 2013]. An example of such a feedback loop is [Stroeh et al., 2013], where alerts are aggregated using clustering techniques. A supervised learning classifier then verifies alert clusters.

### Pre-treatment

The first step of the alert correlation process consists in a pre-treatment phase where alerts are *normalized* and *enriched* to enable them to be analyzed by the other components. This phase is also called *preprocessing* in [Salah et al., 2013].

**Normalization.** As we have seen in Chapter 1, alerts are generated by various IDSs that monitor different abstraction layers and use different conventions and formats. Normalization aims to unify alert syntax, i.e., attribute structure, and semantics, i.e., attributes meaning. Expressing the alerts in such a unified format enables the other components of the architecture to compare them.

The Intrusion Detection Message Exchange Format (IDMEF) has been defined to unify alert syntax [IDM, 2007]. Several approaches relies on IDMEF [Zaraska, 2003] [Liu et al., 2008] and efforts to promote it are still ongoing [IDM, 2016]. For instance, an IDMEF output module for ModSecurity has recently been proposed to feed the Prelude SIEM [Baláž et al., 2018]. On the other hand, fewer approaches pay attention to normalizing alerts semantics. In [Liu et al., 2008], the approach includes a semantics checking module that verifies attributes coherence. Another approach proposes to leverage a taxonomy to extend IDMEF with semantics normalization [Stroeh et al., 2013]. Syntactic and semantics normalization are implicitly assumed in the literature. The different approaches often focus on given components of the whole alert correlation process.

**Enrichment.** In addition to alert syntactic and semantics gaps, the information they contained might not be enough to perform better correlations. Several approaches propose to enrich alerts to overcome this issue.

This enrichment is generally done by leveraging static knowledge databases. These databases are first populated before being queried to give information alerts are missing, e.g., information regarding targets, sources, or IDS that produce it [Debar & Wespi, 2001] [Morin et al., 2009] [Sadighian et al., 2013]. Similarly, the authors of [Mustapha et al., 2012] propose to enrich alerts using information collected from honeypots. Alerts can also be enriched semantically to enable an aggregation component to group alerts corresponding to the same action [Sundaramurthy et al., 2011].

### Verification

As we have already mentioned in Subsection 2.1.1, IDSs are subject to the false positive problem. The verification component aims at recognizing and discarding non-relevant alerts in the monitored system context. This component corresponds to the *alert reduction module* in [Salah et al., 2013].

Following the taxonomy from [Kruegel et al., 2004], verification can be either static or dynamic. Dynamic approaches monitor observable consequences of attacks to determine whether they succeeded or not [Kruegel et al., 2004]. On the other hand, the main approach of static verification is the vulnerability knowledge correlation. These approaches discard alerts that refer to the exploitation of a non-existent vulnerability in the monitored system [Chyssler et al., 2004] [Liu et al., 2008]. Here again, static verification might rely on knowledge databases such as M4D4 [Morin et al., 2009] or ONTIDS [Sadighian et al., 2013]. Another type of approach proposes to generate IDS verification rules automatically by leveraging supervised machine learning [Massicotte et al., 2008]. Such rules can, in turn, feed knowledge databases.

### Aggregation

Aggregation components aim at reducing the overall number of alerts by grouping them into meta-alerts. They can be classified into three categories, namely, *fusion*, *elementary attacks identification*, and *attack scenario identification*.

**Fusion.** Fusion components are responsible for recognizing and combining alerts of the same type, which represent the "independent detection of the same attack instance by different intrusion detection systems" [Valeur et al., 2004]. This definition can be extended with the combination of alerts representing the detection of the same attack instance by the same IDS. Fusion can be further refined into three cases [Morin, 2004]:

1. *Split* - Combining alerts, generated by the same IDS, which correspond to the same attack instance. In other words, the alerts are caused by the same action, e.g., a port scan that makes a NIDS trigger several alerts. The split case is closely related to elementary attacks identification component.

2. *Recurrence* - Combining alerts, generated by the same IDS, which correspond to a recurrent attack. Fusion components that deal with recurrence are similar to the *attack thread reconstruction* component proposed in the functional architecture of Valeur et al. [Valeur et al., 2004]. The

mentioned case example is an attacker that runs his exploit several times to guess the parameters' correct values, e.g., memory addresses and offsets for buffer overflows. Another interpretation of recurrence is proposed in [Viinikka et al., 2009]. The authors rely on time series to identify and discard periodic alert floods. They hypothesize that these periodic phenomenons correspond to benign behavior.

3. *Redundancy* - Combining alerts, generated by various IDSs of the same type, which correspond to the same attack instance [Debar & Wespi, 2001]. Here again, the alerts are caused by the same action.

Alert fusion techniques generally rely on similarities between alerts' attributes [Julisch, 2003] [Valdes & Skinner, 2001]. The idea is that alerts that have similar attributes and are generated in a given time frame likely have the same root cause.

**Elementary Attacks Identification.** Elementary attacks identification components are closely related to fusion components. They aim to identify and combine alerts, generated by different types of IDSs, that correspond to the same attack. These components are similar to the *attack session reconstruction* component proposed in the functional architecture of Valeur et al. [Valeur et al., 2004]. For instance, in [Chyssler et al., 2004], the authors leverage Snort NIDS alerts, Samhain HIDS alerts, Syslog events, and machine learning techniques to perform elementary attacks identification.

**Attack Scenario Identification.** All the outputs of the previously presented components can, in turn, be analyzed by the attack scenario identification components to identify those pertaining to multi-step attack scenarios. Attack scenarios identification is the main focus of the work presented in this thesis. Related techniques will be presented in Section 2.3.

*Impact and Priority Analysis*

In addition to alert aggregation, alert correlation also aims at prioritizing the information that needs to be processed by cyber defense analysts. To do so, the alert correlation process also includes an impact and priority analysis component. Alerts and meta-alerts are ranked depending on criteria such as IDS confidence [Pérez et al., 2014] or potential impact of corresponding attacks [Porras et al., 2002].

### 2.2.2 Alert Correlation Types

In order to achieve its goals, alert correlation leverage many other sources of information to enhance its correlation capabilities. These data sources can be considered as a taxonomy of alert correlation types [Shittu, 2016] and have been described and summarized in several approaches [Salah et al., 2013] [Gagnon et al., 2009]. The alert correlation types, as in [Shittu, 2016], are the following:

*Correlating Alerts with Topology, Cartography and Detection Capabilities*

Topology and cartography information aim to provide an accurate representation of the monitored system. Topology information describes the positions and links of nodes. It typically includes network-related information such as IP addresses and their associated names, subnets, or virtual LANs. Cartography information describes each nodes' components, i.e., its users, OS, software, processes, services, and their version information [Morin, 2004] [Gagnon et al., 2009]. Such databases can be populated using agents that dynamically gather information [Yu et al., 2004] [Chyssler et al., 2004]. Topology and cartography knowledge have been used early to enhance alert correlation capabilities [Goldman et al., 2001]. For example, it can be used to perform alert verification, e.g., by discarding alerts corresponding to attacks against software that are not part of the cartography. Such knowledge has also been leveraged to perform alert prioritization in [Porras et al., 2002]. As we will describe in the next paragraph, this knowledge can be coupled to vulnerability databases to perform a risk assessment of the monitored system.

Detection capability corresponds to the knowledge of the deployed IDSs' characteristics such as their nature, i.e., application-based, host-based or network-based, their topological visibility or their ability to detect attacks in given conditions for example [Morin et al., 2002] [Morin, 2004]. Coupled with topology and cartography knowledge, detection capability knowledge enables the semi-automatic generation of correlation rules to detect complex attack scenarios [Godefroy et al., 2015b]. This approach will be presented in more detail in the Section 2.3 on attack scenario identification techniques.

*Correlating Alerts with Vulnerability Knowledge*

Vulnerability assessment plays an important role in risk analysis. Such knowledge allows security monitoring teams to elaborate their strategy regarding security, and more specifically, regarding prevention and detection mechanisms. In the context of intrusion detection and alert correlation, alerts can be symptoms of attacks that rely on the exploitation of specific vulnerabilities. Considering these kinds of alerts, the monitored system's vulnerability information allows filtering out false alerts regarding the monitored system's context [Kruegel et al., 2004].

Knowing the monitored system's configuration information, e.g., its topology and cartography, cyber defense analysts can identify disclosed vulnerabilities, i.e., known vulnerabilities described in vulnerability databases, using vulnerability scanner technology. Such inspection allows a cyber defense analyst to know whether a target is safe against the security flaws used by a given attack. It thereby allows to filter out alerts corresponding to failed attack attempts that leverage exploits of non-existent vulnerabilities in the monitored system context.

Vulnerability databases such as MITRE's Common Vulnerabilities and Exposures (CVE)[1] list and National Vulnerability Database[2] provide common

---

1 https://cve.mitre.org
2 https://nvd.nist.gov

identifiers for publicly known vulnerabilities, i.e., disclosed vulnerabilities affecting public software.

Coupled with topology and cartography knowledge, vulnerability knowledge can also be used to build attack graphs [Sheyner et al., 2002]. The next section will describe how attack graphs can be used to generate correlation rules automatically.

---

The correlation types presented in the two previous paragraphs, namely, correlation of alerts with topology, cartography, and detection capabilities, and correlation of alerts with vulnerability knowledge, are partially and, sometimes, fully included in databases called *knowledge databases* in the alert correlation research field. These technologies bring the monitoring environment context that alerts and, more generally, events, are missing.

Knowledge can be organized using technologies such as ontologies [Sadighian et al., 2013] or other frameworks such as M4D4 [Morin et al., 2009]. To the best of our knowledge, these work correspond to state of the art knowledge databases for alert correlation. They include all the alert correlation types previously described. More specifically, M4D4 has been extended to enable the semi-automatic generation of event correlation rules [Godefroy et al., 2015a].

### Correlating Alerts with Alerts

As described in [Kent & Souppaya, 2006], approaches that leverage this type of correlation "match multiple log entries from a single source or multiple sources based on logged values, such as timestamps, IP addresses, and event types." For example, De Alvarenga et al. aggregate NIDS alerts that have the same source IP address and consider that they are part of the same attack scenario in [de Alvarenga et al., 2018]. Similarity-based correlation, statistical methods [Julisch & Dacier, 2002] [Gu et al., 2008] or visualization tools [Leichtnam et al., 2017] fall into this alert correlation type.

### Correlating Alerts to Attack Knowledge

As its name suggests, attack knowledge includes any knowledge related to attacks. Such knowledge comes from information security experts and is expressed in various forms, i.e., as a taxonomy of attacks, or as the expression of dreaded attack scenarios.

Attack knowledge can be expressed as a characterization of attacks, generally in the form of taxonomy, that allows answering questions such as:

- What are the attack targets?
- What are the main observation means to record the actions related to this attack?
- What is the attack class?

In his PhD thesis [Godefroy, 2016], Godefroy presents the different attack knowledge taxonomies which can be classified into five classes:

1. *Violation type*. This taxonomy includes classes such as vulnerability exploitation, security policy violation, or suspicious activity. Contrary

to the other taxonomies, the violation type attack taxonomy has a fairly high level of abstraction.

2. *Technique used*. This taxonomy generally corresponds to the main type of classification. It allows the description of the different steps and techniques used to perform an attack. Some approaches propose a simple list of attack classes [Porras et al., 2002]. The ATT&CK knowledge database proposes to classify techniques by attack tactics, e.g., initial access, persistence, privilege escalation, lateral movement, command and control or exfiltration. The Common Attack Pattern Enumeration and Classification (CAPEC[3]) list presents techniques as attack patterns. The classification is done in a tree format where a root node describes a generic attack class (e.g., injection attacks), intermediate nodes refine the attack class, and leaf nodes describes techniques (e.g., severity level, required steps to perform the attack, required privileges, examples of traces left by such attack).

3. *Attacker's required privileges or attack's prerequisites*. This taxonomy corresponds to a classification of the required privileges needed by the attacker to perform an attack. It allows cyber defense analysts to determine whether a given attack is possible or not, according to the system to protect, and to assess its probability to happen. Privileges can correspond either to the attacker's location, i.e., distant access (from the internet), local access to the internal network or physical access, or its user's privileges, i.e., simple user or root. For each technique presented in their databases, ATT&CK and CAPEC also describe the permissions required to perform the technique.

4. *Attack's consequences*. This taxonomy corresponds to a classification of the consequences of attacks. Consequences correspond to the attack impact on the system's integrity, confidentiality, and availability. These types of taxonomies generally include obtained privileges through a privilege escalation, e.g., gaining root access, leaked information on the system, e.g., files, users, or topology, or services' performance degradation.

5. *Attack's target*. This taxonomy corresponds to a classification of the attack's target, i.e., a network device, an OS, an application, a service, or memory.

The reader may refer to Godefroy's PhD thesis [Godefroy, 2016] to get more details on attack taxonomies. Such taxonomies, especially the attack's prerequisites and consequences, allow leveraging alert correlation techniques such as *prerequisite and postcondition* approaches, which is well illustrated by the lambda language [Cuppens & Ortalo, 2000]. Coupled with detection capability knowledge, attack knowledge taxonomies can be leveraged to identify attacks that can actually be detected in the context of the monitored system.

---

3 https://capec.mitre.org

**Attack Knowledge from the Attacker's Perspective.** Attack knowledge can also refer to dreaded attack scenarios description from the attacker's perspective. This type of approach only deals with the attack modeling, i.e., the attacker's actions. They do not deal with events, nor their correlations. This type of knowledge can be found in various forms. For instance, attack trees and attack graphs, presented in Section 1.1.1, respectively express attack scenarios and the set of possible attack paths according to the topology, cartography, and vulnerability knowledge databases.

**Attack Knowledge from the Defender's Perspective.** Contrary to the previous type of approach, the type of approach presented in this paragraph does not deal with the attacker's actions. This type deals with events, alerts, and their correlations. It thereby represents the modeling of attack knowledge with the defender's perspective. Attack scenarios description is done using attack description languages such as [Eckmann et al., 2002] [Totel et al., 2004] [Goubault-Larrecq & Olivain, 2008]. Such languages allow to explicitly correlate events to elaborate and express event correlation rules. These rules thereby represent attacks through their traces, i.e., the events that represent the observed and recorded actions. These rules are later interpreted by a correlation engine that analyzes events and retains those described in the correlation rules. This type of knowledge is also referred to as case-based knowledge [Salah et al., 2013] [Navarro et al., 2018].

———————

The common idea behind all these types of correlation is to enhance the context of alert investigation to make it more efficient. They can be found in the different components of the alert correlation functional architecture presented in Section 2.2.1. Proposed approaches generally focus on some components of the alert correlation process and do not address all of them at once. A majority of them focus on the problem of attack scenario identification. Section 2.3 presents how the alert correlation research community proposed to solve this problem.

## 2.3 A FOCUS ON ATTACK SCENARIO IDENTIFICATION

Attack scenarios identification has been thoroughly studied in the alert correlation research field. Leveraging the different kinds of available knowledge (e.g., the knowledge presented in the alert correlation types in Section 2.2.2), and produced data (i.e., alerts and events generated by the monitoring systems), attack scenario identification aims at helping cyber defense analysts to identify and retrieve the alerts and events corresponding to the same multi-step attack scenarios.

The problem of attack scenario identification in alert correlation has been addressed using various kinds of methodologies and several taxonomies have been proposed to categorize them [Yusof et al., 2008] [Salah et al., 2013] [Godefroy, 2016]. More specifically, these various methodologies compute correlation links between events and alerts. According to the used methodology, attack scenarios can be either predefined, reconstructed, or discovered.
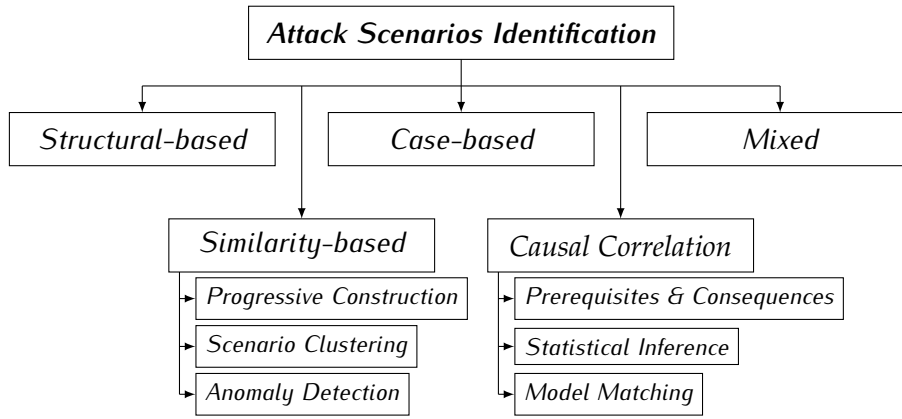
**Figure 2.2:** Attack scenario identification taxonomy.

Presented approaches will be presented using Navarro et al.'s taxonomy, as previously proposed taxonomies are included in their work [Navarro et al., 2018]. Figure 2.2 illustrates the different categories of the taxonomy, namely, *similarity-based*, *causal correlation*, *structural-based*, and *case-based* techniques. Each category will be presented in the following paragraphs.

### 2.3.1 Similarity-based Approaches

Similarity-based approaches are described as follows in [Navarro et al., 2018]: "The degree of similarity between traces determines the construction of the attack scenarios." The idea is that similar alerts are likely to be related to the same root cause, i.e., to the same attack scenario [Salah et al., 2013]. Contrary to causal correlation techniques, which focus on alert and event sequences and their causal structure, similarity-based techniques focus on the computation of a similarity degree between alerts and events. The similarity degree is generally computed after a combination of several event record fields.

The main advantage of similarity-based techniques is that they are likely to return unknown attack scenarios. However, choosing the right linking process is still a hard task. Simple linking processes are subject to the generation of false positive correlations. On the other hand, complex linking processes, e.g., using an alert correlation matrix [Zhu & Ghorbani, 2006], might be too specific to capture the different kinds of multi-step attacks.

Similarity-based methodologies are divided into three groups: progressive construction (by attribute matching or correlation), scenario clustering, and anomaly detection.

1. **Progressive construction** - Potential attack sequences are built step by step by appending traces to scenarios that contain similar traces, according to their record fields. Built sequences follow a logical progression where the order of the actions matters. Thus, a time window is considered when comparing traces' similarities. Sequence building methods are divided into two subcategories:

    a) *attribute matching*, which relies on exact comparison of chosen record fields, e.g., network-related fields such as IP addresses

and/or ports. For instance, in [Chen et al., 2006], the authors instrumented the Bro NIDS to actively correlate network events coming from the same IP address than a given alert.

b) *attribute correlation*, which computes a correlation coefficient among events and links them if a threshold is attained. Such correlation coefficient can, for example, be computed by leveraging partial matching of record fields [Dain & Cunningham, 2001], an alert correlation matrix [Zhu & Ghorbani, 2006] and/or supervised learning algorithm [Zhu & Ghorbani, 2006] [Pei et al., 2016]. In [Zhu & Ghorbani, 2006], the authors mentioned that the usage of their alert correlation matrix technique, which corresponds to a "knowledge base that encodes statistical correlation information of alerts," allows the inference of causal relationships when the correlation coefficient between two alerts is high enough.

2. **Scenario clustering** - As its name suggests, this category includes methodologies that identify groups of similar actions, considered as potential multi-step attack scenarios, using clustering algorithms. As it is mentioned in [Navarro et al., 2018], "The degree of similarity between traces belonging to the same scenario should be higher than the degree between traces from different scenarios."

3. **Anomaly detection** - Similarly to the anomaly detection presented in the intrusion detection section (Section 1.3.2), these methodologies learn the normal behavior, corresponding to normal sequences of events, and consider any deviating sequence as an attack scenario. Hidden Markov Models (HMM) are often used to perform such anomaly detection as they model well sequences of events.

### 2.3.2 Causal Correlation Approaches

Navarro et al. describe causal correlation approaches as follows in [Navarro et al., 2018]: Scenario reconstruction "is focused on the anatomy of multi-step sequences and the causal relationship between their steps. In other words, previous steps determine the ones that follow, and a causal scheme can be derived from this relationship." The name of this category is interesting as it merges the notions of causality and correlation, although they are different.

According to Navarro et al.'s [Navarro et al., 2018] and Salah et al.'s [Salah et al., 2013] studies, the main advantages of causal correlation techniques are their ability to: highlight sequences of steps, which make results easily interpretable by human analysts; discover slight variations of known attacks; potentially uncover the causal relationship between events and alerts. Causal correlation techniques are also said to be subject to false positives.

Causal correlation methodologies are divided into three groups: prerequisites and consequences, statistical inference, or model matching.

1. **Prerequisites and consequences** - Methods from this subcategory rely on the alert to attack knowledge correlation type previously described

in Section 2.2.2. They associate a set of prerequisites, also called pre-conditions, and a set of consequences, also called post-conditions, to each alert [Benferhat et al., 2003]. The goal is to automatically correlate the post-conditions of an alert to the preconditions of another one to build sequences of potential multi-step attacks. Thus, attack scenarios are also progressively constructed in this type of methodology. The LAMBDA language [Cuppens & Ortalo, 2000] and the JIGSAW language [Templeton & Levitt, 2001] correspond to the references of pre-requisites and consequences approaches. Even if this approach seems promising, it is unfortunately impossible to precisely encode and enu-merate all attack prerequisites and consequences into alert pre/post-conditions.

2. **Statistical inference** - Methods from this subcategory learn a statis-tical model from a training dataset, based on the frequencies of at-tack actions and their sequences, that can, in turn, be used to detect and/or predict attacks. Most popular statistical inference methods for attack scenario identification are based on HMMs and Bayesian infer-ence. The work proposed in [Ren et al., 2010] is an example of a sta-tistical inference approach that leverages a Bayesian network to extract causal relationships among alerts and predict future steps of ongoing detected attacks.

3. **Model matching** - "Methods doing model matching assume that every multi-step attack follows a certain structure. They model this structure and try to find sequences that adapt to them" [Navarro et al., 2018]. Contrary to case-based methods, which rely on the expression of spe-cific attack scenarios, model matching methods operate at a higher level of abstraction, e.g., a model can represent the skeleton of an at-tack scenario, such as the different stages of an Advanced Persistent Threat [Luh et al., 2017]. Navarro et al. mentioned approaches that rely on the formalism of HMM to model the abstract stage sequences. Model matching methods mainly leverage alert to attack knowledge correlation type and alert to vulnerability knowledge correlation type.

### 2.3.3 Structural–based Approaches

Navarro et al. describe structural-based approaches as follows in [Navarro et al., 2018]: "Incoming traces are projected to a model of the network, where future attack paths can be predicted." The word structural refers to the struc-ture of the defended network regarding its topology, cartography, and vul-nerabilities. Structural-based methods do not depend on attackers' actions. They only consider the latter information. They thereby only leverage alert to topology, cartography, and detection capability correlation type and alert to vulnerability knowledge correlation type.

Such structural information is generally coded in the form of an attack graph, i.e., an abstract representation of the network containing the moni-tored system's vulnerabilities in each node [Sheyner et al., 2002]. Incoming alerts are then projected into the attack graph to detect attack scenarios and

predict future attack steps. This projection can be performed using different kinds of correlation engines such as a dedicated attack graph engine [Noel et al., 2004] [Jajodia et al., 2005] or the usage of an attack description language and its related correlation engine [Lanoë et al., 2018].

Given that topology, cartography, and vulnerability knowledge are available, structural-based methods are easily deployed as attack graph models can be automatically generated. They thereby represent an easy way to compute alert correlation rules. However, they only contain known information about the monitored system. Structural-based methods cannot detect unknown attack scenarios or attack steps that do not leverage a known vulnerability.

### 2.3.4 Case–based Approaches

Navarro et al. define case-based approaches as follows in [Navarro et al., 2018]: "Detection of well-known attack scenarios as an ensemble of traces." Case-based approaches represent an important part of the alert correlation research field. The attack scenario knowledge-base can be either manually populated by cyber defense analysts or automatically populated by leveraging techniques that extract attack scenarios from datasets. As we have already mentioned before, the former method, i.e., manually describing attack scenarios through their projected events and alerts on the monitored system, is one of the most popular methodologies used in the SIEM industry. This category of approaches relies on attack description languages, e.g., STATL [Eckmann et al., 2002], OrchIDS [Goubault-Larrecq & Olivain, 2008], and ADeLe [Totel et al., 2004], and leverage the alert to attack knowledge correlation type and the topology, cartography and detection capability correlation type described in Section 2.2.2. It corresponds to the *predefined scenarios* category in Salah et al.'s taxonomy [Salah et al., 2013].

Case-based methods' main advantage is their low false positive rate. In fact, the exact occurrence of attack scenarios is searched in the alert and event streams. Only known attack scenarios included in the knowledge database can be detected. Unfortunately, most of the case-based methods are limited to IDS alerts and do not consider other events. Moreover, most approaches do not consider the reliability of monitoring systems. In fact, a given sensor might miss the detection or recording of a significant action regarding an attack scenario. The missing event or alert would prevent case-based approaches to identify attack scenarios included in the knowledge database. For instance, this limitation has been addressed in [Lanoë et al., 2018], where the authors propose a correlation engine that is able to detect incomplete attack scenarios.

Due to their resemblance, prerequisites and consequences-based, model matching-based, structural-based, and case-based can be wrongly considered as being the same methodologies. Therefore, it is important to highlight the fact that they are different. In prerequisites and consequences approaches, each step of an attack scenario is represented by an alert enriched by its corresponding prerequisites and consequences knowledge. Attack scenar-

ios' sequences are not defined. Model matching approaches define abstract views of attack scenarios where a given view corresponds to many attack scenario instances. Structural-based approaches leverage network-related information such as its topology and vulnerabilities to build attack graphs. Attack graphs represent all attack paths that coherently exploit known vulnerabilities, according to the monitored system. Finally, case-based are manually built and represent specific attacks.

### 2.3.5 Mixed Approaches

Navarro et al. define mixed approaches as follows in [Navarro et al., 2018]: "More than one of the approaches are followed but none of them stands out among the others." In fact, many of the approaches proposed in the attack scenario identification field leverage several techniques. A few examples of the mixed category are presented with the following approaches. For instance, HERCULE [Pei et al., 2016], presented in Section 1.4.5, has several processing stages that fall into different categories. Its first event processing stage falls into the attribute matching subcategory of the similarity-based's progressive construction category. A correlated event graph is built by applying a set Boolean logic rules, which compares event record fields, for each pair of the normalized event set. More specifically, nodes correspond to events, and edges correspond to the Boolean logic rules result, i.e., a binary vector where matching rules are set to 1 and the other to 0. This multi-dimension graph is then transformed into a flat weighted graph using a supervised learning algorithm. This second stage of HERCULE falls into the causal correlation's statistical inference. Finally, a clustering algorithm is applied to the weighted graph to group events corresponding to the same attack scenario. In [de Alvarenga et al., 2018], the authors identify attack scenarios in NIDS alert sets using process mining, i.e., an attribute matching technique, and hierarchical clustering, which correspond to a scenario clustering technique of the similarity-based category. The approach proposed in [Qin & Lee, 2004b] is an example of a prerequisites and consequences approach, coupled with statistical inference approaches. The authors propose to correlate alerts using two different techniques: a Bayesian-based technique that relies on the hypothesis that earlier attack steps positively affect later ones and the expert knowledge of alerts' prerequisites and consequences; a statistical analysis, which does not rely on expert knowledge, based on the hypothesis that alerts have temporal and statistical patterns.

---

Previous sections presented a holistic view of the alert correlation's definition, process, and technique diversity. So far, its challenges and limitations have not been exposed yet.

## 2.4 ALERT CORRELATION'S CHALLENGES AND LIMITATIONS

Section 2.4 discusses the challenges faced by security monitoring, and, more specifically, the challenges and limitations of alert correlation.

First of all, as we have seen at the end of the first chapter, computer systems are becoming more and more complex. Elaborating a monitoring strategy for such environments is the first challenge. For instance, monitoring systems' clocks have to be synchronized to analyze produced events conveniently. Otherwise, the chronological order of observed actions could be altered.

A second challenge is the detection of single-step attacks. In fact, individual steps composing a multi-step attack might be missed by deployed monitoring systems. This may be caused by deployment or configuration limitations, e.g., the lack of observation means for a given abstraction layer, or technical limitations, e.g., current detection capabilities [Chen et al., 2006].

Another challenge lies in the nature of multi-step attacks: the time interval between consecutive attack steps can be very high, i.e., hours, days, or even months [Chen et al., 2006].

The following paragraphs present in more detail the current limitations of alert correlation, namely, the fact that expressing dreaded attack scenarios is a difficult task and that proposed alert correlation approaches often exclusively rely on NIDS alerts. Then, the necessity to consider heterogeneous events to detect and understand better multi-step attacks is highlighted. Finally, we present our opinion regarding the relation between alert correlation and the concept of causal dependency relationships among events.

### 2.4.1 Expressing a Dreaded Scenario is Hard

As we have previously mentioned, the alert and event correlation research field is directly linked to the security monitoring industry, i.e., the SIEM industry. Even though many different kinds of approaches have been proposed, as it has been illustrated in the previous section, classical SIEM technologies apply rule-based correlation based on expert knowledge [Navarro et al., 2018], i.e., case-based approaches relying on alert and event correlation languages. In other words, current SIEM practices rely on the explicit expression of correlation links between alerts and events.

**On the Attacker's Perspectives.** In order to elaborate event correlation rules for attack scenario identification, cyber defense analysts need to think like attackers. This is already a challenge because being exhaustive in the listing of possible attack seems impossible. In fact, attackers can elaborate multiple attack plans in parallel and choose to continue or stop them according to their possibilities [Qin & Lee, 2004b]. They might not need to follow a precise order to perform their attack, which implies that possible action sequences can be complex. Moreover, attackers try to evade detection by performing actions that are incoherent with their previous actions.

**Merging the Defender's and the Attacker's Perspectives.** Detecting multi-step attack scenarios rely on the capability to infer the single-step attacks that make them up, together with the links between them. In practice, alert and event correlation rules are considerably *difficult* to formulate because cyber defense analysts have to combine the *attacker's perspective* by building possible attack scenarios and representing them with attack description languages, as described in Section 2.3.4; as well as the *defender's perspective* by having a precise knowledge of the system to defend, i.e., its topology, cartography, vulnerabilities, known attacks, and detection capabilities. The combination of these points of view enables them to project the attack steps based on the set of observable events that the deployed monitoring systems can produce; accordingly, they can explicitly frame the sequence of events that represents the attack.

Additionally to the efforts needed to write alert and event correlation rules, rules are static. Considering the fact that an organization's network can be dynamic, the fact that rules are static represents another limitation. Rules have thereby to be manually updated to correspond to the new state of the monitored network.

Moreover, according to Godefroy [Godefroy, 2016], the availability of correlation rules are taken for granted. Consequently, only a few research work focus on the different ways of writing correlation rules.

**Towards Automatic Correlation Rule Generation.** In order to address these issues, recent work aim to perform automatic alert and event correlation rule generation. For instance, the authors of [Godefroy et al., 2015a] propose to simplify the process of event correlation writing by dissociating: (1) the attack scenario specification from the knowledge of the monitored system; (2) leveraging a knowledge database, such as M4D4 for instance [Morin et al., 2009], which contains organized information on the monitored system, i.e., its topology, cartography, vulnerabilities, and deployed monitoring systems. Different teams of information cyber defense analysts can thereby focus on either of the points of view. Following their methodology, attack scenarios' specifications are then semi-automatically translated into explicit alert and event correlation rules according to the information contained in the knowledge base.

Another example of automatic correlation rule generation is the usage of attack graphs [Jajodia & Noel, 2010]. Using a similar knowledge database as the previous presented approach, an attack graph can be used to represent all possible observable attack paths according to the deployed monitoring systems in the monitored system. These observable attack paths can, in turn, be automatically translated into explicit alert and event correlation rules.

**Towards Automatic Attack Scenario Detection Methods.** Much hope is being placed in automatic attack scenario detection methods, i.e., automatically constructing attack scenarios from alerts [Ning & Xu, 2010]. Such methods would ideally reduce the time spent by cyber defense analysts in potential threats investigation and avoid human errors in the attack signatures development [Navarro et al., 2018]. Based on the fact that most dangerous attacks only happen rarely, datasets have a few examples of multi-step at-

tacks. Therefore, it is difficult to apply automatic attack scenario detection methods, which are based on learning techniques.

### 2.4.2 IDS Alerts' Exclusivity

**Simplifying the Translation of Dreaded Attack Scenarios.** In Section 2.4.1, we have seen that cyber defense analysts have to explicitly and specifically frame the sequence of events that represents an attack scenario. When this process is done through the expression of a dreaded scenario, the translation from the high abstraction level view to the alerts and events sequence that represents it is very difficult. This translation is often omitted, e.g., risk assessment approaches presented in Section 1.1.1, or simplified by only using NIDS alerts, for instance [Godefroy, 2016]. Attack graphs also illustrate that issue. The vast majority of attack graph-based approaches only consider alerts. More specifically, these alerts correspond to the monitored system's known vulnerabilities. As attack graphs rely on topology information, the considered alerts generally correspond to NIDS alerts.

**On Events' Informational Quality.** Alert correlation's efficiency greatly depends on the informational quality of produced events. In fact, "detection methods count only with a limited amount of information, such as the one contained in events or network packets, to infer attack scenarios." [Navarro et al., 2018]. For instance, IDS alerts generally do not contain explicit information about their triggering root cause [Salah et al., 2013].

Unfortunately, a majority of approaches only correlate alerts, particularly NIDS alerts. Navarro et al.'s study "A systematic survey on multi-step attack detection" illustrates it well [Navarro et al., 2018]. 85% of the 181 publications they reviewed propose approaches that exclusively leverage alerts. This high percentage could be explained by the lack of public datasets containing heterogeneous events.

**The Majority of Available Datasets Only Contain NIDS Alerts.** Obtaining datasets to evaluate proposed approaches is also difficult. Authors often rely on public datasets that contain poor heterogeneity of event types, i.e., they generally only contain NIDS alerts. As a matter of illustration, the vast majority of the approaches presented in Navarro et al.'s multi-step attack detection survey [Navarro et al., 2018] assess their approach on the famous DARPA 2000 dataset[4]. The fact that the famous dataset DARPA 2000 contains alerts exclusively might have greatly influence alert correlation research directions. In fact, many publications of this survey, i.e., 87 out of the 181, use the DARPA 2000 dataset for their experiments.

### 2.4.3 The Need for Heterogeneous Event Correlation

In fact, attackers attempt to be as stealthy as possible to avoid triggering IDS alerts. Moreover, as we have previously mentioned, attack scenarios are

---

4 https://www.ll.mit.edu/r-d/datasets/2000-darpa-intrusion-detection-scenario-specific-datasets

sequences of single-step attacks. Such single-step attacks might correspond to harmless steps or legitimate actions outside of the context of the attack. Thus, similarly to Bass [Bass, 2000] and Yusof et al. [Yusof et al., 2008], we argue that considering IDS alerts solely is not sufficient to conduct attack scenario detection. Single-step attacks can be observed by IDSs, general-purpose monitoring systems, or both. The last ones can provide critical contextual information to enable the understanding of the full picture of an attack scenario. Consequently, heterogeneous event correlation is needed.

Heterogeneity of data sources has been studied early in the alert and event correlation research field [Welz & Hutchison, 2001]. Valeur et al. already included such correlation capabilities in their functional architecture through the *attack session reconstruction* component [Valeur et al., 2004]. Its goal is to link network-based and related host-based alerts. Chyssler et al. proposed an alert and event correlation process that leverages Snort, Samhain, and Syslog events [Andersson et al., 2002]. In the same year 2004, Li et al's proposed UCLog [Li et al., 2004], which organizes audit logs from heterogeneous sources, i.e., system calls and network packet captures, to take advantage of their naturally existing correlation. In [Dreger et al., 2005] and [Almgren & Lindqvist, 2001], the authors propose to enhance NIDS events, e.g., Bro NIDS events, with host-based context. More specifically, they modified an Apache web server to enable the cooperation with the NIDS. In that sense, they propose to correlate network-based and application-based events.

Heterogeneous events correlation's benefits are multiple. First, it allows the reduction of alerts number that cyber defense analysts must address by grouping events that correspond to the same incident. Additionally, it can enhance detection capabilities by giving several points of view of the overall attack, thus allowing cyber defense analysts to get a more holistic view of the incident, and improving the detection confidence. This property is also called *adaptive scrutiny* [Dreger et al., 2005], i.e., heterogeneous monitoring systems provide complementary indications and coverage that allow enhancing the detection confidence. As we have previously mentioned, a priori legitimate events can be part of an attack scenario. Even though these events can seem innocuous, they can be included in the heterogeneous events analysis to determine whether they relate to an attack or not. Finally, heterogeneous monitoring systems also contribute to anti-evasion, as attackers will have to increase their efforts to evade all the different types of monitoring systems.

### 2.4.4 The Lack of a Clear Definition of the Links among Events

Leveraging data generated by diverse monitoring systems and IDSs deployed in several critical places of the monitored system would ideally make the explanation and detection of attack scenarios easier. As a reminder, a cyber defense analyst starts his investigation by building the set of events that are linked to a given event of interest, e.g., an IoC like an IDS alert. Accordingly, the cyber defense analyst attempts to identify links among the events to retrieve those that correspond to the traces of the attacker's actions in the

monitored system. Indoing so, he also attempts to reconstruct the attack scenario related to the event of interest. Previous work presented in the *attack scenario identification* section (Section 2.3) aims at building these connections among events and between attack steps and helping cyber defense analysts in their attack investigation task. In practice, this type of link is not trivial to define and discover, especially when considering heterogeneous events. Hence, there is a real necessity to formally describe and define the semantics of these links in the literature. Navarro et al. also emphasized this issue in their survey on multi-step attack detection [Navarro et al., 2018]. According to them, the diversity of proposed methods illustrates the fact that "the research community still does not know how to provide a solid definition of a link between the steps of an attack." Looking for correlations among events, our research led us to the notion of causal dependencies among heterogeneous events. We argue that the link definition our research community is looking for corresponds to a causal dependency among events.

**The Search for Causality.** As the discipline's name suggests, it is important to emphasize the fact that finding links between alerts and events relies on *correlation*, which is different from *causality*. In fact, correlations between alerts and events do not imply the existence of causal dependency relationships between them. However, we argue that attack scenario identification techniques *ideally* aim at identifying and highlighting actions performed by an attacker, as well as the cause–effect relationships among them, through alerts, events, and context analysis. Of course, the different correlation categories presented are more or less close to causal reasoning. This section aims to present how much the attack scenario identification technique categories are related to causal reasoning.

As it is, the relationship between the taxonomy proposed in [Navarro et al., 2018] and the concept of causality is not straight forward. Let's review all its categories from the perspective of causality.

**Similarity–based Methods and Causality.** Generally, similarity-based methods do not imply causal dependencies among events. However, even if this is not explicitly mentioned in their study, Navarro et al. seem to consider that attack scenario identification approaches leveraging information flow-based techniques at the operating system level falls into this category. Contrary to the other types of approaches in the similarity-based category, we argue that information flow-based techniques directly deal with causal dependencies. The next chapter will present information flow-based techniques in more detail.

**Causal Correlation Methods and Causality.** As the category's name suggests, the methods classified into the causal correlation category are really close to causal dependencies. When an alert's preconditions match another one's post-conditions, the two alerts are likely to be causally dependent, i.e., the related action of the first alert prepares for the conditions needed to perform the action that triggers the second alert. Statistical inference techniques aim to capture the implicit causal relationships between alerts and

events. For instance, HERCULE aims to infer causal dependency relationships among events that are part of the same activity, either malicious or benign, i.e., events pertaining to the same community should be causally linked [Pei et al., 2016]. Considering model matching techniques, two alerts with similar features that correspond to two consecutive attack stages are likely to be causally linked.

**Structural-based Methods and Causality.** Regarding structural-based methods, the attack paths included in an attack graph represent potential attack scenarios. Two alerts pertaining to a given attack path within a predetermined time window are likely to be causally dependent.

**Case-based Methods and Causality.** Regarding case-based methods, we have seen that attack description languages, or in other words, alert and event correlation languages, allow the precise and specific description of attack scenarios. There is thereby a close relationship between the knowledge of attack scenarios and causal reasoning: given an attack scenario they want to express, cyber defense analysts know the logical progressions of the attacker's actions. More specifically, they know the cause–effect relationships among them and can, therefore, transfer and express this knowledge into alert and event correlation rules that capture causal dependency relationships. Overall, leveraging attack knowledge allows attack scenario identification methods to highlight causal dependency relationships among events explicitly.

**Towards the Study of Causal Dependency Relationships in Computation.** In the previous paragraphs, we shared our opinion regarding alert correlation: we highlighted the fact that all the alert and event correlation techniques dedicated to attack scenario identification ideally aim to discover causal dependency relationships among events. In fact, apart from statistical inference, very few methods focus on the specific computation of causal dependencies among events in the alert correlation research field. From this observation, we guided our work towards the search of causality with the following idea in mind: if causal dependency relationships among events can be defined and computed, the discovery of attack scenarios translates to simple graph traversals. Such graphs correspond to causal dependency graphs with events as nodes and causal dependency relationships as directed edges. The next chapter presents how causal dependency relationships have been studied in other research fields such as the distributed systems, the information flow, and the provenance communities.

## 2.5 SUMMARY

Following the first chapter, which has introduced the reader to the concept of observation in the general and security monitoring contexts, Chapter 2 presents the alert correlation research field.

**Addressing Monitoring Systems' Limitations.** Briefly, alert correlation makes up for IDSs and monitoring systems limitations in the security monitoring context. It has several roles: solving alert flooding issues, reducing false positive and false negative rates, aggregating alerts or events corresponding to the same action, and above all, enabling the discovery and enhancing the understanding of complex attacks made up of multiple steps.

**Alert Correlation's Definition.** Delving deeper into the concepts surrounding alert correlation, we defined and presented the concept of correlation in Section 2.2. This section presents alert correlation functional architecture, which is made up of four components, namely, pre-treatment, verification, aggregation, and impact and priority analysis. These four components involve different types of correlation approaches. More specifically, alerts and, more generally, events, can be correlated to different types of information: descriptive information of the monitored system such as its topology, cartography, vulnerabilities and observation capabilities; attack knowledge; or any other events. Many approaches have been proposed, especially for attack scenario identification, which is part of the aggregation component. In particular, the main categories of attack scenario identification methods have been presented in Section 2.3.

**Alert Correlation's Limitations.** Following the global presentation of alert correlation, we have discussed alert correlation's limitations and highlighted the necessity to consider heterogeneous events, i.e., events emanating from different abstraction layers, to fully comprehend complex multi-step attack scenarios in Section 2.4. More specifically, we presented our opinion regarding the relationship between alert correlation and the concept of causal dependency relationships among events. Following the description of the various attack scenario identification method categories, we concluded that the community does not provide and converge to a clear definition of the semantics of the relationships among events. Our opinion is the following: *all the alert and event correlation techniques dedicated to attack scenario identification ideally aim to discover causal dependency relationships among events*. Thus, the relationships among events correspond to causal dependencies. The rationale is that defining these causal dependencies and enabling their computation would simplify the attack scenario identification process. Our research thereby led us to the study of causal dependencies computation among heterogeneous events in a broader context than security monitoring. The next chapter presents how causal dependency relationships have been studied in other research fields such as distributed systems, information flow, and provenance.

# 3 | CAUSAL DEPENDENCIES—IN THE SEARCH OF THE HOLY GRAIL

The last chapter introduced the reader to the alert correlation research field. Its different components have been presented and defined. In particular, we have paid close attention to one of its objectives: identifying multi-step attacks. Going through the alert correlation's state of the art, we highlighted and argued the fact that this objective relies on causal reasoning. Therefore, our research naturally led us to the study of *causality* in a broader context than security monitoring.

This chapter is dedicated to the study of causal dependency relationships in computation. Section 3.1 starts by laying the foundational concepts surrounding causality, namely, causes and effects, counterfactuality, and causal reasoning. Then, it introduces the reader to the particularity of causal dependency relationships regarding the context of a computer system: causal dependencies can be either internal or external. More specifically, internal causal dependencies can be perceived by the system, e.g., information flows among the system's entities. On the other hand, external causal dependencies happen in the physical world the computer system interacts with. In fact, external causal dependencies are hard to identify. Moreover, identifying internal causal dependencies seems to be already promising regarding the objective of discovering multi-step attack scenarios. Thus, we focused our research on internal causal dependencies. Our research led us to study causality in the distributed systems (Section 3.2), information flow (Section 3.3), and provenance (Section 3.4) research fields. More specifically, Section 3.2 and 3.3 present the two models our work is inspired from, i.e., Lamport's happened-before relationship and d'Ausbourg's causal dependency relationship, respectively.

## 3.1 CAUSALITY PRIMER

So far, we have studied attack investigation from the perspective of security monitoring. To attain their goal, cyber defense analysts seek to understand, explain, and diagnose the causes and effects of a suspicious event. Doing so, they are *reasoning about causality* and do the best they can to *infer* attack scenarios and their impact. In practice, their causal reasoning relies on the identification of causal dependencies among the heterogeneous events emanating from the monitored system.

The concept of causal dependence among events is difficult to define and, more generally, all the concepts surrounding causality may lead to confusion. This first section starts by introducing them.

The concept of *causality* has long been discussed in various disciplines, from philosophy, legal reasoning to computer science. Defining "what it means for an event[1] *A* to be an *actual cause* of an event *B*" is subtle.

### 3.1.1 Cause, Effect, Counterfactuality and Causal Dependency

Let's start by defining the notions of *cause* and *effect*. According to Oxford's English dictionary, their definitions are the following:

**Definition 3.1 - Cause:** *"A person or thing that gives rise to an action, phenomenon, or condition."*

**Definition 3.2 - Effect:** *"A change which is a result or consequence of an action or other cause."*

Many definitions of the concept of causality have been proposed. Its most basic definition is based on the concept of *counterfactual dependence*. Counterfactuality addresses the question "what if something had been different in the past" [Halpern, 2006] [Lewis, 2013]. Using this concept, causality is defined as the following:

**Definition 3.3 - Causality:** *"A is a cause of B if, had A not happened (this is the counterfactual condition, since A did in fact happen), then B would not have happened."* More formally, $(A \implies B) \iff (\overline{A} \implies \overline{B})$.

Although this definition is simple, we do not need a more complex and complete definition than the one based on counterfactuality in the context of our research. The reader may refer to Pearl's book "Causality: models, reasoning and inference" [Pearl, 2000] for more complete definitions of causality.

#### *Defining Causal Dependency Relationship*

In the previous chapter, we have already introduced the concept of *causal dependency* as the relation between cause and effect. However, as it is, the proposed definition of *causality* (Definition 3.3) seems to refer to the notion of *event* exclusively. Following the definition of *cause* (Definition 3.1), we can notice that the definition of cause applies to a larger context than events. The definition of causality thereby encompasses a larger scope, i.e., *A* and *B* do not have to be events, using the casual definition of "event," to be a cause or an effect.

Given a cause *A* and its effect *B*, *B* is said to be *causally dependent* on *A*. For example, if an event *E* needs an object *O* to be in a particular state to happen, then *E* is causally dependent on *O*. Thus, there are various kinds of causal dependency according to the nature of the causes and effects. This is well illustrated by the Open Provenance Model presented in Section 3.4.2. Different types of causal dependency relationships will be presented in the following sections and chapter.

---

1 In this sentence, the word *event* refers to its casual definition, i.e., anything that happens. It does not correspond to the definition of Chapter 1.

### 3.1.2 Causal Reasoning, Inference and Core Notions

In practice, reasoning about causality relies on underlying notions such as *time*: a cause always precedes its effect. In [Kayser & Lévy, 2009], Kayser et al. highlight these underlying notions which they call *core notions*. The following list describes the ones we are interested in considering our research context:

**TIME** – It can be viewed in two ways: (1) as an order, i.e., the cause precedes the effect; (2) as a metric, i.e., given a presumed cause, the fact that the effect happens too long after might be a hint that the effect has another cause.

**COUNTERFACTUALITY** – As we have already mentioned, many causality definitions are based on counterfactuality.

**CORRELATION** – "The repetition of A followed by B is a strong hint of a causal relation between A and B."

**NECESSARY CONNECTION** – "When a correlation is non accidental, it witnesses a causal relation, and we have a clear intuition of what is accidental and what is not."

**COHERENCE** – "Detecting incoherence among a set of observations is a strong incentive to challenge an alleged causal relation. Furthermore, a causal analysis is required when there is incoherence between our observations and our expectations."

Core notions are non causal concepts. However, they are needed to define and understand causality. More specifically, they are the tools needed to *reason* about causality and draw conclusions about the analyzed situations.

#### Causal Reasoning and Causal Inference

The "simple" exercise of defining the concept of causality is interesting. It *caused* us to reason about the notions of cause and effect. This analysis corresponds to the concept of *causal reasoning*. More specifically, reasoning about causality enables *causal inference*, i.e., drawing conclusions about the relation between the cause(s) and effect(s) by leveraging our knowledge. Causal reasoning and causal inference are thereby strongly linked.

Delving deeper into causal reasoning, we can observe that it relies on three types of reasoning, namely the deductive, inductive, and abductive reasonings. These different types of reasoning have been studied for a long time regarding the history and philosophy of science. More specifically, the fact that deductive and inductive reasonings lead to different knowledge claims is well settled [Herley & Van Oorschot, 2017]. Let's define these different types of reasoning.

**DEDUCTIVE REASONING** – Deriving *logical* conclusions from premises known or assumed to be true, i.e., a self-consistent set of axioms. One of the famous examples of deductive reasoning is the following:

$$\frac{\text{All men are mortal (1}^{\text{st}}\text{ axiom)} \quad \text{Socrates is a man (2}^{\text{nd}}\text{ axiom)}}{\text{Therefore, Socrates is mortal (conclusion)}}$$

**INDUCTIVE REASONING** – "Drawing conclusions about the empirical world using observations and inferences from those observations" [Herley & Van Oorschot, 2017]. It is important to emphasize that the observations used to infer conclusions might not be representative of the empirical world. For example, one can inductively infer that "all swans are white" after observing $n$ white swans:

$$\frac{\text{Swan}_1\text{ is white} \quad \text{Swan}_2\text{ is white} \quad ... \quad \text{Swan}_n\text{ is white}}{\text{Therefore, all swans are white}}$$

However, inferred rules from inductive reasoning can always turn out to be wrong when encountering an observation that violates the rule. For example, the observation of a black swan makes the inductive inference rule "all swans are white" wrong.

**ABDUCTIVE REASONING** – "Abduction, or inference to the best explanation, is a form of inference that goes from data describing something to a hypothesis that best explains or accounts for the data" [Josephson & Josephson, 1996]. Abduction follows the following kind of pattern:

$$\frac{\begin{array}{c}\text{D is a collection of data (facts, observations, givens)}\\ \text{Hypothesis H explains D (would, if true, explain D)}\\ \text{No other hypothesis can explain D as well as H does}\end{array}}{\text{Therefore, H is probably true}}$$

The research work presented in this thesis relies on the deductive and abductive reasonings.

### Causality Purposes

We apply causal reasoning in everyday life for many different purposes. Kayser et al. highlighted them when presenting their causal reasoning insights in [Kayser & Lévy, 2009]. For instance, causality can be sought when trying to make sense of a situation or to predict its outcome. The following list illustrates some of the purposes of causality that are interesting in the context of our research:

**GOAL–DIRECTED REASONING** – Setting up a sequence of actions to reach an objective. For instance, counter-acting the identified attacker's plan.

**SENSE–MAKING** – Applying abductive reasoning on the observed effects to presume their causes;

**DIAGNOSTIC** – "Causal reasoning enables: (1) circumscribing the possible faulty components when a system behaves improperly; (2) finding a sequence of tests to determine what has to be done." ;

**EXPLANATION** – "Filling some holes in the prior knowledge of the addressee." Two scenarios can be considered: (1) given that the addressee knows

that *A* causes *B*, why *A* is true has to be explained. This generally implies the need for a transitive chain of causal relations; (2) given that the addressee knows that *A* is true, the fact that *A* causes *B* has to be explained;

**PREDICTION** – Anticipating the future or imagining the effects of a decision. Decision-making thereby strongly relies on causal reasoning.

These different purposes can be closely interrelated. In fact, sense-making can be seen as the hidden goal of diagnostic and explanation. Moreover, explanation supports diagnostic. In the context of our research, explanation, sense-making, and diagnostic are the main purposes a cyber defense analyst is using when investigation an attack. For instance, when investigating an alert, the verification process relies on sense-making by checking the veracity of the alert according to the monitored system context. The analyst can then try to retrieve the causal chain of the attacker's actions through alert correlation approaches. This involves the explanation causality purpose. Goal-directed reasoning and prediction apply when the cyber defense analyst identifies the actions needed to recover from an attack and assesses the impact of these actions on the production environment.

### 3.1.3 Causal Dependency Relationships in Computer Systems

*Internal and External Causality*

So far, we have presented the concept of causality, and causal dependencies, in a general context. However, it is necessary to further refine it to properly make sense of it in computer systems, and in the context of our research. In fact, we can consider that there are two types of causality [Baquero & Preguiça, 2016]:

**INTERNAL CAUSALITY** – Causal dependencies that can be perceived inside the monitored system, e.g., information flows among information containers or message exchanges inside a distributed system.

**EXTERNAL CAUSALITY** – Causal dependencies that cannot be perceived inside the monitored system. Indeed, the monitored system interacts with the physical world (e.g., users interact with it), and there are also causal dependencies that cannot be perceived inside the system.

The example presented in [Baquero & Preguiça, 2016] by Baquero et al. illustrates well external causality. Let's consider a couple planning a night out using a system that manages reservations for restaurants and movies. One person makes the reservation for dinner and calls the second one to let her know. Following the phone call, the second person goes to the same system and makes the reservation for a movie. The first reservation actually caused the second one. However, the system has no way of knowing that. It is thereby an external causality and can only be approximated by physical time. Indeed, time can, at best, only totally order all events, whether related or not. It cannot substitute causality.

Regarding our research context, multi-step attack scenarios are made up of both internal and external causality. As an illustration of external causality in the context of a multi-steps attack scenario, let's consider an attacker that already gained a foothold inside the monitoring network. The attacker could gain knowledge about its target through data collection (e.g., shared network drives) and perform its next attack step based on this knowledge. These two attack steps are causally dependent, the first one being the cause of the second one. However, the monitored system cannot perceive it.

### Causal Dependencies Computation Strategies

Causal dependencies computation strategies naturally depend on the considered type of causality. As the goal of our research is to help the cyber defense analyst by providing him with the set of events that are causally dependent on a suspicious event (e.g., an alert or an indicator of compromise), we are especially interested in the explanation and diagnostic causality purposes. More specifically, we are interested in *tracking* causal dependencies among heterogeneous events. First of all, to achieve this goal, causal dependencies have to be identified. Computation strategies can mainly be categorized into two groups: (1) discovering causal dependency relationships knowledge through data analysis; and (2) explicitly highlighting causal dependencies.

**Discovering causal dependency relationships knowledge through data analysis.** This category leverages data analysis techniques such as statistical techniques, i.e., techniques from the statistical inference subcategory of "causal correlation" approaches presented in Section 2.3.2.

Used techniques include Bayesian networks [Ren et al., 2010], or Granger causality when data is modeled as time series [Qin & Lee, 2004b]. Considering attack scenario identification, such techniques have mainly been applied to the analysis of NIDS alerts in the alert correlation literature. The advantage of this category is that it enables the discovery of both internal and external causality. However, obtaining a relevant dataset is hard, especially when considering datasets made up of heterogeneous events.

**Explicitly highlighting causal dependencies.** Case-based approaches well illustrate this category. In fact, such an approach can capture the external and internal causality relationships among events, as described in the paragraph "The Search for Causality" in Section 2.4.4.

Similarly to case-based approaches that explicitly correlate events using event correlation rules, distributed systems' logical clocks approaches (presented in Section 3.2), and information flow approaches (presented in Section 3.3) allow to track explicit causal dependencies. However, contrary to case-based approaches that express specific dreaded attack scenarios through attack description languages, these approaches directly describe causal dependencies among entities of the monitored system. Thus, they enable the tracking of causality throughout the system. Unfortunately, this tracking is limited to internal causality only.

The work presented in this manuscript mainly focuses on internal causality and approaches that explicitly highlight causal dependencies. The following sections introduce the reader to causal dependency computation in the distributed systems, information flow, and provenance research fields.

---

This first section presented the different concepts surrounding causality and causal dependency. More specifically, we introduced *counterfactuality*, the core notions underlying causality, the deductive, inductive, and abductive reasonings that support causal reasoning, the various purposes of causality, and the specificities of causal dependencies in the context of computer systems. Following sections present how the concept of causality has been studied in different research fields. The next section is dedicated to distributed systems.

## 3.2 TEMPORAL CAUSALITY IN DISTRIBUTED SYSTEMS

In 1978, Lamport published "Time, Clocks, and the Ordering of Events in a Distributed System," a seminal work in the field of distributed computing theory [Lamport, 1978]. 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 of the lack of a global clock in the system.

Beyond the study of the specific temporal properties of distributed systems, Lamport's work has been the stepping stone to the study of causality in distributed computation. As we are interested in finding causal dependencies among events, our research naturally led us to study the concept of causality in the distributed systems research field.

This section presents the research field of distributed systems with the perspective of causal reasoning. It starts by defining what distributed computation is. Then it presents why causality is needed for distributed computation. Finally, it introduces the reader to Lamport's happened-before relationship and the different concepts surrounding the notion of logical clocks, also called logical time.

### 3.2.1 Defining Distributed Computation

The research field of *distributed computation* was born out of the interest of researchers and practitioners to "take into account the intrinsic characteristic of physically distributed systems" [Raynal, 2013], as well as the need to understand how to design, realize and test them [Schwarz & Mattern, 1994]. In his book "Distributed Algorithms for Message-Passing Systems," Raynal defines *distributed computation* as the following [Raynal, 2013]:

**Definition 3.4** *"**Distributed computation** arises when one has to solve a problem in terms of distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) such that each entity has only partial knowledge of the many parameters involved in the problem that has to be solved."*

Distributed computation can occur either in a close location (e.g., with concurrent processes running on a single machine, or several nodes inside a data-center), or different locations spread across the globe (e.g., geographically distributed databases).

### 3.2.2 Why is Causality Needed for Distributed Computation?

"These days hardly any service can claim not to have some form of distributed algorithm at its core" [Baquero & Preguiça, 2016]. Distributed systems are everywhere and their capabilities naturally rely on the quality of the distributed algorithms that run on them. In fact, the concept of causal dependencies among actions is fundamental to the design of distributed algorithms. It helps to solve various problems in distributed systems such as [Raynal, 2013]:

- The design of distributed algorithms, e.g., maintaining consistency in replicated databases or detecting deadlocks;
- The tracking of dependent events which can be leveraged to model the behavior of a distributed system and perform anomaly detection, as previously described in Section 1.4.5;
- The measure of progress of other processes to perform garbage collection of obsolete information or detect termination for example.

These few examples illustrate how important causality is in the distributed systems research field.

### 3.2.3 Temporal Causality Relationship—Lamport's Happened–Before Relationship among Actions

In order to study distributed systems, the research community converged to a simple, yet sufficient model to reason about causality in distributed systems.

*Distributed Systems' Model*

A distributed computation is considered as a collection of single-threaded processes that: (1) can solely communicate through message exchanges; (2) are disjoint, i.e., processes do not share a common memory. Each process produces a sequence of totally ordered actions determined by a local algorithm. Actions can be either: sending a message; receiving a message; or simply an internal action, e.g., a function call. In a distributed system, the message transmission might suffer from a non-zero delay; First in first out order is not assumed for message delivery; The availability of perfectly synchronized local clocks, i.e., a global clock, is not assumed. Generally, only relevant actions are considered. "The concurrent and coordinated execution of all local algorithms forms a distributed computation" [Schwarz & Mattern, 1994].

*Happened-Before Relationship and Causality*

In order to reason about the ordering of actions in distributed computation, Lamport defined a relationship called *happened-before*. This relationship, denoted by "$\prec$," enables the construction of a partial order on the set of actions that are performed by the concurrent running processes. It is defined as follows: Given two distinct 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 $m$ 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 $a \nprec b$ and $b \nprec a$. 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.

Lamport's relationship is often referred to as the *causality relationship*, as Schwarz and Mattern do in their paper "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail" [Schwarz & Mattern, 1994]. In fact, $a \prec b$ means that it is *possible* that action $b$ is causally dependent on action $a$, $a$ being any action that satisfies $a \prec b$.
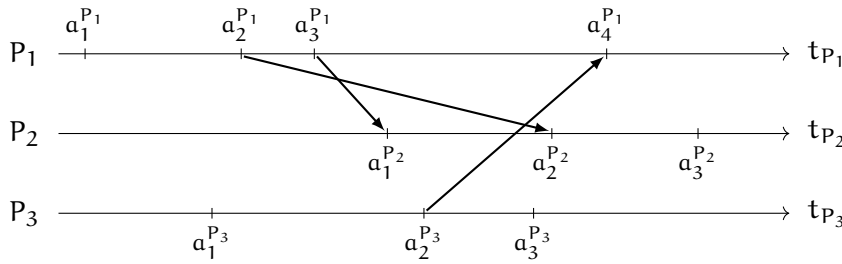


**Figure** 3.1: Space-time diagram of a distributed computation made up of three processes.

A distributed computation can conveniently be visualized with *space-time diagrams*. Figure 3.1 shows a simple example of distributed computation with three processes. Each process has its own timeline, i.e., with its own local clock, depicted, as well as the sequential actions it performs. Messages are represented by arrows connecting send actions to their corresponding receive actions. In this example, we can visualize the fact that $a_1^{P_1}$ may causally affect all the following actions of $P_1$, as well as all the actions of $P_2$. On the other hand, $a_2^{P_2}$ cannot causally affect $P_3$'s actions.

### 3.2.4 Distributed Systems and Logical Clocks

*Causal History and Causality Tracking*

Thanks to the happened-before relationship, the tracking of causal dependencies among actions can be easily done using *causal histories*. Each action $a$ can be assigned to its causal history $H(a)$, a set containing all actions which

causally precede $a$ [Schwarz & Mattern, 1994]. More specifically, given A the set of actions of a distributed computation, $H(a) = \{a' \in A | (a' \prec a)\} \cup \{a\}$.
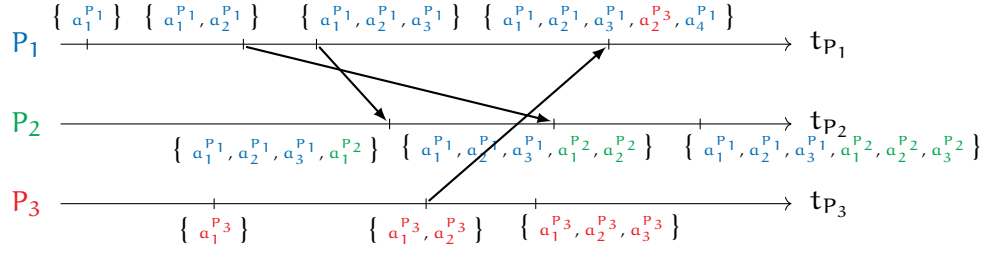


**Figure 3.2:** Causal histories of a distributed computation made up of three processes.

Figure 3.2 represents the same example of distributed computation and illustrates the causal histories of each action. Each process locally keeps its causal histories. Each time a process performs a send action, it also sends the causal history of the send action. For instance, the causal history of $a_1^{P_2}$ corresponds to the merging of the causal history of the send action $a_3^{P_1}$ with the local history, which corresponds to $\{a_1^{P_2}\}$. Causality tracking is straightforward, thanks to causal histories. Indeed, two actions $a$ and $b$ are said to be causally linked when $a \in H(b)$ or $b \in H(a)$.

The simple concept of causal history can already be leveraged to perform useful tasks such as intrusion detection. In fact, some of the approaches presented in the Event Monitoring section (Section 1.4.5) rely on causal histories to build behavior models [Lanoë et al., 2019].

As it is, causal histories are not very compact as their size is of the order of the total number of actions that occur during the distributed computation. The following paragraphs present how they can be represented with more compact representations, i.e., logical time or logical clocks [Raynal, 2013].

### Linear Time

Linear time, also called scalar time, has been proposed by Lamport [Lamport, 1978]. Each action $a$ is associated to a logical date $date(a)$, which is consistent with the happened-before relationship, i.e., $(a \prec b) \implies (date(a) < date(b))$. Without additional information on the logical date system, we cannot assure that $(date(a) < date(b)) \implies (a \prec b)$. The logical date system presented in this paragraph, i.e., the linear time, illustrates this remark.

The simplest logical date that respects the causality relationship among actions is the sequence of increasing integers. Thus, a logical date is an integer. This logical date system is called *linear time*. The following algorithm describes an example, using pseudocode, of the local algorithm for process $P_i$:

```
When producing an internal action a do:
    t_Pi = t_Pi + 1
    Perform action a

When sending a message MSG(m) to Pj do:
    t_Pi = t_Pi + 1
    send MSG(m, t_Pi)
```

```
When receiving a message MSG(m, t) do:
      t_{P_i} = max(t_{P_i}, t)
      t_{P_i} = t_{P_i} + 1
```

Several properties follow directly from the definition of linear time:

(1) $(date(a) \leqslant date(b)) \implies (b \nprec a)$;

(2) $(date(a) = date(b)) \implies (a \| b)$.
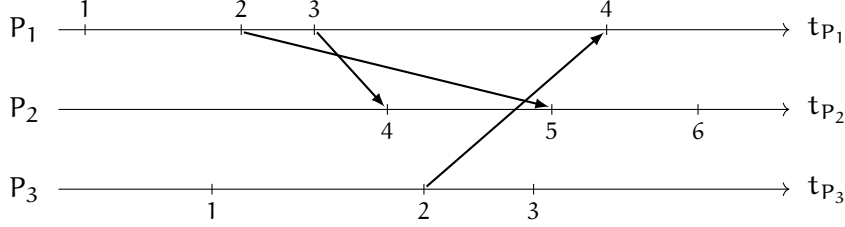


**Figure 3.3:** Example of a linear time system.

Figure 3.3 shows the same example with a linear time system. It illustrates very well that it is possible to have $(date(a) < date(b)) \land (a \nprec b)$. In this example, we have $date(a_1^{P_3}) = 1$, $date(a_2^{P_1}) = 2$ and $(a_1^{P_3} \nprec a_2^{P_1})$. Linear time is thereby not sufficient to capture causal dependencies among actions.

### Vector Clocks

Vector clocks have been developed independently by Fidge [Fidge, 1988], Mattern [Mattern, 1989]. The system is based on the following observation on causal histories: given an action $a^{P_i}$, if $a^{P_i} \in H(a^{P_j})$, then all actions performed by $P_i$ that precede $a^{P_i}$ are included in $H(a^{P_j})$, i.e., $H(a^{P_i}) \subset H(a^{P_j})$. It is thereby sufficient to store the most recent action from each process.
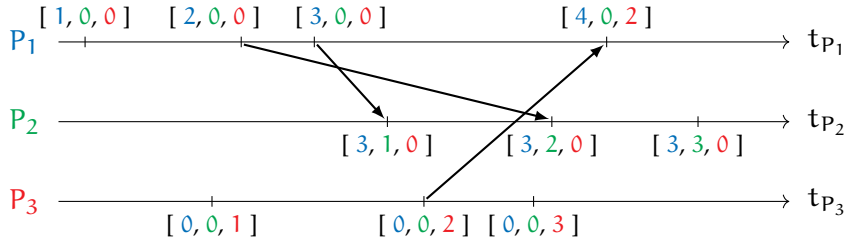


**Figure 3.4:** Example of a vector clocks system.

Figure 3.4 shows the same example with a vector clocks system. In our example, the causal history $H(a_2^{P_2}) = \{ a_1^{P_1}, a_2^{P_1}, a_3^{P_1}, a_1^{P_2}, a_2^{P_2} \}$ can be compacted to the following map: $\{ P_1 : a_3^{P_1}, P_2 : a_2^{P_2}, P_3 : 0 \}$. Leveraging this map, vector clocks rely on the observation that each process represented in the map can be sufficiently characterized by its largest index. Thus, $H(a_2^{P_2})$ can be further compacted to the following vector clock: $[ 3, 2, 0 ]$.

More specifically, given a distributed computation with $n$ processes, each action $a^{P_i}$, produced by the process $P_i$, is associated to a vector clock $V(a^{P_i})$ of size $n$. Given $k \in [1..n]$, $V(a^{P_i})[k]$ corresponds to $P_i$'s knowledge about the number of relevant actions performed by $P_k$.

Several properties follow directly from the definition of vector clocks:
(1) $(H(a) \subset H(b)) \iff (V(a)[k] \leqslant V(b)[k]$ for all $k \in [1..n])$;
(2) $(a \prec b) \iff (V(a) \neq V(b)) \wedge (V(a)[k] \leqslant V(b)[k]$ for all $k \in [1..n])$. Similarly to the actions, that can be partially ordered with Lamport's happened-before relationship, vector clocks can be partially ordered with the relationship denoted "<," which is defined as the following: $(V(a) < V(b)) \iff (V(a) \neq V(b)) \wedge (V(a)[k] \leqslant V(b)[k]$ for all $k \in [1..n])$. Thus, we have: $(a \prec b) \iff (V(a) < V(b))$. Contrary to linear time, vector clocks fully capture the causality implied by the happened-before relationship. In other words, given two actions $a$ and $b$, implementing such logical date system allows to easily verify whether $b$ might be causally dependent or not by comparing their vector clocks $V(a)$ and $V(b)$. The following algorithm describes an example, using pseudocode, of the local algorithm for process $P_i$:

```
When producing an internal action an do:
      V(an) = V(an−1)
      V(an)[i] = V(an)[i] + 1
      Perform action an

When producing a send action an to Pj with a message MSG(m) do:
      V(an) = V(an−1)
      V(an)[i] = V(an)[i] + 1
      send MSG(m, V(an))

When producing a receive action an with the message MSG(m, V(b)) do:
      V(an) = V(an−1)
      V(an)[k] = max(V(an)[k], V(b)[k])
      V(an)[i] = V(an)[i] + 1
```

### Matrix Clocks

More complex clocks, such as matrix clocks, have also been proposed. Matrix clocks represent a generalization of vector clocks: "while vector time captures *first-order* knowledge (a process knows that another process has issued some number of events), matrix time captures *second-order* knowledge (a process $P_i$ knows that another process $P_j$ knows that...)" [Raynal, 2013]. The reader may refer to the previous reference, i.e., Raynal's book "Distributed Algorithms for Message-Passing Systems," to find more details about matrix clocks.

————————

This section introduced the reader to the concept of causality in the context of distributed systems. All the work presented in this section relies on the partial order relationship *happened-before* defined by Lamport [Lamport, 1978]. The section starts with the presentation of the common modeling of distributed systems, describes the need to study the causal dependency relationship among actions performed in a distributed computation, and finishes with the presentation of proposed solutions for causality tracking (e.g., logical clocks and vector clocks). The next section is dedicated to the study of causality in the information flow research field.

## 3.3 INFORMATION FLOW–BASED CAUSALITY

One of the main goals of computer systems consists in the *automatic processing of data*, i.e., the storage, manipulation, and protection of information. The French translation of computer science, i.e., *informatique*, captures well this goal. Indeed, its etymology corresponds to the association of the French word *information*, which has the same meaning in English, and the suffix *—ique*, which means "which is proper to." More specifically, computer systems store information inside *information containers*, e.g., variables, files, sockets, processes, databases. Processes have the ability to perform actions to manipulate the information contained in these containers by transforming, or transferring it from one container to another one. As the information is transferred from one container to another, information is flowing. This corresponds to the concept of *information flow*.

### 3.3.1 Information and Computer Security—A Brief History of Security Mechanisms

As we have already seen in Chapter 1, the relationship between information and security is clear: information cannot be separated from the notions of *confidentiality* and *integrity*. In order to protect the information, different mechanisms have been proposed. For instance, *access control* protects the access of data objects by assigning a security level to them. Each security level is associated with explicit and mandatory permissions that allow the reading or modification of the objects assigned to it. The limitation of access control lies in the fact that it only prevents unauthorized users to *directly* read or modify files they don't have permission to access. In fact, they might be able to access it *indirectly* by "collaborating in arbitrarily ingenious ways with other users who have authority to access the information" [Denning, 1976].

*Information Flow Control* (IFC), also called *Information Flow Tracking* (IFT), addresses this limitation. Information flow control mechanisms monitor the dissemination of information throughout the monitored system to detect the ones that violate the predefined information security policy. IFC is thereby a means to implement policy-based IDSs, as described in Section 1.3.2. It forms a full research domain since Denning's seminal work, "A lattice model of secure information flow" in 1976 [Denning, 1976].

In order to implement security policies related to information, the information flow control community developed various kinds of information flow monitors and tracking techniques. Tracking is often implemented as a *label propagation policy* [Myers & Myers, 1999], also called *tag propagation policy* [Hossain et al., 2017] or *taint propagation policy* [Clause et al., 2007] [Balliu et al., 2017].

The great advantage of information flow monitors is that they can generally be used in a broader context than IFC. More specifically, tracking information flows enables the explanation and understanding of how a given information container came to be in a given state. This type of knowledge serves several purposes. For instance, information flows can be used to analyze and detect malware [Yin et al., 2007]. Another example is the usage of

information flows to retrieve the entry point of an attack [Newsome & Song, 2005]. In other words, information flows can also be leveraged in security monitoring. The term IFT is thereby more suited.

### 3.3.2 Information Flow and Causality—Introducing d'Ausbourg's Causal Dependency Relationship among Object States

***Information Containers' Dependency***

In the previous paragraph, we have mentioned that IFT enables the explanation of how a given information container came to be in a given state. More specifically, it captures the *causal chain of actions* that explains the information container's state.

In fact, information flows are produced by specific actions in the system. Therefore, these actions represent the cause of information flows. Then, an information flow implies that information is flowing from a source information container to a destination container. Consequently, these information container states are causally dependent as the value of the destination is directly dependent on the value of the source. This type of causal reasoning falls into deductive reasoning.

***Formalizing the Causal Dependency Relationship***

The relationship among information containers states has been especially formalized by d'Ausbourg [d'Ausbourg, 1994]. More specifically, he describes the relationship of causal dependency, denoted by "$\rightarrow$," among the states of the system. The system corresponds to a set of objects. A state, denoted $(o, t)$, is the value of an object o at a given time t. Using d'Aubourg's terminology, an information container, therefore, corresponds to an object. Formally, 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 $(o, t')$: states are causally dependent if and only if an *information flow* occurs among these states. The concept of an object is considerably flexible. For example, it can be variables in a program execution or system process states, file states, or socket states.

The model we defined in our work [Xosanavongsa et al., 2019a] is partly inspired from d'Ausbourg's causal dependency relationship among object states. It will be presented in the second part of this manuscript.

### 3.3.3 Information Flow Monitoring in Different Abstraction Layers

Information flow monitors have been implemented for many abstraction layers of computer systems. These abstraction layers contain various kinds of information containers, which in turn have different granularity levels. The fact that IFT has been studied in programming languages, operating systems, or databases, is a good illustration of how active this research field is.

Information flows are represented as directed graphs where nodes and edges respectively correspond to information containers and information flows.

### Programming Languages—Source Code Instrumentation

Information flows can be traced inside a process or a group of processes. Considered information containers correspond to the program's variables, inputs, and outputs, e.g., manipulated file descriptors. Information flows among them are produced by the programming language instructions, i.e., there is a flow of information from the variable x to the variable y if y's value depends on x's value. This dependence can be either direct or indirect. When it is direct, there exists a data flow between x and y. This type of dependence is called *explicit information flow*. Example 3.1 shows a snippet of Python code that illustrates this case.

**Example 3.1** *Illustration of explicit information flow in Python*

```python
x = 1
y = 1 + x
```

The dependence is said to be indirect when the value of a variable is affected through a control dependence. This type of dependence is called *implicit information flow*. Example 3.2 shows a snippet of Python code that illustrates this case. The value of x is affected by a through a control dependence.

**Example 3.2** *Illustration of implicit information flow in Python*

```python
def foo (a: int):
    if (a > 10):
        x = 1
    else:
        x = 2
    print(x)
```

Information flow monitors have been implemented for various programming languages. For instance, JBlare [Hiet et al., 2008] and JFlow [Myers & Myers, 1999] monitor information flows in the Java programming language. JSgraph [Li et al., 2018] does it for the JavaScript programming language to reconstruct web attack scenarios.

All of the programming language-related information flow monitors rely on a fine-grained tracking of the monitored programming language's instructions set. Therefore, they can be considered as *tracing systems* according to the terminology presented in Chapter 1.

### Instruction Level—Dynamic Information Flow Tracking

Information flows can be traced throughout the whole system by leveraging hardware facilities such as processor tracing technologies. More specifically, IFT techniques based on hardware facilities leverage tracing technologies presented in the *Tracing Systems* section of Chapter 1 (Section 1.1.4). This

family of techniques is called *Dynamic Information Flow Tracking* (DIFT) [Suh et al., 2004], *Dynamic Data Flow Tracking* (DFT) [Kemerlis et al., 2012] or *dynamic taint analysis* [Newsome & Song, 2005] [Clause et al., 2007].

Delving deeper into the information flow-related understanding of the processor-related abstraction layer, considered information containers correspond to processor's registers and memory words. Similarly to the previous paragraph, those information containers are handled by the processor's instruction set, which corresponds to the actions that produce the information flows.

As an illustration, the following x86 assembly code moves the 4 bytes in memory at the address contained in the ebx register into the eax register:

```
mov eax, [ebx]
```

In other words, the mov instruction produces an information flow from the memory address referenced by ebx into eax.

The great advantage of this family of technologies is that they do not rely on source code instrumentation. Some of the approaches leverage real-time binary rewriting to perform DFT [Newsome & Song, 2005]. For instance, the libdft library is coupled with Intel Pin tools (presented in Section 1.4.4) to dynamically instrument binaries for tracing.

Of course, tracing information flow dynamically has an impact on performance, i.e., performance overhead. Great research efforts have been made to reduce it. To address this limitation, Kannan et al. have proposed to decouple DIFT-related operations and regular computation by dedicating a co-processor for DIFT-related operations [Kannan et al., 2009].

Similarly to programming languages information flow monitors, processor information flow monitors also face the challenge of implicit information flow tracking and address this issue with control flow graph methodologies [Clause et al., 2007] [Balliu et al., 2017].

### Operating System—System Call-Based Information Flow Tracking

Applied to the OS abstraction layer, information flows are deduced from system call invocations. Considered information containers correspond to kernel objects, e.g., processes, files, sockets, and pipes. It is worth mentioning that research efforts have mainly been made on the Linux kernel. This abstraction layer has been thoroughly studied since King et al.'s seminal work "Backtracking Intrusions" in 2003 [King & Chen, 2003]. Their implementation relied on the instrumentation of a hypervisor. The monitoring OS was thereby virtualized. It is interesting to highlight the fact that King et al. do not mention the usage of information flows in their work. They immediately refer to it as "causal dependencies." Thus, information flow graphs have also been dubbed *dependency graphs*.

OS kernels have seen many improvements since 2003. Commercial off-the-shelf (COTS) kernel logging frameworks were created to enhance their visibility, i.e., enabling better debugging and auditing. For example, such frameworks are *auditd* for the Linux kernel, *DTrace*[2] for FreeBSD, and *Event Tracing Windows* (ETW) for Windows.

---

2 https://wiki.freebsd.org/DTrace

Many of the attack scenario reconstruction approaches are based on these COTS frameworks. All of these approaches rely on the construction of information flow graphs. For instance, the authors of PrioTracker propose a methodology to prioritize the traversal of information flow graphs [Liu et al., 2018]. Instead of computing a naive breadth-first traversal, they perform a search based on an information flow rarity score. The score is calculated according to the occurrence frequency, at an organization's scale, of the information flow between two given information containers. The rationale is that attack paths rely on rare actions. Other approaches focus on the investigation process of cyber defense analysts regarding system call-related events, i.e., events causing information flows [Gao et al., 2018]. Other approaches couple COTS framework system call events with tag propagation policies (based on access control and attack detection policies) to detect policy violations and retrieve all related system call events [Hossain et al., 2017].

Many approaches also propose to instrument the kernel to optimize the capture process of information flows and include secure data acquisition schemes [Ma et al., 2016] [Bates et al., 2015] [Pasquier et al., 2017]. For instance, auditd records Linux system calls without any modification. Further treatment needs to be done to translate a Linux system call into an information flow [Gehani & Tariq, 2012].

### Interoperating IFT among Different Abstraction Layers

Previous paragraphs of this section (Section 3.3.3) mainly presented approaches that solely focus on a single abstraction layer, e.g., source code, processor, or operating system. Information flow tracking can also be performed through several abstraction layers.

For instance, Hauser et al. developed a taint propagation technique over the network using an extension of the Internet protocol. Their idea is to enable taint tracing among several machines pertaining to the same network [Hauser et al., 2012]. Another example of information flow tracking inter-operation among abstraction layers is JBlare, which tracks information flows from the Java virtual machine down to the Linux operating system [Hiet et al., 2008].

---

This section presented the concept of information flow, its relation to causality, a formalization proposed by d'Ausbourg, as well as how information flow has been studied in different abstraction layers. Pursuing our search of causality, our research led us to the *provenance* research field, and its sub-field called *data provenance*. The next section is dedicated to them.

## 3.4 PROVENANCE PRIMER

### 3.4.1 Defining Provenance

According to Oxford's English Dictionary, *provenance* is defined as "the source or origin of an object; its history and pedigree; a record of the ulti-

mate derivation and passage of an item through its various owners." This concept has been well studied in the context of: *Art*, where it refers to "the documented history of an art object"; *Digital libraries*, where it refers to "the documentation of processes in a digital object's life cycle"; *Science*, where it ensures reproducibility of the scientific processes and analysis.

These few examples illustrate the diversity of application provenance addresses. Its definition is, therefore, dependent on the context in which it studied. In fact, the various user communities that use the term "provenance" have different underlying applications and needs in mind [Cheney et al., 2009]. However, they all seek to *explain*: where a given object comes from; how given object ended up in its state; or which objects were influenced by a given object. This search involves *causal reasoning*, and more specifically, deductive reasoning, as we have previously mentioned in Section 3.1.2.

### 3.4.2 Representing Provenance—The Open Provenance Model

In the words of the standardization proposed by Moreau et al. in [Moreau et al., 2011], the Open Provenance Model (OPM) standard allows the characterization of "what caused *things* to be, i.e., how *things* depended on others and resulted in specific states." *Things* can, for example, represent simulation results, physical objects, or digital data. Provenance is represented as a typed directed graph where nodes are linked by *causal relationships*.

*OPM Nodes*

The OPM is based on three types of nodes:

**ARTIFACT** "Immutable piece of state, which may have a physical embodiment in a physical object, or a digital representation in a computer system."

**PROCESS** "Action or series of actions performed on or caused by artifacts, and resulting in new artifacts."

**AGENT** "Contextual entity acting as a catalyst of a process, enabling, facilitating, controlling, or affecting its execution."

*OPM Edges—Causal Dependency Relationships*

Similarly to nodes, an OPM graph can have different types of edges:

**USED** "The process required the availability of the artifact to be able to complete its execution."

$$\text{Artifact} \xleftarrow{\text{used}} \text{Process}$$

**WASGENERATEDBY** "The process was required to initiate its execution for the artifact to have been generated."

$$\text{Process} \xleftarrow{\text{wasGeneratedBy}} \text{Artifact}$$

**WASCONTROLLEDBY** "The start and end of process $Process$ were controlled by agent $Agent$."

$$Agent \xleftarrow{wasControlledBy} Process$$

**WASTRIGGEREDBY** "The start of $Process_1$ was required for $Process_2$ to be able to complete."

$$Process_1 \xleftarrow{wasTriggeredBy} Process_2$$

**WASDERIVEDFROM** "$Artifact_1$ needs to have been generated for $Artifact_2$ to be generated. The piece of state associated with $Artifact_2$ is dependent on the presence of $Artifact_1$ or on the piece of state associated with $Artifact_1$."

$$Artifact_1 \xleftarrow{wasDerivedFrom} Artifact_2$$

These different types of edges represent the different kinds of causal dependency relationships considered in the OPM standard. It defines a causal relationship as "an arc that denotes the presence of a causal dependency between the source of the arc (the effect) and the destination of the arc (the cause)." These five kinds of causal dependency relationships are coherent with the one we defined in Section 3.1.1. More specifically, they encompass a cause, an effect, and the counterfactual dependency.

### 3.4.3 Data Provenance and Computer Security

In the context of our research, we are interested in the specific research subfield of *data provenance*. As advocated by the DARPA's Transparent Computing program [Defense Advanced Research Projects Agency, 2014], data provenance aims at providing transparency to "large opaque systems." Data provenance has naturally represented an important topic of the program. Regarding security monitoring, the value of data provenance is clear. It enables causal analysis: the explanation and understanding of how a suspicious digital artifact came to be in a given state [Bates & Hassan, 2019].

#### *Data Provenance's Relation to Information Flow*

Data provenance's literature mainly focuses on the study of digital artifacts such as processes, files, sockets. Doing so, considerable efforts have been made to analyze "audit events," i.e., system call events produced at the OS abstraction level. In fact, data provenance approaches are based on information flow monitors. This is simply explained by the fact that information flows represent causal dependency relationships among digital artifacts, i.e., information containers. Referring to the causal dependency types defined in the OPM standard (Section 3.4.2), information flows can correspond to the *used*, *wasGeneratedBy*, *wasTriggeredBy* and *wasDerivedFrom* causal dependency relationships.

### 3.4.4 Data Provenance in the Different Abstraction Layers

Similarly to information flow tracking, data provenance has been applied to the various abstraction layers composing a system.

*Operating System Abstraction Layer*

One of the most represented abstraction layers in the data provenance literature is the OS abstraction layer. Related data provenance monitors observe and record information-flow induced by system calls. This has been done either using COTS system call monitoring frameworks, such as SPADE which leverages auditd [Gehani & Tariq, 2012], or the approach proposed by Ma et al. which leverages Windows' ETW [Ma et al., 2015]; or by instrumenting the OS Kernel, e.g., PASS [Muniswamy-Reddy et al., 2006], Hi-Fi [Pohly et al., 2012], Linux Provenance Module (LPM) [Bates et al., 2015], CamFlow [Pasquier et al., 2017] and CamQuery [Pasquier et al., 2018].

More specifically, PASS, which stands for Provenance Aware Storage System, is one of the first provenance-aware operating system [Muniswamy-Reddy et al., 2006]. It was totally integrated into the Linux Kernel. Hi-Fi leveraged Linux Security Module (LSM) to hook system calls [Pohly et al., 2012]. Hi-Fi has also added netfilter hooks to handle socket-related provenance better. LPM also relies on LSM [Bates et al., 2015]. Additionally to system call recording, LPM also addressed the data acquisition security, i.e., securing the recording and storage of provenance inside the Linux Kernel. CamFlow [Pasquier et al., 2017] and CamQuery [Pasquier et al., 2018] go further in system call tracing by integrating Georget et al.'s work [Georget et al., 2017]. More specifically, Georget et al propose to update the LSM framework to track information flows in the Linux Kernel correctly. They highlighted the fact that LSM was initially designed for mandatory access control and that more information flow tracking-specific hooks needed to be added to correctly and completely track information flows. On top of the correct and complete recording of system calls, CamFlow and CamQuery also track information flows through shared memory and among threads instead of processes. These two features were not addressed by the approaches previously presented.

*Addressing the Dependence Explosion Problem*

All the approaches previously presented in the OS abstraction layer have a conservative assumption: any output information container of a process, or a thread in the case of CamFlow and CamQuery, is considered as causally dependent on the preceding input objects. This assumption can introduce false causal dependency relationships that can distort the provenance history of a given information container. This problem is known as the *dependence explosion problem* [Lee et al., 2013a][Ma et al., 2016]. In order to mitigate it, Lee et al's have introduced the notion of execution units with BEEP [Lee et al., 2013a]. Based on the observation that long-running processes (e.g., servers or Internet browsers) are designed with input-triggered loops that are generally independent and autonomous, processes' executions can be partitioned into execution units, a unit being a run of a loop. Execution

units are identified using software testing methodologies through static and dynamic binary analysis. These analyses can be considered as a learning stage. Inter-units' dependencies through shared memory are also identified by instrumenting the standard C library LibC. Once execution units have been identified, binaries are then modified and instrumented in order for execution units to invoke a specific system call when they start. This system call allows the distinction of execution units directly in the system call audit trail. This execution units-aware system call audit trail can then be used to build a finer-grained provenance graph where "process" nodes (in the sense of provenance graphs) correspond to execution units, instead of OS' classical "processes." This methodology has been thoroughly explored for the Linux Kernel with BEEP [Lee et al., 2013a] and ProTracer [Ma et al., 2016], and Windows [Ma et al., 2015].

### Towards Smaller Entities for Data Provenance

In fact, approaches presented in the previous paragraph propose solutions for a finer-grained tracking of information in the kernel abstraction layer. To achieve this goal, they proposed to decrease the size of digital artifacts, i.e., considering threads instead of processes (e.g., CamFlow [Pasquier et al., 2017] and CamQuery [Pasquier et al., 2018]), or going deeper, considering execution units instead of processes (e.g., BEEP [Lee et al., 2013a]).

This concept also applies to other types of information containers. The general idea is to track their inputs and outputs (I/O) with more precision. For instance, it has been applied to SQL databases in [Lee et al., 2013b]. The authors propose to consider a table entry as a *data unit*. A MySQL server has been completely instrumented to track such fine-grained information flow. This allows information flow tracking from the mysql process to table entry data unit, and vice versa, at the OS abstraction layer. It has also been applied to files in [Xu et al., 2016]. The information flow monitor they leverage tracks the lines written or read by processes. This awareness allows them to perform a finer-grained tracking of information flows among information containers: if process $p_1$ writes the first line of file $f$ and process $p_2$ reads the line $n$ of $f$, no information flows from $p_1$ to $p_2$ and the two processes' states are not causally dependent.

Going further in the search for accurate data provenance, DIFT techniques can also be leveraged.

### Programming Language Abstraction Layer

Another way to get finer-grained tracking of information flow is to work on programming languages directly. For instance, compilers can be loaded with specific modules to automatically instrument code according to a given policy. This is the case of Loom, an LLVM module developed in the context of the CADETS project [Strnad et al., 2019]. Another example of programming language instrumentation is the method leveraged in MPI [Ma et al., 2017]. Based on the observation that BEEP-related execution units were too low level for attack investigation, the authors of MPI propose to directly instrument the source code of applications to create semantics-aware execution units. Annotations would directly be added to the source code by their

developers that know its structure and mechanics, i.e., its semantics. MPI proposes different tools, such as annotation placement suggestions, to help them annotate the source code. Annotations serve as compilation indications to add binary code that invokes specific system calls. These special system calls indicate units, as well as their dependencies at the OS abstraction layer.

Finer-grained tracking of information flow can also be attained by instrumenting libraries, especially shared libraries, by precisely recording which function has been called [Wang et al., 2018]. Such monitoring enables the observation of attacks that leverage libraries.

### *Interoperating Provenance among Different Abstraction Layers*

Similarly to information flow monitors inter-operation through different abstraction layers, the data provenance research field proposed to merge provenance computed from different abstraction. This concept is called *provenance layering*. It was first introduced by the authors of PASS [Muniswamy-Reddy et al., 2006] [Muniswamy-Reddy et al., 2009]. More specifically, PASS is embedded in a Linux Kernel to enable data provenance tracking at the OS abstraction layer. To enable provenance-aware applications to send their provenance data to the PASS, an API, called the "Disclosed Provenance API" (DPAPI), has been developed. This API is based on specific DPAPI system calls to transfer provenance data from the provenance-aware applications to the PASS.

The concept of provenance layering continued since the seminal work of PASS. Recent work proposes to leverage network sniffing techniques, i.e., protocol dissection, and Linux Kernel instrumentation, namely, the Linux Provenance Module LPM, to track data provenance from the network to the Kernel [Bates et al., 2017]. The authors of LProv propose to leverage shared libraries instrumentation with BEEP binaries to compute fine-grained provenance graph [Wang et al., 2018]. The project CADETS proposes to enhance FreeBSD's auditing tool, i.e., *DTrace*, to handle information flow tracing in distributed systems [Strnad et al., 2019]. It couples it with the provenance data emanating from the instrumentation of source code using its *Loom* module for the LLVM compiler.

## 3.5 SUMMARY

Chapter 3 introduces the reader to the concept of causality and presents how causality and its surrounding concepts have been studied in the distributed systems, information flow and provenance research fields, i.e., the ones that are the closest to our research interests. Even if these research fields relate to a more general context, we have seen that they have also been applied to security monitoring, either in an IDS setup, e.g., IDS for distributed systems, or for attack scenario identification purposes in the case of information flow, and provenance.

**On Causality.** In more details, Section 3.1 started with the presentation of the surrounding concepts of causality: *counterfactuality*, i.e., given two events or object states A and B, $(A \implies B) \iff (\overline{A} \implies \overline{B})$; *causal reasoning* and its types, i.e., deductive, inductive and abductive; *internal and external causal dependencies*, i.e., causal dependencies that can respectively be explicitly perceived, or not, by a computer system; *causal dependencies computation*, i.e., leveraging data analysis techniques to discover relationships or leveraging approaches that explicitly highlight them. After the laying down of these foundational concepts, we present the main research fields that made up our research journey.

**Causality in Distributed Systems.** Section 3.2 presents the distributed systems research field, and its temporal causality relationship called *happened-before*. It starts by defining what a distributed system is: a system made up of distributed entities that only have partial knowledge of the various parameters involved in the problem that has to be solved. Secondly, it illustrates why causality has been studied in this research field. Then, Lamport's seminal work on the happened-before relationship [Lamport, 1978], denoted by "$\prec$," is presented. This relationship allows the construction of a partial order on the set of actions performed by the distributed entities of a distributed system. More specifically, given two actions $a$ and $b$, $a \prec b$ means that it is possible that $b$ is causally dependent on $a$. Finally, the end of this section presents different means to implement Lamport's relationship in order to compute causal dependencies among distributed entities' actions.

**Causality and Information Flow.** Section 3.3 presents the information flow research field as well as its relation causality. This research field models systems as sets of information containers where information can be transformed and/or transferred from one to another. These flows of information actually correspond to causal dependencies among information containers' states. In particular, this concept has been formalized by d'Ausbourg [d'Ausbourg, 1994]. This section presents its formalization. It then illustrates how information flows have been studied in various abstraction layers, e.g., programming languages and operating systems. It also illustrates how information flow monitors have been leveraged in order to perform attack scenario identification.

**Causality and Provenance.** Section 3.4 presents the provenance research field and its sub-field called data provenance. Provenance's relation to causality is clear: provenance seeks to *explain* where a given object comes from, how given object ended up in its state, or which objects were influenced by a given object. Data provenance represents a noticeable growing research field in the context of security monitoring, and more specifically, intrusion detection and attack scenario identification. The techniques leveraged by data provenance greatly rely on the work done in the information flow research field. Data provenance can also be seen as a sub-field of information flow that is specialized in tracking and storing objects' causal histories. Similarly to the previous section, Section 3.4 finishes by illustrating how data provenance has been studied in various abstraction layers.

**Towards a Formal Definition of the Event Causal Dependency Relationship.** Causality is an active subject in various research fields of computer science. Regarding the specific context of attack scenario identification for security monitoring, the amount of work done illustrates well how much causality seems promising to the research community. However, we observed that the literature lacks a formalization of the causal dependency relationships among heterogeneous events. We think that defining this link would help the research community. Based on the relationships defined by Lamport and d'Ausbourg, we formally define the heterogeneous event causal dependency relationship in the following chapter.

# Part II

# Towards a Unified Causality Model

# 4

## DEFINING A CAUSAL DEPENDENCY RELATIONSHIP AMONG HETEROGENEOUS EVENTS

Previous chapters have laid the fundamental concepts needed to comprehend our research field: the investigation of multi-step attack scenarios. Chapter 1 introduced the reader to the basics needed to perform security monitoring: enabling the observation of actions performed in this monitored system. Observed actions are recorded by monitoring systems as events. Chapter 2 presented how the alert correlation research field addresses the problem of attack identification through the analysis of events and alerts produced by the deployed monitoring systems. More specifically, alert correlation approaches aim at building connections among events and between attack steps. However, we have seen that these links are not trivial to define and discover, especially when considering heterogeneous events. Based on (1) the observation that the literature lacks of a formal definition of these connections, and (2) the insights gained through the study of causality and causal dependency computation presented in Chapter 3, Chapter 4 presents our contribution, i.e., the *event causal dependency* relationship. More specifically, Section 4.1 presents our problem through an attack scenario example. This example serves as an illustration throughout the second part (Part II) of this manuscript. Thereafter, the following sections introduce our model gradually, from the definitions of the concepts of *contextual actions*, *contextual events*, and their respective causal dependency relationships, to the definition of the causal dependency relationship among heterogeneous events.

## 4.1 ILLUSTRATION OF THE PROBLEMATIC AND PROPOSED MODEL

This section introduces the concept of causality analysis in heterogeneous and distributed event logs via a motivating attack scenario example. This example is used to further illustrate how each type of event is treated in our model.

The web server architecture consists of two machines, which respectively host an Apache server and a MySQL database. The security monitoring team deploys several monitoring systems, at different abstraction layers (as described in Chapter 1), to collect data and perform intrusion detection on the servers. Specifically, the servers are equipped with the Linux audit framework auditd, which enables the observation of the monitored system at the OS layer. The Apache and MySQL logging modules are activated for the web server and database server, respectively. These logging modules enable the observation of the monitored system at the application layer. Moreover, the database server is equipped with Zeek NIDS, which enables the obser-

vation of the network layer; auditd and netfilter are particularly configured to record system calls of interest and established connections, respectively.
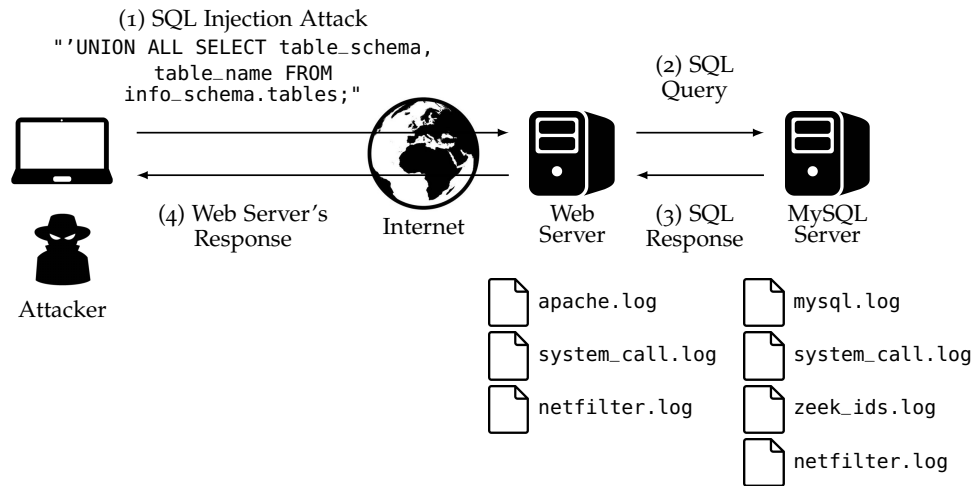


Figure 4.1: SQL injection attack scenario on a vulnerable web server.

The attack scenario, illustrated in Figure 4.1, is described as follows. After a discovery step, a malicious user performs an SQL injection via the POST parameters on an HTTP request to obtain the database table scheme. The cyber defense analyst is alerted by a Zeek alert, which describes an SQL injection attempt. Because this behavior is considerably suspicious, the cyber defense analyst decides to investigate the alert.

Because the web server initiates the SQL query, the cyber defense analyst immediately checks the events logged by the Apache server. However, because the Apache web server is not configured to record POST parameters, the cyber defense 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 cyber defense 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.

If the cyber defense analyst could automate this task, then he would have to specify that he intends to retrieve the attacker's traces, i.e., the different events corresponding to the observed and recorded attacker's actions, as defined in Section 1.1.3. These events are scattered in various heterogeneous logs. For simplicity, he can draw a space-time diagram, such as that illustrated in Figure 4.2, which captures the semantics of the 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 event is interpreted as representing

**Figure 4.2:** Visualization of events, alerts, and information flows on a space-time diagram.

the relationship among these entities and is placed on the timeline of the entity it describes. The different types of 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 Zeek 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 events pertain to relationships between two entities. In reading space-time diagrams by backtracking starting from the alert or the IoC, it can be deduced that several process activities can explain attackers' steps.

For example, by using the space-time diagram illustrated in Figure 4.2 and by backtracking from the NIDS alert, $e^{Net}_{conn,alert}$, the cyber defense analyst can see that it is linked to a netfilter event, $e^{Net}_{conn}$, which describes a connection related to a network socket of the Web Server host. This network socket is, in turn, linked to a write() system call from Apache, described by the $e^{Sys}_{write}$. Continuing with the reading of the space-time diagram, the cyber defense analyst can see that this write() system call event is further linked to an HTTP connection event, $e^{Net}_{Req}$. In the same fashion, the cyber defense analyst can leverage the space-time diagram to perform a forward tracking from the NIDS alert, and discover the various linked events that happened after it.

---

In the following sections, we formally present these relationships as causal dependencies among log entries and define a model that permits the generation of causal dependency links among events. All concepts are illustrated using the above example. 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.

## 4.2 LIMITATIONS OF LAMPORT'S AND D'AUSBOURG'S RE-LATIONSHIPS

In Chapter 3, we have presented the main research fields that deal with causal dependency computation, as well as their foundational relationships: Lamport's *happened-before* relationship [Lamport, 1978] and d'Ausbourg's causal dependency relationship [d'Ausbourg, 1994]. More specifically, we have seen that Lamport's relationship is defined on the set of actions performed by the processes of a distributed system; and d'Ausbourg's relationship is defined on the set of objects' states of a system. Chapter 3 already started to emphasize the particularities of these two types of models. We will further present them, as well as their limitations, in the following paragraphs. Thereafter, in the following sections, we explain how we have merged these two approaches to exploit them and define a new causality dependency notion called the *contextual action causal dependency* relationship.

### 4.2.1 Lamport's Happened–Before Relationship's Limitations

In Section 3.2.3, we mentioned that $a \prec b$ means that it is possible that action b is causally dependent on action a, a being any action that satisfies $a \prec b$. In other words, given an action b, the set of actions $\{a/a \prec b\}$ is a superset of actions that can influence b, i.e., it might contain actions that do not influence b. Therefore, this set might be an over-approximation of the set of actions that actually causally influence b. Actually, we can only assure that b is causally dependent on a when a corresponds to the sending of a given message and b corresponds to its reception, i.e., the second case of Lamport's relationship defined in Section 3.2.3. In fact, Lamport's relationship is temporal and does not take into consideration the context in which the actions are produced. Moreover, in this model, only application-level actions performed 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 events cannot be explicitly computed.

As it is, the concept of causal dependencies among concurrent processes' actions denotes *abductive reasoning*, as defined in Section 3.1.2: according to the distributed computation modeling, the relationship is the best hypothesis possible regarding actual causal dependencies among actions. In the context of a distributed computation, this causal dependency is actually an internal causal dependency, as described in Section 3.1.3.

### 4.2.2 D'Ausbourg's Causal Dependency Relationship's Limitations

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. Example 4.1 illustrates this point.

**Example 4.1** *Illustration of d'Ausbourg's and Lamport's relationships using a simple Python code*

```
a = 1      # (action_1, 1)
a = a + 1  # (action_2, 2)
a = 0      # (action_3, 3)
```

Consider a variable $a$ as an object. The Python code snippet of this example yields three states of $a$, with $(a, 1) = 1$, $(a, 2) = 2$, and $(a, 3) = 0$. As a result, $(a, 1) \rightarrow (a, 2)$, and $(a, 2) \nrightarrow (a, 3)$ 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 involves the causal dependency between $action\_2$ and $action\_3$, simply because $action\_2$ temporally *happens before* $action\_3$.

D'Ausbourg causal dependency relationship is applied to object states and not on events produced by monitoring systems that observe applications, operating systems, or the network. It is thereby challenging to use this model because object states are not easily captured by applications or system executions. In fact, events only represent the actions and information that are actually captured by monitoring mechanisms. Accordingly, we must define a model that takes into account both the action causal dependencies in time (similarly to Lamport's model) and the contextual states in which these actions are produced (similarly to d'Ausbourg's model).

This section presented the limitations of Lamport's and d'Ausbourg's relationships and started to explain why we want to define a more precise model. The following sections introduce the reader to the concept of event causal dependency relationship by gradually defining all the concepts surrounding it. Section 4.3 starts by introducing the concept of contextual action and the related causal dependency relationship.

## 4.3 CAUSAL DEPENDENCY AMONG CONTEXTUAL ACTIONS

The purpose of the *contextual action causal dependency* relationship 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*.

### 4.3.1 Contextual Action Definition

*Active and Passive Objects*

A contextual action consists 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, two categories of objects must be distinguished: *active objects*

(processes or network interfaces that produce actions) and *passive objects* (information containers, such as files, sockets, memory, and pipes that do not produce any action).

### Introducing Object Actions

An active object is supposed to produce actions that can be linked to the 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 4.1** *ObjectActions*$(o) = \{a_i\} \cup \{\emptyset\}$ *with* $\{a_i\}$ *the set of 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 consider that all library function invocations are actions of *ObjectActions*($p$).

### Defining Contextual Actions

We now formally introduce the concept of contextual action:

**Definition 4.2** *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* $t$.

As an example, given a process $p$ that calls a function $f()$, $p$ is actually performing a contextual action where the action, which corresponds to the call of $f()$, is performed in the specific state of the active object $p$, at the time of the function call, i.e., $(f(), (p, t_{f()}))$.

### Introducing Sessions

We have seen that the distributed systems research community supposes that for two actions, $a$ and $b$, produced by a given process such that $a \prec b$, $b$ is potentially 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, Section 3.4.4 presents approaches that study actions that delimit sessions inside long-running processes to reduce

the number of false causal dependencies among actions. The definition of the notion of session is as follows.

**Definition 4.3** *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* $Session_n(o) = \{(a_i, (o, t_i)) / (o, t_i) \rightarrow (o, t_{i+1}) \text{ and } (o, t_{end_{n-1}}) \not\rightarrow (o, t_{start_n}) \text{ and } (o, t_{end_n}) \not\rightarrow (o, t_{start_{n+1}})\}$; $t_{start_n}$ *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* $Session_n(o)$.
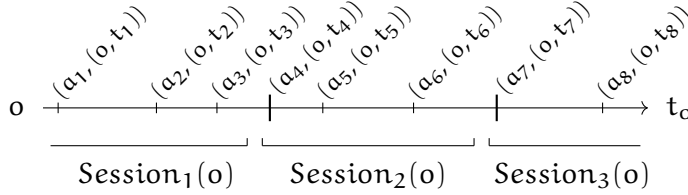


**Figure 4.3:** Sequence of contextual actions and sessions.

The execution of an object $o$ is the union of all $Sessions(o)$, as shown in Figure 4.3. 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 previous states, i.e., $(o, t_3)$ and $(o, t_6)$ respectively. 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 application or the OS can actually perform an action that starts a new session. 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.

### 4.3.2 Definition of the 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.

**Definition 4.4** *Given two contextual actions,* $(a_1, (o_1, t_1))$ *and* $(a_2, (o_2, t_2))$, *the latter is causally dependent on the former, written as* $(a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2))$:

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 $(c, (o, t)) \mapsto (a_2, (o_2, t_2))$.*
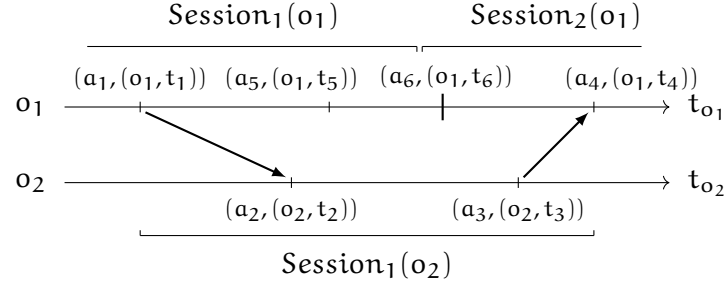


**Figure 4.4:** Contextual action causal dependency in different sessions.

Figure 4.4 illustrates the use of our model. Given two objects, $o_1$ and $o_2$, we have the following relationships among the different contextual actions:

- $(a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2)) \mapsto (a_3, (o_2, t_3)) \mapsto (a_4, (o_1, t_4))$;
- $(a_1, (o_1, t_1)) \mapsto (a_5, (o_1, t_5))$;
- $(a_6, (o_1, t_6)) \mapsto (a_4, (o_1, t_4))$.

Even if the contextual actions $(a_1, (o_1, t_1))$ and $(a_4, (o_1, t_4))$ belong to two different sessions, we have $(a_1, (o_1, t_1)) \mapsto (a_4, (o_1, t_4))$ because of the transitivity property of the the relationship "$\mapsto$." Moreover, as action $a_6$ starts a new session, it is implied that $(a_5, (o_1, t_5)) \not\mapsto (a_6, (o_1, t_6))$. Note that objects $o_1$ and $o_2$ have their own clocks, i.e., $t_{o_1}$ and $t_{o_2}$, respectively; no time synchronization is needed.

### Differences with Lamport's Relationship

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.

### Differences with d'Ausbourg's Relationship

At first, the difference between points (2) and (3) of Definition 4.4 might look ambiguous, i.e., an explicit message exchange between two active objects could be seen as an information flow. However, Lamport's relationship does not only deal with explicit message exchanges, it also encompasses implicit message exchanges as in the case of a *Rendez-Vous* between two processes.

Figure 4.5 illustrates an example of a Rendez-Vous between two threads. Such Rendez-Vous can be illustrated by the usage of a semaphore, where
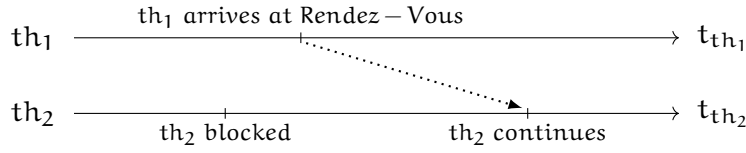
**Figure 4.5:** Rendez-vous between two threads.

a thread $th_2$ waits for a thread $th_1$ to release a common resource to continue its computation. In fact, the information is actually contained in the semaphore which is handled by the operating system. No message is explicitly exchanged by the threads in this example, i.e., $th_1$ does not send an actual message to $th_2$. However, we have necessarily $(th_1 \text{ arrives at } Rendez-Vous) \prec (th_2 \text{ continues})$. This situation can be seen as the exchange of an implicit message between the two threads.

---

This section described the notions of contextual actions and the causal dependencies among them. The following section introduces the notion of contextual event, as well as the relationship between contextual actions and contextual events.

## 4.4 FROM CONTEXTUAL ACTIONS TO CONTEXTUAL EVENTS AND EVENTS

### 4.4.1 Contextual Event Definition

As we have seen in Chapter 1, a given action can either be: (1) observed by different monitoring systems and be recorded as several log entries in heterogeneous logs; or (2) unobserved, thus implying that it is not recorded in a log entry. It can consequently be missed by a cyber defense analyst.

Given the set of the system's events, denoted $\mathbb{E}$, each 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 4.5** *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* $e$.

The relationship between events and contextual events is pretty straightforward: each contextual event corresponds to a unique event. Using the example of Section 4.1, $(e_{Req}^{App}, apache, t_{e_{Req}^{App}})$ represents a contextual event of the observed active object $apache$. However, Definition 4.5 needs to be extended to cover the lack of observation of a contextual action. According to definition 4.2, the action $a$ of a given contextual action, $(a, (o, t_a))$, might represent a real action or the lack of action; hence, the action $a$ 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 $t_a$.

*Contextual Events as Observations of Contextual Actions*

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 4.6** *Given an action* $a \in \text{ObjectActions}(o)$ *occurring at time* $t_a$, *the observation of a contextual action is* $\text{Obs}((a, (o, t_a))) = \{(e_i, o, t_{e_i})\} \cup \{(\emptyset, o, t_a)\}$, *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.*

Using the previous example used for Definition 4.5, $(e_{Req}^{App}, apache, t_{e_{Req}^{App}})$ is to the observation of the contextual action corresponding to the execution of a LPS by the active object $apache$. Thus, in this example, the Obs function is implemented as an LPS.

*On Event Timestamp and Actual Action Time*

It must be noted that in Definition 4.6, 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, e.g., application, network or OS, and its observations are the same as that of the observed object.

### 4.4.2 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 4.7** *Given two contextual events* $(e_1, o_1, t_{e_1})$ *and* $(e_2, o_2, t_{e_2})$, *with* $e_1 \in \mathbb{E}$ *and* $e_2 \in \mathbb{E}$, *the latter is causally dependent on the former, written as* $(e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2})$, *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* $(e_1, o_1, t_{e_1}) \in \text{Obs}((a_1, (o_1, t_1)))$ *and* $(e_2, o_2, t_{e_2}) \in \text{Obs}((a_2, (o_2, t_2)))$.

### 4.4.3 Definition of the Event Causal Dependency Relationship

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 4.8** *Given two events,* $e_1 \in \mathbb{E}$ *and* $e_2 \in \mathbb{E}$, *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.*

---
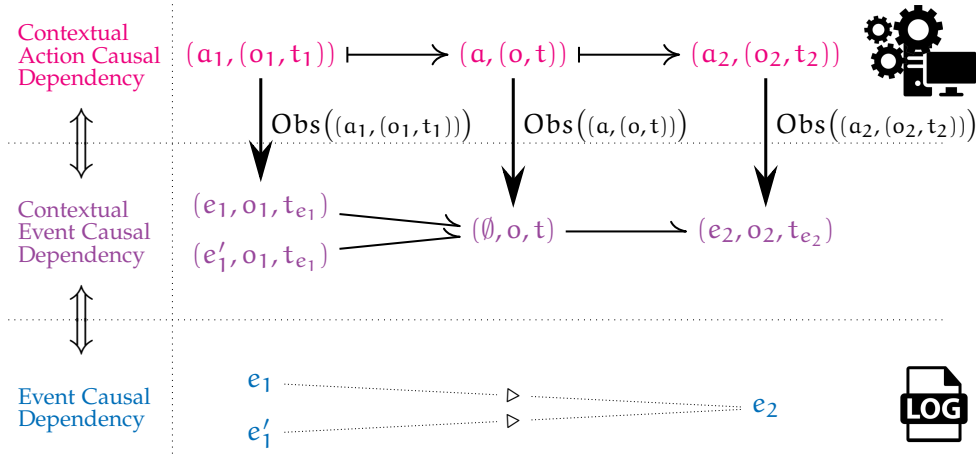
**Figure 4.6:** Illustrative summary of the three relationships we defined.

We have defined three new causal dependency relationships, "$\mapsto$," "$\rightharpoonup$," and "$\triangleright$," that define partial orders on the set of contextual actions, the set of contextual events, and the set of events, respectively.

Figure 4.6 summarizes the new relationships we defined in this chapter. These relationships can be seen as three representations of the monitored system. The model at the top (i.e., the contextual action causal dependency (CACD) model) considers the actions, the object states, as well as their causal dependencies. The bottom model (i.e., the event causal dependency (ECD) model) considers the events produced by the heterogeneous monitoring systems deployed, as well as their causal dependencies. The middle model (i.e., the contextual event causal dependency (CECD) model) considers contextual events, a notion that bridges the gap between contextual actions and *raw* events, as well as their causal dependencies.

The left part of the figure highlights the fact that the three causal dependency relationships we define are *equivalent*. Delving deeper in the details, the term "equivalent" is to be understood as a double implication. The following examples illustrate this clarification.

**From CACD to CECD.** Given two *observed* and *causally dependent* contextual actions, $(a_1, (o_1, t_1))$ and $(a_2, (o_2, t_2))$, and their related contextual events, $(e_1, o_1, t_{e_1}) \in \text{Obs}((a_1, (o_1, t_1)))$ and $(e_2, o_2, t_{e_2}) \in \text{Obs}((a_2, (o_2, t_2)))$, we have: $((a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2))) \implies ((e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2}))$.

**From CECD to CACD.** Given two *causally dependent* contextual events, $(e_1, o_1, t_{e_1})$ and $(e_2, o_2, t_{e_2})$, we *know* that these contextual events are actually the observation of two contextual actions, $(a_1, (o_1, t_1))$ and $(a_2, (o_2, t_2))$, with $(e_1, o_1, t_{e_1}) \in \text{Obs}((a_1, (o_1, t_1)))$ and $(e_2, o_2, t_{e_2}) \in \text{Obs}((a_2, (o_2, t_2)))$, and that these contextual actions are *causally dependent*, i.e., $((e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2})) \implies ((a_1, (o_1, t_1)) \mapsto (a_2, (o_2, t_2)))$.
However, we might not be able to specifically identify and/or compute these contextual actions. In practice, complete object states are rarely captured by monitoring systems. Therefore, the contextual action causal dependency abstraction layer is hard to compute and can only be approximated most of the time.

**From CECD to ECD.** Given two *causally dependent* contextual events, $(e_1, o_1, t_{e_1})$ and $(e_2, o_2, t_{e_2})$, their related events are *causally dependent*. Thus, we have: $\left( (e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2}) \right) \implies (e_1 \triangleright e_2)$.

**From ECD to CECD.** Given two *causally dependent* events, $e_1 \in \mathbb{E}$ and $e_2 \in \mathbb{E}$, we *know* these events emanate from the context of monitored objects, denoted $o_1$ and $o_2$, i.e., $e_1$ and $e_2$ are related to the contextual events $(e_1, o_1, t_{e_1})$ and $(e_2, o_2, t_{e_2})$. Thus, we have:
$(e_1 \triangleright e_2) \implies \left( (e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2}) \right)$.
However, similarly to the case of contextual actions, we might not be able to compute contextual events from *raw* events. For example, a raw event might not contain the information needed to identify its related observed object.

---

The next section introduces the reader to the aim of defining a causal dependency relationship among heterogeneous events: enabling the computation of the cause and dependence graphs of an event of interest.

## 4.5 CAUSE AND DEPENDENCE GRAPHS

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

**Definition 4.9** *Given $e \in \mathbb{E}$, the cause graph of $e$ is defined as:*

$$\text{cause}(e) = \{e'/e' \triangleright e\}$$

**Definition 4.10** *Given $e \in \mathbb{E}$, the dependence graph of $e$ is defined as:*

$$\text{dep}(e) = \{e'/e \triangleright e'\}$$
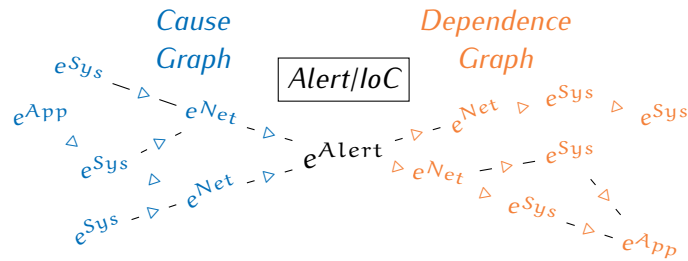


**Figure 4.7:** Cause and dependence graphs of an event of interest.

Figure 4.7 illustrates the cause and dependence graphs of a given event of interest, e.g., an alert produced by an IDS. Of course, cause and dependence graphs can be defined for contextual actions and contextual events as well.

*Event Cause and Dependence Graphs of the Attack Scenario Example*

Chapter 5 describes how each type of events used in the example of Section 4.1 is processed to compute contextual events and their related causal dependencies. It explains how to obtain the space-time diagram shown in Figure 4.2. Setting all contextual events on their related object timelines, we can build the cause and dependency graph for alert $(e_{conn,alert}^{Net}, netw_{zeek}, t_{alert})$. Then, event causal dependencies can be easily deduced using Definition 4.8.
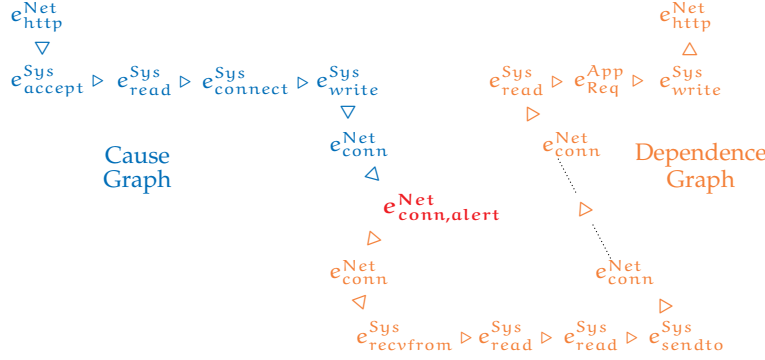


**Figure 4.8:** Event cause and dependence graphs of the NIDS alert.

Figure 4.8 represents the resulting event cause and dependence graphs of the attack scenario example of Section 4.1. The cyber defense analyst would have all the events that causally contribute to or causally depend on the raised NIDS alert.

---

## 4.6 SUMMARY

Chapter 4 presents the main contribution of the work presented in this manuscript: the formal definition of the causal dependency relationship among heterogeneous events. The model we define aims at unifying previous work on causal dependencies presented in Chapter 3, and at addressing the lack of the definition of the semantics of the links among events. This chapter starts by presenting the example that will serve as an illustration throughout the rest of the manuscript. The following sections of the chapter gradually introduce our model.

**Limitations of Lamport's and d'Ausbourg's Models.** Section 4.2 starts by explaining the limitations of the foundational causal dependency relationships presented in Chapter 3, namely, Lamport's *happened-before* relationship [Lamport, 1978] and d'Ausbourg's causal dependency relationship [d'Ausbourg, 1994]. Briefly, Lamport's relationship is temporal and does not take into consideration the context in which the actions are produced. On the other hand, D'Ausbourg causal dependency relationship is applied to object states and not on events produced by monitoring systems. It is thereby challenging to use this model because object states are not easily captured by applications

or system executions. Building upon Lamport's and d'Ausbourg's relationships, we aim at defining a model that takes into account both the action causal dependencies in time (similarly to Lamport's model) and the contextual states in which these actions are produced (similarly to d'Ausbourg's model). The model we propose is gradually introduced in Sections 4.3 and 4.4. This new causality model is made up of three relationships that represent three different perspectives of the monitored system.

**Defining a New Model: Causal Dependency Among Contextual Actions.** Section 4.3 starts with the definition of the *contextual action causal dependency* (CACD). Additionally, it explains the differences between the CACD relationship and the ones defined by Lamport and d'Ausbourg. Succinctly, the CACD relationship we define is based on the insight that a system can be modeled as *a set of active and passive objects that have states*. It considers the actions, the object states, as well as their causal dependencies.

**Towards the Definition of the Event Causal Dependency.** Building upon the contextual action causal dependency relationship, we gradually define the notions that are necessary to link the contextual actions to their corresponding events and define the *event causal dependency* (ECD) relationship in Section 4.4. More specifically, the definition of the ECD relationship is based on the insight that events, actions, and object states are closely related, i.e., *monitoring systems enable the observation and recording of actions or states in the form of events*. The idea is the following: *If we can determine causal dependencies among actions, among states, and between actions and states, we can propagate this knowledge to the events.*

**Enabling the Computation of Cause and Dependence Graphs.** Finally, Section 4.5 introduces the aim of defining a causal dependency relationship among heterogeneous events: *enabling the computation of the cause and the dependence graphs of an event of interest* (e.g., an alert or an indicator of compromise, produced by an attacker action).

The work presented in this chapter has been published and presented at the 4th IEEE European Symposium on Security and Privacy (EuroS&P 2019) [Xosanavongsa et al., 2019a], as well as at the 5th *Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information* (RESSI 2019) [Xosanavongsa et al., 2019b]. The next chapter presents how the new causal dependency model we define can be computed.

# 5 | MODEL IMPLEMENTATION

The previous chapter presented the model we have defined to unify two existing work on causal dependencies among heterogeneous events. This chapter presents how this new model can be implemented. More specifically, Section 5.1 illustrates the two strategies that can be adopted to implement the model, i.e., the *top-down* and the *bottom-up* strategies. The following sections present how we can implement these strategies in the context of our research interest: the monitoring of the computer network of an enterprise. Section 5.2 describes how our model would be ideally implemented with the top-down strategy. Finally, Sections 5.3 and 5.4 present our implementations of the contextual event causal dependency relationship, using the bottom-up strategy, for the Linux OS and the Windows OS, respectively. More specifically, Section 5.4 introduces the reader to the project called VESTA. VESTA consists in the development of an endpoint detection and response product and embeds a part of the work presented in this manuscript.

## 5.1 TOP–DOWN AND BOTTOM–UP PERSPECTIVES

The first section of this chapter aims to introduce the reader to the two strategies that can be adopted to implement our causal dependency model, namely, the *top-down* and the *bottom-up* strategies.

As we have seen with Figure 4.6, the three relationships we have defined can be seen as three representations of the monitored system. As these relationships are equivalent, computing the event causal dependency relationship, based on Figure 4.6, can be done either:

1. by adopting a *top-down* strategy, i.e., tracking causal dependencies among contextual actions, and computing the $Obs$ function that maps a contextual action to its corresponding set of contextual events. Section 5.1.1 describes this first strategy;

2. or by adopting a *bottom-up* strategy, i.e., computing contextual events, as well as their causal dependencies, from *raw* events. Section 5.1.2 describes this second strategy.

### 5.1.1 Top-Down—From the Contextual Action Causal Dependency Relationship to the Event Causal Dependency Relationship

The top-down strategy's global idea is the following. If (1) the causal dependencies among contextual actions are known; and (2) the observability relationship between contextual actions and contextual events is known through the computation of the $Obs$ function; Then, we can compute causal dependencies among contextual events and among events, based on the causal dependencies among contextual actions.

In order to adopt this strategy, we should thereby be able to compute the contextual action causal dependency model. This implies that we should be able to:

1. observe actions performed by the monitored active objects;
2. observe the object states in which these actions take place;
3. identify, observe and track the causal dependencies among contextual actions, as defined in Section 4.3.2;
4. compute the Obs function that maps a contextual action to its corresponding set of contextual events.

The following section (Section 5.2) describes how the top-down strategy can be implemented.

### 5.1.2 Bottom-Up—Leveraging Events Information to Compute the Contextual Event Causal Dependency Relationship

The bottom-up strategy's global idea is the following. If *raw* events explicitly contain the needed semantic information; Then we might be able to compute contextual events and their causal dependencies from events. Additionally, timestamps may be used to approximate the ordering of contextual actions performed on the same machine.

Building upon the contextual event causal dependency model, we would need to compute contextual actions from contextual events. Again, it depends on the semantic information contained in the events. Events generally contain information related to actions or notice changes of states. However, it is mostly impossible to have access to the object states. Therefore, it is very difficult to compute the contextual action causal dependency model.

———————

This section presented the two strategies that can be adopted to compute the three relationships we defined in Chapter 4. Of course, the implementation of these strategies is conditioned by the nature of the monitored system. In our context, the monitored system consists in a classical enterprise computer network without additional deployed monitoring systems. The rest of this chapter is organized into two parts.

The first part, consisting in Section 5.2, describes how a classical system could be modified to attain an implementation of the top-down strategy. More specifically, it shows how current technologies enable the computation of parts of the prerequisites for the application of top-down strategy.

The second part, consisting in Section 5.3 and 5.4, presents our implementations of the model. These implementations adopt the bottom-up strategy.

## 5.2 TOP-DOWN STRATEGY—IDEAL IMPLEMENTATION DESCRIPTION

There is currently no complete implementation of the overall model we defined in Chapter 4. However, the contextual action causal dependency

computation prerequisites, described in the previous section, can be partially satisfied by current causal dependency computation methodologies, as presented in Chapter 3. An ideal implementation of the top-down strategy would, therefore, consist in the merging of several methods and technologies. In fact, all these methods and implementations allow the observation and recording of a subset of actions performed by active objects in the monitored system. More specifically, each implementation permits observations at a given level of the system and thus provides partial information of what really happens in the system. Accordingly, it is important to note that the existing work only enables an *approximation* of the correct model. This section presents how existing work enables the computation of parts of the contextual action causal dependency model. It also presents why we can only attain an approximation of the contextual action causal dependency model by introducing the reader to the limitations of presented methodologies.

### 5.2.1 Message Passing Systems—Causal Dependencies among Actions

As we have presented in the chapter on causality (Chapter 3), every work in this distributed system research field implements various means to deduce causal dependencies among distributed processes' actions. Actually, any of these technical solutions is relevant for computing contextual action dependencies of different processes that communicate. Going into more detail, a message exchange consists of two causally dependent actions: a sending action and a reception action. Thus, message passing-related causal dependencies relate to causal dependency among actions.

### 5.2.2 Information Flow Monitoring—Causal Dependencies among Object States

The various kinds of information flow monitoring we presented in Chapter 3 can be leveraged to compute causal dependencies among contextual actions. More specifically, we have seen that an information flow represents a kind of causal dependency among information containers, i.e., object states.

#### *Delving Deeper into Kernel-Level Information Flow Monitors*

Sections 3.3.3 and 3.4.4 presented information flow monitors at the OS abstraction level, i.e., the Kernel abstraction level. These work monitor system calls invoked by active objects, i.e., processes, threads, or execution units, and deduce information flows from them.

**Monitoring of System Call Invocations.** These monitors allow the computation of *causal dependencies among object states* by tracking the *invocations* of system calls. In practice, this family of monitors allows the identification of the objects involved in observed system call invocations. The most fine-grained monitors also allow the recording of the value of the information that is actually flowing through the system call. However, they do not observe the actual states of the objects involved in system calls. Thus, they don't allow the computation of object states nor the computation of contextual actions.

**Approximated Occurrence Time.** Additionally, these monitors capture system call-related actions at a certain point in time, which is not the exact time of occurrence of the actions, as discussed in Section 4.4.1. Even if object states could be observed and recorded, the previous remark implies that object states might not be observed at the *actual* time the actions occurred, i.e., object states might change during this time frame.

**False Causal Dependencies.** As we have already mentioned in Section 3.4.4, system call-related information flow monitors are subject to the dependence explosion problem. They thereby might introduce false causal dependencies among contextual actions. This also contributes to the fact that only an approximation of the contextual action causal dependency model can be computed.

### *Delving Deeper into Programming Language–Level Information Flow Monitors*

The same conclusion applies to information flow monitors of the programming language abstraction layers, i.e., work presented in Sections 3.3.3 and 3.4.4. As the name *data flow tracking* suggests, only the flow of data is tracked. Such systems do not observe the values of the variables. Thus, the values of object states cannot be computed by such methodologies. However, these methodologies are more precise than Kernel-level information flow monitors.

---

The existing work presented in the previous paragraphs only observe and record active object actions. They do not provide a means to observe and record object states, either active or passive, at any given time. Hence, they do not enable the computation of the contextual action causal dependency model.

### 5.2.3 Eidetic Systems—Towards State Awareness

Computing the complete contextual action causal dependency model requires the capability to observe and record object states at any moment during system execution. Such object state awareness is enabled in debugging and replaying tools, where an action can be stopped and its context captured, i.e., the state of the object that performed the action and the states of involved objects. Such capture mechanisms can be embedded in processors, virtual machine hypervisors, or emulators, such as QEMU, as that in [Chow et al., 2008].

### *Defining Eidetic Systems*

In 2014, Devecsery et al. introduced the concept of state-aware systems [Devecsery et al., 2014]. They termed them *eidetic systems*, i.e., "a computer system which can *record* and *replay* any prior computation, just as if it were happening again". This concept can be applied at different abstraction layers, e.g., the OS or application abstraction layers.

### Eidetic Systems at the Operating System Layer

Devecsery et al. applied it to the Linux operating system. Their implementation is embedded in the Linux Kernel and enables the recording of process states and their evolution to replay subsets of processes [Devecsery et al., 2014].

Building upon Devecsery et al.'s work, Ji et al. have proposed to use this record and replay technology to perform attack investigation [Ji et al., 2017] [Ji et al., 2018]. Hence, they record the required information to perform classical Kernel-level information flow tracking, as well as process replay. More specifically, recorded information corresponds to the context of processes I/O, e.g., system call results, or shared memory reads and writes. Their idea is to leverage the replay capability of the eidetic Linux Kernel to perform fine-grained information flow tracking at the instruction level, i.e., Dynamic Information Flow Tracking (DIFT) techniques (as described in Section 3.3.3). In practice, using DIFT is not realistic for real-time usage as it induces too much performance overhead. However, using DIFT in a replay setting allows avoiding the induced performance overhead in the production context. Thus, this idea allows to address the dependence explosion problem (presented in Section 3.4.4) and to retrieve the traces of an attacker with more precision as well.

### Eidetic Systems at the Application Layer

These principles can also apply at the application abstraction level. For example, Vadrevu et al. 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 [Vadrevu et al., 2017]. By enabling the finer-grained logging of action contexts, these methodologies also allow the computation of a better approximation of the contextual action causal dependency model.

---

In fact, the implementations presented in this section (Section 5.2.3) instrument the studied abstraction layer to record the information required to enable replay: they build *tailor-made* monitoring systems. However, such heavyweight instrumentation is not always possible or desirable and is difficult to maintain.

### 5.2.4 Obs Function Implementation Ideas

Based on the hypothesis that contextual actions and their related causal dependencies can be computed, an implementation of the Obs function would allow us to compute contextual events, as well as their corresponding causal dependencies.

Existing technologies that correspond to an implementation of the Obs function can be system call monitors, application logging systems, or network packet sniffers, for instance.

Contextual Action Causal Dependency [Known]

$(a_1, (o_1, t_1))$ $\longmapsto$ $(a_2, (o_2, t_2))$

$\mathrm{Obs}\big((a_1, (o_1, t_1))\big)$ $\qquad$ $\mathrm{Obs}\big((a_2, (o_2, t_2))\big)$

Contextual Event Causal Dependency

$(e_1, o_1, t_{e_1})$ $\qquad\qquad\qquad$ $(e_2, o_2, t_{e_2})$

Event Causal Dependency

$e_1$ $\qquad\qquad\qquad\qquad$ $e_2$
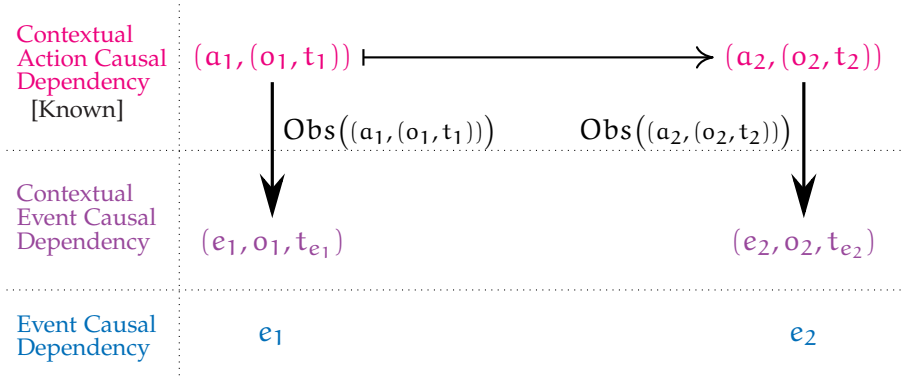
Figure 5.1: Illustration of a causal dependency tracking system.

We could also design a more global implementation of the Obs function. For instance, we can imagine a tracking system for contextual actions and their causal dependencies. Tracking information could be recorded by the Obs function. Figure 5.1 illustrates this implementation idea. Additionally to the recorded information, e.g., the performed action and the object state at the recording time, events would also have the same tracking information as their corresponding contextual actions. Thus, with the example illustrated in the figure, we have $(e_1, o_1, t_{e_1}) \rightharpoonup (e_2, o_2, t_{e_2})$ and $e_1 \triangleright e_2$.

_____

In practice, it is difficult to gather and integrate all these technologies together to get the complete model of the contextual actions and their causal dependencies. Therefore, as a first step, we have decided to adopt a bottom-up strategy in order to compute the event causal dependency model. The bottom-up strategy is more realistic to implement. However, it is based on an incomplete coverage of the actions performed by the monitored system. The computed representation is thereby an approximation of the model we defined in Chapter 4. The following section presents our implementation of the contextual event causal dependency model using the bottom-up strategy.

## 5.3 BOTTOM–UP STRATEGY—A LIGHTWEIGHT APPROACH

The security monitoring strategy is the key to enabling the computation of contextual events and their related causal dependencies. Depending on the chosen monitored abstraction layers, different kinds of monitoring systems can be deployed to produce events containing the necessary semantics information for this computation.

This section presents our implementation of the contextual event causal dependency model using exclusively existing monitoring system facilities to record events. Our implementation does not rely on any additional instrumentation of the monitored system (e.g., source code instrumentation). More specifically, we chose to observe the network, OS, and application abstraction layers. In the next sections, as well as in Chapter 6, it is shown that the use of existing events generated by various monitoring systems, such as

auditd, netfilter, application logging systems, and Zeek NIDS, already yields a good approximation of the model.

Building upon the semantic information contained in raw events, our implementation adopts a bottom-up strategy, as described in Section 5.1.2. This section is organized into three parts. This first one describes how a cyber defense analyst would leverage our model to identify and retrieve all the events that are causally linked to the suspicious event that is being investigated (Section 5.3.1). The second one presents the architecture of our implementation (Section 5.3.2). Finally, the third one goes into more detail by describing how the contextual event causal dependency model is computed from raw events (Section 5.3.3).

### 5.3.1 Investigating an Event—A Cyber Defense Analyst's Perspective

Actually, the lightweight implementation that is proposed in this section involves taking the perspective of a cyber defense analyst who only has events and alerts.

Starting from an event of interest $e$, a cyber defense analyst tries to find links between heterogeneous events and $e$ to understand its context better. In fact, as described in Chapter 2, we argue that these links correspond to causal dependencies among heterogeneous events. A cyber defense analyst is, therefore, performing graph traversals, i.e., by building the cause and dependence graphs, of the *event causal dependency* model presented in Chapter 4.

The implementation of the bottom-up strategy we propose aims at automating the computation of these links and the retrieval of the set of events that are causally dependent on $e$. Thus, we provide a cyber defense analyst an interface to request the cause and dependence graphs of any event of interest. More specifically, if this event is not a part of a real attack, then the dependence and cause graphs will not contain traces of an attack. Accordingly, a cyber defense analyst should begin with a suspicious event. In this case, the probability that the graph contains all events related to an attack is higher. Such capability represents a means to assert the veracity of a given alert, as well as a means to identify multi-step attack scenarios.

Following sections describe our implementation of the bottom-up strategy:

- Section 5.3.2 presents the design of our implementation. More specifically, it details why we have chosen the *ArangoDB*[1] graph database to implement the contextual event causal dependency model;
- Section 5.3.3 focuses on the computation of contextual events, as well as their causal dependencies, from raw events. Additionally, it answers the following questions:
    - What actions do we monitor?
    - How do we generate events?

---

[1] https://www.Arangodb.com/

### 5.3.2 Implementation's Architecture

The architecture of our implementation is divided in four parts: (1) The monitoring systems deployment; (2) The Extract, Transform, Load (ETL) pipelines; (3) the graph database; and (4) the visualization interface that allows the cyber defense analyst to interact with the database and investigate events. Figure 5.2 illustrates the big picture of our implementation's architecture.
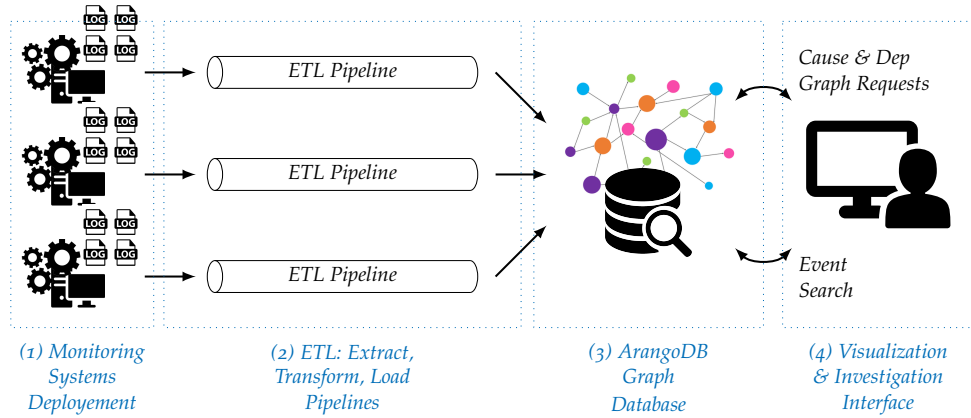


Figure 5.2: Overview of the architecture of our implementation.

#### *Monitoring Systems Deployment*

The monitoring systems deployment depends on the adopted monitoring strategy. More specifically, the monitoring strategy consists in choosing:

1. which objects' actions, states, or related causal dependencies are observed and recorded to enable efficient attack detection and investigation.
2. produced events' formats, according to the format output possibilities offered by the deployed monitoring systems.

The monitoring strategy, in turn, depends on the context, i.e., the system to monitor. In our case, the system to monitor consists in a small enterprise network. Additionally, we have decided to implement our model using only COTS monitoring systems. In this monitoring context, we monitor network objects, i.e., network interfaces of the monitored machines, as well as Kernel-level objects, e.g., processes, files, pipes, sockets. Moreover, we consider that application events are produced by processes. Monitored object actions correspond to system call invocations, message exchanges over the network, and the actions observed by the monitored applications' logging systems. Section 5.3.3 details how each type of data source is handled to produce contextual events and causal dependencies among contextual events pertaining to different objects.

#### *ETL Pipelines*

As its name suggests, an ETL pipeline corresponds to a process that moves data from a source (extraction) to a destination (load), and modifies it (transformation) along the way.
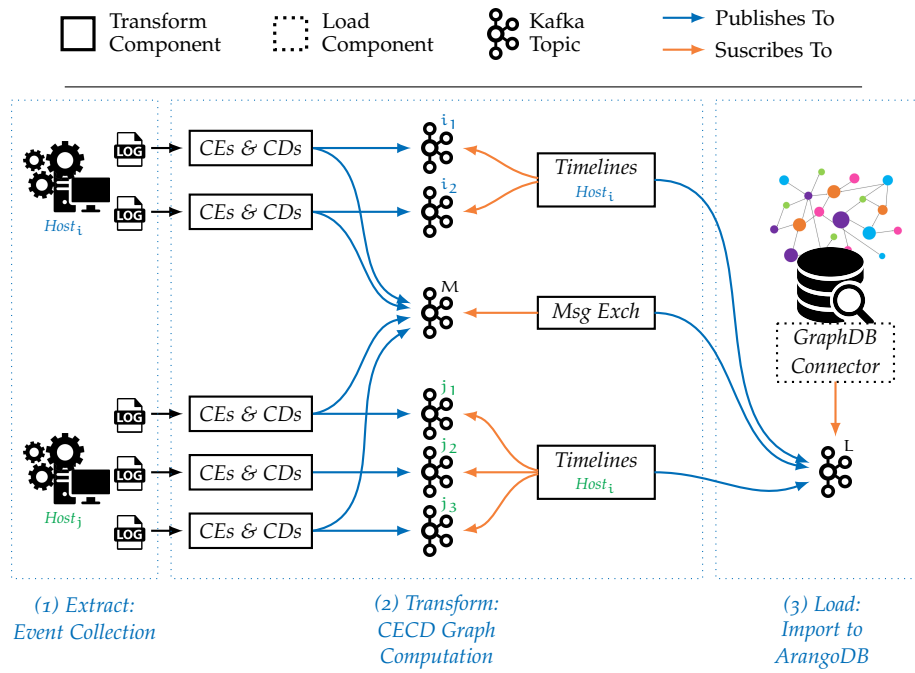
**Figure 5.3:** Illustration of the ETL pipelines architecture.

Figure 5.3 represents the architecture of our ETL pipelines. They consist in: (1) collecting the raw events on the monitored machines; (2) computing contextual events and, when possible, their causal dependencies; and (3) loading the computation results in the graph database.

**Kafka Topics.** ETL pipelines follow a publish-subscribe model, also called the producer-consumer model, i.e., each component of the pipelines publishes, subscribes, or both, to streams of events. In our implementation, event streams are handled by Apache Kafka[2], a scalable distributed streaming platform that allows building ETL pipelines. Kafka enables the reliable transportation of events from their sources, to the different contextual events and causal dependencies computation components (i.e., transform components), and the graph database. More specifically, event streams are handled by *topics* in Kafka. Thus, extract, transform, or load components publish, subscribe, or both, to topics, as we can see in Figure 5.3. Kafka is specially configured to keep the publishing order of messages.

---

The following paragraphs describe the extract, transform, and load components of the ETL pipelines. More specifically, the transform part of the ETL pipelines is made up of three types of components: (1) Contextual Events and Causal Dependencies (CEs & CDs) transform components; (2) Timelines transform components; and (3) Message exchange transform component.

**Extract.** The extract part of the ETL pipelines currently consists in the reading of event log files by the *CEs & CDs transform components*. These components parse raw events and compute the contextual events and their causal

dependencies before publishing them to one or several Kafka topics. We currently suppose that each event log file is temporally ordered.

**CEs & CDs Transform Components.** As we can see in Figure 5.3, each *CEs & CDs transform component* publishes to its own dedicated topic. More specifically, it publishes all the contextual events and causal dependencies it computed. When computed contextual events are message exchange-related, e.g., network communication-related, *CEs & CDs transform components* also publish them to the *MSG Exch* topic.

*CEs & CDs transform components* aim to identify the monitored objects by parsing the event. Depending on its type, an event can contain one or several objects. For instance, a system call event contains two objects. Moreover, event semantics can also allow the deduction of causal dependencies, e.g., a system call that produces an information flow between two object states;

Depending on the data source to transform, *CEs & CDs transform components* might need to be stateful, i.e., remembering preceding events. For instance, this is the case with system call and netfilter events.

**Timelines Transform Components.** *Timelines transform components* aim to build object's timelines, as well as object's sessions. Each monitored host has its own *timelines transform component*, which actually builds its local contextual event causal dependency graph. In order to do so, a *timelines transform component* subscribes to all the topics related to a given monitored host. For instance, in Figure 5.3, topics $i_1$ and $i_2$ are dedicated to the *CEs & CDs transform components* of $Host_i$. In our current implementation, we suppose that all the objects pertaining to the same monitored host share the host's local clock. However, we do not consider that different hosts have synchronized clocks. A *timelines transform component* starts by ordering the CEs streams it has subscribed to before appending them to the corresponding objects' timelines. Doing so, a *timelines transform component* actually totally orders all the events emanating from a given host.

Naturally, *timelines transform components* are stateful. They have to remember the last contextual event of each monitored object to build causal dependencies related to their timelines. All *timelines transform components* publish to the topic L.

**Message Exchange Transform Component.** The *message exchange transform component* aims to build causal dependencies deduced from message exchanges, e.g., network communications. This component is actually in charge of the building of the dependencies among local contextual event causal dependency graphs. This component will be further described in Section 5.3.3. The *message exchange transform component* is also stateful. It also has to remember send and receive events to build causal dependencies related to message exchanges. It also publishes to the topic L.

**Load Component.** All the contextual events and causal dependencies computed by the various transform components are eventually published to the topic L, which is dedicated to the load component (called *Graph DB Connector* in Figure 5.3). The subscription to this topic enables the load component

to import the computed contextual events (CECD nodes) and causal dependencies (CECD edeges) into the database using a bulk import strategy.

### Graph database

Given the acyclic oriented graph structure property of the three relationships we defined in Chapter 4, we naturally settled on a graph database to benefit from its efficiency to recover graph structures via a graph-oriented dedicated query language. Such graph-oriented query language enables the writing of simple graph traversal queries. This corresponds to an important database choice criteria as we are interested in computing cause and dependence graphs of events of interest.

In order to properly investigate events and attacks, a cyber defense analyst also needs to be able to *search* through the event database conveniently. This means that the chosen database is tailored for search and retrieval, instead of data update. In fact, once events are stored in the database, they are very likely to be read-only. For example, this requirement is well illustrated by elastic search[3], a document-oriented database, which is one of the main choices when building an open-source SIEM.

Additionally to the graph and search properties needed, the chosen database needs to adapt to the number of events generated by the monitoring systems. Thus, the database has to be scalable.

To satisfy all these requirements, we have chosen ArangoDB, a distributed graph database. ArangoDB has the characteristic to be multi-model, i.e., it supports key-values, document, and graph data models. Moreover, it also has the capability to scale to a large number of events by easily adding machines to the database cluster. Its query language allows a cyber defense analyst to explore events, as they are contained in the computed contextual events, and compute the cause and dependence graphs of interesting events by performing graph traversals of the contextual event causal dependency graph.

### Visualization Interface

In order to investigate events, a cyber defense analyst has to be able to use the contextual event causal dependency graph by requesting for cause and dependence graphs. The graph database we chose already has a visualization interface, however, its visualization tools are not expressive enough to make use of it. We have thereby worked on a simple interface to enable the visualization of cause and dependence graphs.

Figure 5.4 illlustrates the visualization interface we developed. It is made up of three main parts:
1. the visualization of the graph on the left side [1]. The graph's legend is displayed at the bottom of the visualization interface. This visualization displays all the contextual events, as well as their causal dependencies, contained in the requested *cause* and *dependence* graphs. In the example illustrated in the figure, the different colors correspond

---

3 https://www.elastic.co/products/elasticsearch
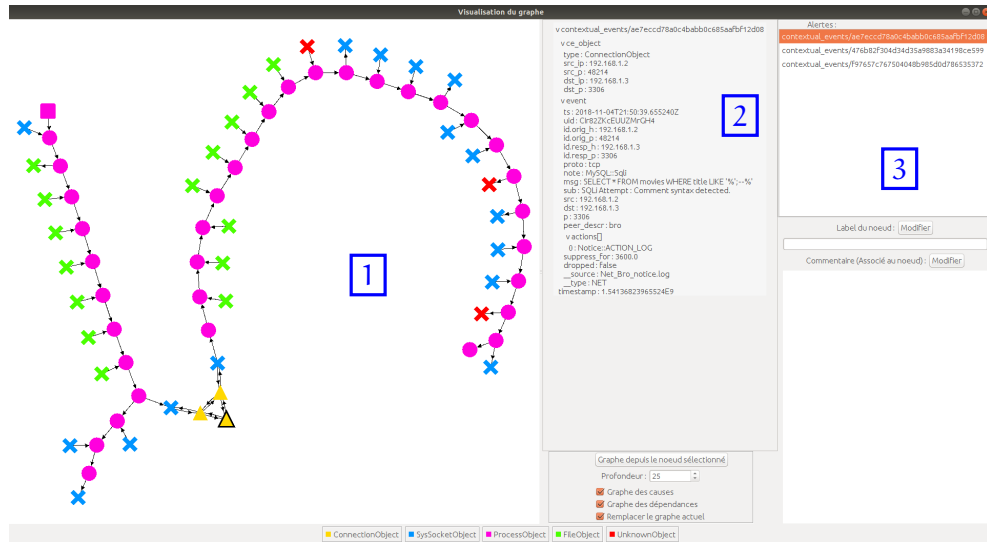
**Figure 5.4:** Visualization interface for contextual event causal dependency graph analysis.

to different types of objects. The yellow triangles represent network-related contextual events. The purple circles represent process-related contextual events, either computed from system call invocation events, or from application events. Blue crosses represent socket-related contextual events. Red crosses represent unknown objects, i.e., file descriptors that could not be identified because the information was missing. And green crosses represent file-related contextual events;

2. the display of the selected contextual event's information in the middle [2]. More specifically, all information contained in the related event is displayed;

3. the display of the events of interest, i.e., alerts in our case, on the right side [3].

Using this interface, a cyber defense analyst can interact with the graph database and easily request for the contextual event causal dependency *cause* and *dependence* graphs of a contextual event of interest.

The work presented in this manuscript does not focus on visualization. Visualization for Cybersecurity is research field in itself. This is well illustrated by the IEEE VizSec[4] conference.

———

The previous section presented the overall design of our implementation. The following section focuses on the computation of the so-called contextual events. More specifically, it presents how contextual events and their related causal dependencies are guessed from the *raw* events produced by the deployed monitoring systems.

———

4 https://vizsec.org/

### 5.3.3 Computing the Contextual Event Causal Dependency Model from Raw Events

Section 5.3.3 describes how each type of event is treated, depending on its data source, to compute contextual events and deduce their causal dependencies. The following sections describe how we deduce contextual events issued from the analysis of: (1) application-related data sources, i.e., application logs and application-level HIDS alerts; (2) system call-related logs and HIDS alerts; and (3) events deduced from network traces or NIDS alerts. Finally, the last paragraph of Section 5.3.3 illustrates the fact that, building upon contextual events computed from raw events, corresponding contextual actions can only be approximated.

*Application Data Source*

**Contextual Events from Application-Level Events.** A process, $p$, running an application, produces a sequence of application events, $[e_i^{App}]$, where the event $e_i^{App}$ describes an action, or a change of state, executed by the observed process. In order to compute the related contextual event of a given raw event, we first need to identify the observed object and its timestamp. These information can generally be retrieved from the event semantics or from an external knowledge base. We can thereby deduce a sequence of the related contextual events, $[(e_i^{App}, p, t_i)]$, where $p$ corresponds to the observed process and $t_i$ corresponds to the recording time of $e_i^{App}$.

Depending on the deployed monitoring system, these events can be analyzed by an application-level HIDS to detect suspicious program behavior, e.g., an abnormal sequence of events. The HIDS analysis uses events from the monitored application; thus, any raised alert is related to the application context. An application-level contextual event, $(e_{alert}^{App}, p, t_{alert})$, can be deduced from a given application-level alert, $e_{alert}^{App}$; with $(e_{alert}^{App}, p, t_{alert}) \in Obs((a, (p, t)))$, $(a, (p, t))$ being the contextual action that is actually observed.

**Example 5.1** *Example of an Apache event*

```
[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 5.1, corresponds to the Apache log entry that records the fact that the request has been treated.

The Apache logging system has been specially configured to output this event format, i.e., a format that contains enough information to deduce the monitored *active* object. In fact, we have configured the Apache logging system to suit our information needs. Another way to enhance the semantic information contained in produced events is to leverage dedicated monitoring modules. For example, such modules can enable the recording of POST requests' parameters.

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 event, i.e., the monitored object. Thus, a contextual event can be computed, by a *CEs & CDs transform component*, by identifying the corresponding Apache active object using the PID information. This *transform component* is stateless. Then, the contextual event can then be placed on its timeline by the CECD graph builder, using the timestamp, $t_{e^{App}_{Req}}$.

In fact, 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, it is necessary to consider the following approximation: the two actions happened at the same time (i.e., the event's timestamp). Moreover, $e^{App}_{Req}$ could also be considered as message exchanges: (1) the reception of a message, i.e., the POST request; and (2) its reply.

The following two paragraphs further illustrate how application events can be leveraged to deduce sessions and message exchanges when possible. In our case, we would need to instrument further the Apache web server. Thus, we did not implement the methodologies presented in these two paragraphs.

**Sessions in Application Contextual Events.** According to the semantic information contained in the application-level events, sessions can be deduced from them.
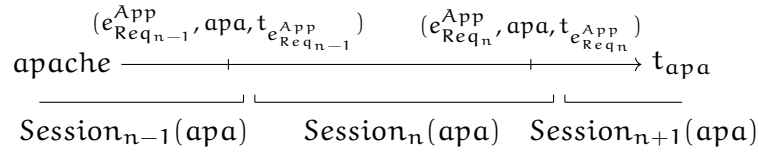


**Figure 5.5:** Apache application contextual event computed from `access.log`.

Figure 5.5 shows the placement of the contextual application event on the Apache object timeline. Considering that Apache is a server with no internal memory between requests, i.e., the beginning of each request processing is independent of the others. In other words, each request would be executed in an independent session, as defined in Section 4.3.1. . Note that the Apache application could be further instrumented for the accurate determination of sessions at the application-level. With two events shown in the example, three sessions are illustrated.

**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 events. This is, for example, the case for the approach presented in [Lanoë et al., 2019] (presented in Section 1.4.5).

Figure 5.6 illustrates the use of application-level contextual events (when they can be deduced from *raw* events) to compute causal dependencies between the two process active objects. Note that the clocks of the two pro-
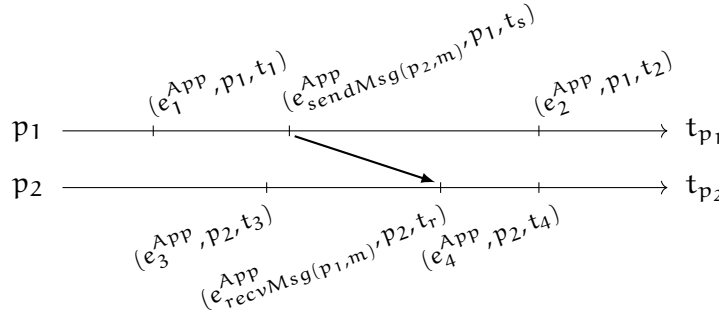
**Figure 5.6:** Message exchange between two applications.

cesses, $t_{p_1}$ and $t_{p_2}$, are not necessarily synchronized. Process $p_1$ records in its log a contextual event, $(e^{App}_{sendMsg(p_2,m)}, p_1, t_s)$, indicating that it sends a message $m$ to $p_2$ at time $t_s$ with the $sendMsg$ function. Process $p_2$ records $(e^{App}_{recvMsg(p_1,m)}, p_2, t_r)$, which indicates that it receives message $m$ from $p_1$ at time $t_r$ with the $recvMsg$ function.

This implies that both contextual events are causally dependent, i.e.: $(e^{App}_{sendMsg(p_2,m)}, p_1, t_s) \rightharpoonup (e^{App}_{recvMsg(p_1,m)}, p_2, t_r)$, in the sense of Lamport. Therefore, we also have $e^{App}_{sendMsg(p_2,m)} \triangleright e^{App}_{recvMsg(p_1,m)}$. 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:

- $e^{App}_1 \triangleright e^{App}_{sendMsg(p_2,m)} \triangleright e^{App}_2$;
- $e^{App}_3 \triangleright e^{App}_{recvMsg(p_1,m)} \triangleright e^{App}_4$;
- $e^{App}_1 \triangleright e^{App}_{sendMsg(p_2,m)} \triangleright e^{App}_{recvMsg(p_1,m)} \triangleright e^{App}_4$.

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, for example.

### System Call Data Source

**Contextual Events from System Call 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 used to record system call invocations. In this trace, contextual events are in the form of $(e^{Sys}, p, t)$; 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 of the actual time of the system call invocation. In that sense, the event $e^{Sys}$ is not produced by process $p$ but is part of the set of contextual events of $p$.

Among the invoked system calls, some produce information flows (e.g., `read()`, `write()`, `send()`, and `recv()`) and others do not (e.g., `wait()`, `mpro-`

`tect()`, and `futex()`). We configured auditd to record all system call actions that produce information flows. The list of the monitored system calls is detailed in Appendix A (Section A.1). 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). Accordingly, we can deduce two causally dependent contextual events from a *raw* event, denoted $e^{Sys}$, which corresponds to a system call that produces an information flow.
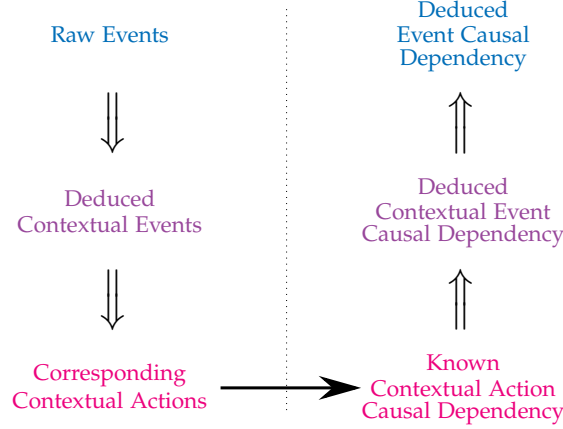


**Figure 5.7:** Bottom-up strategy rationale for system call events.

Regarding system call events, the bottom-up strategy rationale is illustrated in Figure 5.7. Contextual events can be deduced from raw events. Based on the model we defined in Chapter 4, deduced contextual events correspond to the observation of contextual actions. Even if we cannot determine the corresponding contextual actions, we know that, in the context of system call events, the corresponding contextual actions are causally dependent, in the sense of "$\mapsto$," as monitored system calls produce information flows. Therefore, as we previously explained in Chapter 4 with Figure 4.6, we can deduce that the contextual events are causally dependent in the sense of "$\rightharpoonup$." Finally, we can deduce causal dependencies among events. More specifically, depending on the family of the observed system call, the bottom-up strategy rationale is the following:

**Write() System Call Family.**

1. From $e^{Sys}$, we deduce two contextual events: $(e^{Sys}, p, t_{e^{Sys}})$ and $(\emptyset, o, t_{e^{Sys}})$, with $t_{e^{Sys}}$ the timestamp of the event $e^{Sys}$;
2. According to the causality model we defined in Chapter 4, there exist two contextual actions $(write(), (p, t))$ and $(\emptyset, (o, t))$, with $t$ the actual time of the information flow, such that: $(e^{Sys}, p, t_{e^{Sys}}) \in Obs((write(), (p, t)))$ and $(\emptyset, o, t_{e^{Sys}}) \in Obs((\emptyset, (o, t)))$;
3. If the system call is from the `write()` family, then, it produces an information flow from the *active* object, p, i.e., the process that invoked the system call, to the *passive* object, o.
   Thus, we know that the two objects' states are causally dependent and we have: $(write(), (p, t)) \mapsto (\emptyset, (o, t))$;
4. This implies that $(e^{Sys}, p, t_{e^{Sys}}) \rightharpoonup (\emptyset, o, t_{e^{Sys}})$.

**Read() System Call Family.**

1. From $e^{Sys}$, we deduce two contextual events: $(e^{Sys}, p, t_{e^{Sys}})$ and $(\emptyset, o, t_{e^{Sys}})$, with $t_{e^{Sys}}$ the timestamp of the event $e^{Sys}$;

2. According to the causality model we defined in Chapter 4, there exist two contextual actions $(\texttt{read}(), (p, t))$ and $(\emptyset, (o, t))$, with $t$ the actual time of the information flow, such that:
$(e^{Sys}, p, t_{e^{Sys}}) \in Obs\big((\texttt{read}(), (p, t))\big)$ and $(\emptyset, o, t_{e^{Sys}}) \in Obs\big((\emptyset, (o, t))\big)$;

3. If the system call is from the $\texttt{read}()$ family, then, it produces an information flow from the *passive* object, $o$, to the *active* object, $p$, i.e., the process that invoked the system call.
Thus, we know that the two objects' states are causally dependent and we have: $(\emptyset, (o, t)) \mapsto (\texttt{read}(), (p, t))$;

4. This implies that $(\emptyset, o, t_{e^{Sys}}) \rightharpoonup (e^{Sys}, p, t_{e^{Sys}})$.

**Fork() System Call Family.**

1. From $e^{Sys}$, we deduce two contextual events: $(e^{Sys}, p_{parent}, t)$ and $(\emptyset, p_{child}, t_{e^{Sys}})$, with $t_{e^{Sys}}$ the timestamp of the event $e^{Sys}$;

2. According to the causality model we defined in Chapter 4, there exist two contextual actions $(\texttt{fork}(), (p_{parent}, t))$ and $(\emptyset, (p_{child}, t))$, with $t$ the actual time of the information flow, such that: $(e^{Sys}, p_{parent}, t) \in Obs\big((\texttt{fork}(), (p_{parent}, t))\big)$ and $(\emptyset, p_{child}, t_{e^{Sys}}) \in Obs\big((\emptyset, (p_{child}, t))\big)$;

3. If the system call is from the $\texttt{fork}()$ family, then, it produces an information flow from a parent process, $p_{parent}$, that invoked the system call, to a child process, $p_{child}$.
Thus, we know that the two objects' states are causally dependent and we have: $(\texttt{fork}(), (p_{parent}, t)) \mapsto (\emptyset, (p_{child}, t))$;

4. This implies that $(e^{Sys}, p_{parent}, t) \rightharpoonup (\emptyset, p_{child}, t)$.

**Example 5.2** *Example of a system call event produced by auditd*

```
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
type=PROCTITLE msg=audit(1541366508.539:47875): proctitle=2F7573722F7362
    696E2F61706163686532002D6B007374617274
```

Linux Kernel system call events produced by auditd are made up of several entries. Example 5.2 illustrates an $\texttt{accept}()$ system call, which corresponds to the number 288, that produces two contextual events: one for the Apache process and another for the created socket. This event is made up of three entries:

1. The SYSCALL entry records the system call information, i.e., its parameters as well as its output. In more detail, the event semantics depends on the nature of the system call. In this case, for example, the system call returns the number of the file descriptor ($\texttt{exit}=10$) created to reference the network socket.

2. The SOCKADDR entry describes the socket information: "saddr=(AF _INET) 10.0.2.2:56582." This information corresponds to the IP and port of the remote machine. They will be completed with the information contained in the netfilter events, as described in Section 5.3.3 in the *Network Socket Object Identification* paragraph.

3. The PROCTITLE entry "records the full command-line of the command that was used to invoke the analyzed process."[5]

The accept() system call is from the read() system call family. Thus, the Apache process' contextual event is causally dependent on the contextual event of the created socket.

**Approximating the Contextual Event Causal Dependency Model.** 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.

**Stateful Transform Component.** The *CEs & CDs transform component* that handles Linux Kernel system call events is stateful. In fact, the Linux Kernel handles I/O resources with *file descriptors* instead of file paths, for instance. More specifically, the Kernel maintains a file descriptor table for each process. Processes operate with them, through system call invocations, to read from or write to their related I/O resources. As we do not have access to these file descriptor tables, we have to rebuild them from the recorded information contained in system call events. Such a processing step is mandatory to track the flows of information among objects. To implement this requirement, we leveraged the system call handling component of SPADE[6].
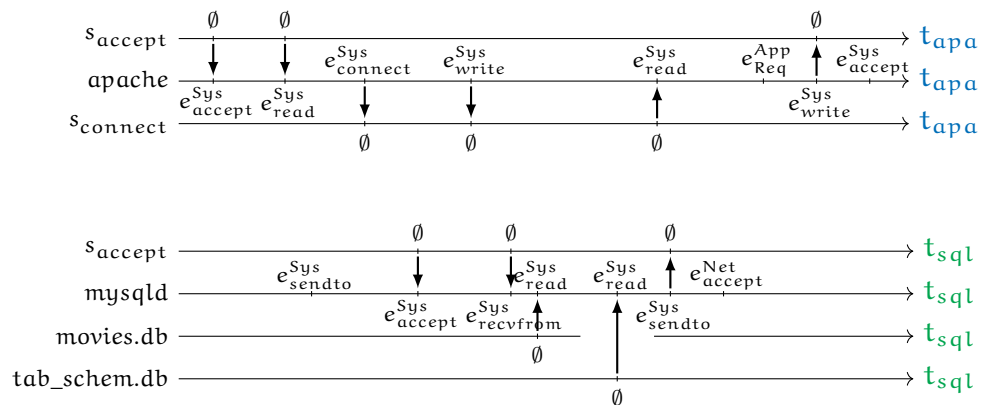


**Figure 5.8:** System call contextual events computed from audit logs.

5 https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/security_guide/sec-understanding_audit_log_files

6 https://github.com/ashish-gehani/SPADE

Following the illustration of application event processing in our attack scenario example, we now illustrate system call event processing. As can be observed in Figure 5.8, system call contextual events are not sufficient to link the Apache and MySQL hosts. To make them readable, the contextual events are not written in figures but are replaced by their corresponding events. In Figure 5.8, 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.

Similarly to the application-level HIDS, and any triggered alert is also considered to be part of the application context. Accordingly, a system call-related contextual event, $(e_{alert}^{Sys}, p, t_{alert})$, can be deduced from the system call-related alert, $e_{alert}^{Sys}$, triggered in the context of $p$ at time $t_{alert}$; with $(e_{alert}^{Sys}, p, t_{alert}) \in Obs((a, (p, t)))$, $(a, (p, t))$ 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.

**Sessions in System Call Contextual Events.**   As mentioned in the application data source section, 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 $apache$ processes.

The system call abstraction layer has been used to indicate sessions in many approaches presented in Section 3.4.4, i.e., approaches that leverage software testing techniques and binary instrumentation to determine sessions and enable their identification in the produced events.

### Network Data Source

**Contextual Events from Network–Level Events.**   Network-level events[7], 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 in the form $(e_{conn}^{Net}, netw, t)$, which means that a network-level event, $e_{conn}^{Net}$, 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 produce actions, i.e., the sending and reception of messages over the physical network layer. Thus, it can be considered

---

7 We refer to log entries deduced from network traces as network-level events.

as a *process in the sense of Lamport's Happened-Before relationship* defined in Section 3.2.3. Messages are transmitted to the OS layer through the usage of sockets.

Additionally, network interfaces can observe, analyze, and record packet flows, and are considered to be stateless, i.e., 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-level event, $e_{conn}^{Net}$, 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. $(\emptyset, socket_{conn}^{src}, t) \rightharpoonup (e_{conn}^{Net}, netw, t)$ if the information flows from the passive object (the socket) to the active object (the network interface, $netw$);

2. $(e_{conn}^{Net}, netw, t) \rightharpoonup (\emptyset, socket_{conn}^{dst}, t)$ 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 Zeek NIDS, which generates event logs from the protocol dissection. A NIDS alert, $e_{conn,alert}^{Net}$, is modeled like any other network-level event, i.e., a contextual event in the form of $(e_{conn,alert}^{Net}, netw, t_{alert})$; with $(e_{conn,alert}^{Net}, netw, t_{alert}) \in Obs((a, (netw, t)))$, $(a, (netw, t))$ being the contextual action that triggered the alert. Because Zeek can generate several events from the same connection, it well illustrates the fact that an action can be observed as several events, i.e., the set $Obs((a, (netw, t)))$ of its related contextual action, $(a, (netw, t))$, can contain several contextual events: $Obs((a, (netw, t))) = \{(e_{i_{conn}}^{Net}, netw, t_{e_i})\}$. Moreover, these 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 5.3** *Example of Zeek NIDS 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']
```
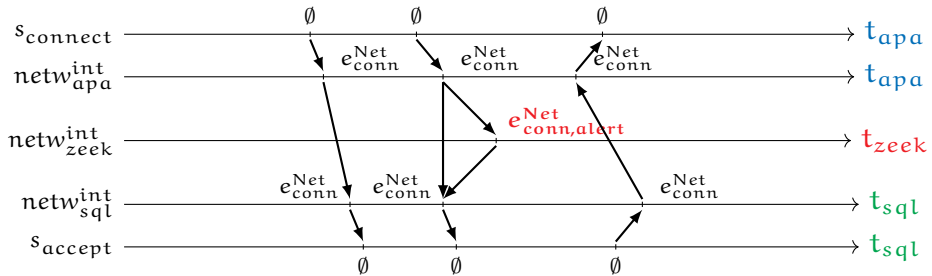


**Figure 5.9:** Network-related contextual events computed from netfilter and Zeek NIDS logs.

Figure 5.9 illustrates the use of contextual events deduced from network traces in our motivating example. Only the internal network side has been shown in this example, i.e., $netw_{apa}^{int}$, $netw_{sql}^{int}$, and $netw_{Zeek}^{int}$ network interface active objects. The event, $e_{conn,alert}^{Net}$, illustrated by Example 5.3, corresponds to a Zeek 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.

Similarly to the *Apache application event transform component*, the *Zeek transform component* is stateless.

**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 $\{ip_{remote}, p_{remote}\}$, where $ip_{remote}$ and $p_{remote}$ respectively correspond to the IP address and port of the remote host, 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 $\{ip_{src}, p_{src}, ip_{dst}, p_{dst}\}$. 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 $\{ip_{remote}, p_{remote}\}$ with the connection information:

1. for incoming connections, i.e., sockets created by the `accept()` system call family, $\{ip_{remote}, p_{remote}\} = \{ip_{src}, p_{src}\}$;
2. for outgoing connections, i.e., sockets created by the `connect()` system call family, $\{ip_{remote}, p_{remote}\} = \{ip_{dst}, p_{dst}\}$.

This matching allows the acquisition of all information required to describe a network socket. It is done by a stateful transform component as it needs to remember system call-related sockets and network-related sockets that did not match yet. Note that this matching does not rely on precise timestamp matching, i.e., the two events' timestamps may differ by a few milliseconds.

**Message Exchanges in Network–related Contextual Events.** Network-related 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., $\{ip_{src}, p_{src}, ip_{dst}, p_{dst}\}$, the nature of the message can easily be identified: whether it corresponds to a sending or a reception.

Given a connection, the two socket objects can easily be matched using their connection information. This matching is done by the second component of the transform part of the ETL pipeline (as described in Figure 5.3). This component is stateful, similarly to the one presented in the *Network Socket Object Identification* paragraph.

The example in Figure 5.10 illustrates the use of network level contextual events to compute causal dependencies utilizing the message exchange part of definition 4.4. Note that the clocks of the three objects ($t_{apa}$, $t_{zeek}$, and

$$netw_{apa}^{int} \xrightarrow{\quad (e_{out(conn)}^{Net}, netw_{apa}, t_s) \quad} t_{apa}$$

$$netw_{zeek}^{int} \xrightarrow{\quad (e_{conn,alert}^{Net}, netw_{zeek}, t_{alert}) \quad} t_{zeek}$$

$$netw_{sql}^{int} \xrightarrow{\quad (e_{in(conn)}^{Net}, netw_{sql}, t_r) \quad} t_{sql}$$
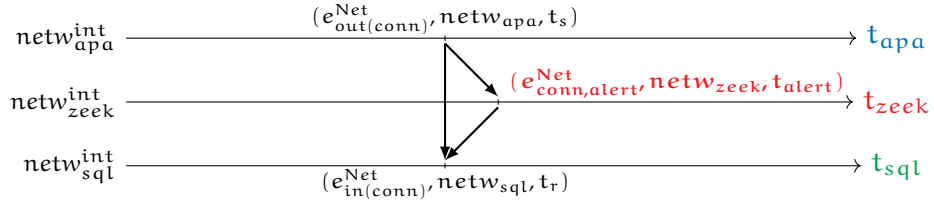
**Figure 5.10:** Message exchange among network objects.

$t_{sql}$) 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 Zeek 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 $(e_{out(conn)}^{Net}, netw_{apa}, t_s) \rightharpoonup (e_{in(conn)}^{Net}, netw_{sql}, t_r)$.

### From Contextual Events to Contextual Actions—An Approximation

In practice, the bottom-up strategy rarely permits the generation of real contextual actions from the system observations (e.g., there is no system implementation that enables the observation of object states and computation of causal dependencies among contextual actions). Indeed, most of the time, events do not contain the overall context in which these actions are performed, i.e., the object states. Thus, we can only produce an approximation of contextual actions from contextual events. A contextual action is surmised from a set of events that correspond to the observations and recordings of this single action by several monitoring systems. 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 one that produced the set of events. To build this set, it is necessary to preprocess contextual events to aggregate the ones that correspond to the same action into a single set.

---

The previous section presented our implementation of the bottom-up strategy in the context of Linux environments. The following section describes how we apply this strategy in the windows environment.

## 5.4 VESTA INDUSTRIAL PROJECT

This section introduces the reader to the project called VESTA, which partly supported the research work presented in this manuscript.

### 5.4.1 Introduction to the VESTA Endpoint Detection and Response

VESTA stands for Vigilant Endpoint Security Tools and Agents. This project consists in the development of an Endpoint Detection and Response (EDR) product for information systems and industrial control systems, as well as the development of its operational center. In simple terms, VESTA follows a master-slave architecture where the agents execute actions and commands led by the operational center, e.g., producing logs that are collected by the operational center.

*Project's Objectives*

VESTA aims to enable:
- the enumeration of the monitored endpoints in the computer network;
- intrusion detection capabilities based on misuse-based and anomaly-based techniques. These detections can be done either online or offline;
- threat hunting capabilities by scanning file systems for given IoCs;
- forensics capabilities by enabling memory and disk dump or collecting events of interest;
- response actions led by cyber defense analysts;
- and the bridging of the gap between intrusion detection research and operational intrusion detection domains.

The initiative of the VESTA EDR development is based on the fact that solely monitoring the network, via network sniffing tools such as NIDS, is not sufficient. Additionally, it aims to propose a French sovereign product solution to address the security monitoring of French operators of vital importance (*Opérateur d'Importance Vitale in French*).

### 5.4.2 Bottom-Up Strategy—Leveraging Windows System Calls

VESTA aims to be multi-platform. It has firstly been developed for the Windows OS. As VESTA does not aim to instrument applications, nor the Windows OS itself, we based our work on events that can natively be produced by Windows, i.e., system call events recorded via the event tracing windows facility. Similarly to our approach for the Linux Kernel, We adopted a bottom-up strategy to implement our work in the project.

*Windows' System Calls and Information Flows*

In practice, the analysis of Windows system call events corresponds to the implementation of an information flow monitor for the Windows OS. However, it is simpler on Windows as system calls are already categorized into:
- process-related operations, e.g., the creation of new processes;

- image loading operations, e.g., initializing applications' contexts or loading Dynamic-Link Libraries (DLLs) at run time;
- file system operations on artifacts, e.g., read or write a file;
- network operations, e.g., send to and receive from a TCP or UDP communication.

Similarly to the case of the Linux Kernel, Windows Kernel system calls allows cyber defense analysts to deduce information flows among objects of the OS's Kernel abstraction layer. As we have previously presented in Chapters 1 and 3, Windows system calls can be monitored by ProcMon[8] or logman, which both leverage the ETW facility.

Contrary to the *Linux Kernel system call transform component*, the one that handles Windows Kernel system call does not need to be stateful. In fact, recorded Windows system call events contain more information than the Linux ones, e.g., they relate file paths in each system call. Therefore, passive objects can directly be identified with the information contained in the system call events.

### *Operational Constraints*

Contrary to the setup presented in the previous section, the monitoring strategy adopted in VESTA is more restrictive. The monitoring of system calls is by default disabled. When analyzing a triggered HIDS alert, the cyber defense analyst can subsequently decide to enable to monitoring of system calls and collect them. The consequence of such choice is that it prevents the building of cause graphs. Moreover, the building of the dependence graph is also harder as some important events might be missed between the effective action that triggered the alert and the system call invocations recorded at the start of the monitoring process. In this context, we can therefore only compute a degraded dependence graph at best.

Due to time constraints, we did not have access to data produced by the project. We thereby conducted our own experiments. These experiments are presented in the following chapter (Chapter 6).

---

8 https://docs.microsoft.com/en-us/sysinternals/downloads/procmon

## 5.5   SUMMARY

Building upon Chapter 4, this chapter further explains the three relationships we defined and shows how they can be implemented.

**Implementation Strategies of our Model: Top-Down and Bottom-Up.** Section 5.1 introduces the reader to the two implementation strategies that can be adopted to compute the event causal dependency relationship, namely, the *top-down* and *bottom-up* strategies. The first one is based on the computation and tracking of contextual action causal dependencies. These causal dependencies among contextual actions are then leveraged to compute causal dependencies among contextual events, and events. The second strategy consists in leveraging raw events' semantics to compute the contextual event causal dependency model. More specifically, the bottom-up strategy allows obtaining an approximation of the contextual event causal dependency model that depends on the semantical quality of events.

Section 5.2 presents how the merging of existing technologies would enable the top-down strategy computation of different parts of the overall model. Of course, other methodologies could emerge to address the implementation of the model. Unfortunately, it is a challenge to gather and integrate all these technologies together to compute contextual actions and their causal dependencies. Therefore, as a first step, we have decided to adopt a bottom-up strategy in order to compute the event causal dependency model.

**COTS-Based Implementation of the Bottom-Up Strategy.** Sections 5.3 and 5.4 describe our implementation of the model using the bottom-up strategy. More specifically, we elaborate a monitoring strategy, based on COTS monitoring systems, to enable the identification of multi-step attack scenarios through the computation of the contextual event causal dependency model. We describe our implementation's architecture, as well as how each data source is handled to compute contextual events and their causal dependencies.

The following chapter presents the assessment of our implementation and discusses the obtained results.

# 6 | ASSESSMENT

This chapter proposes an assessment of the lightweight approach we described in Chapter 5. It aims to verify whether the approach we propose allows the identification and understanding of real attack scenarios. Throughout the chapter, we will discuss the usefulness of our approach and methodology.

Section 6.1 starts by presenting the notions surrounding datasets as they are one of the key assessment elements in the attack scenario identification research field. Section 6.2 discusses an assessment of our lightweight approach implementation. It questions the relevance of the cause and dependence contextual event graphs resulting from our approach and examines the performance of the different components presented in the architecture of our implementation. Finally, Section 6.3 discusses the limitations of our current implementation.

## 6.1 BUILDING DATASETS TO ASSESS OUR APPROACH

As we have seen in the previous chapters of this manuscript, attack scenario identification approaches mainly rely on log analysis. More specifically, they aim to help cyber defense analysts investigate and recover from attacks by retrieving all the events corresponding to the projection of an attack scenario on the monitored system.

This section introduces the reader to the assessment context of our approach and implementation presented in the two previous chapters. It starts with the presentation of the ideal dataset for the attack scenario identification research field in Section 6.1.1. Then, Section 6.1.2 presents a few publicly available datasets and explains why we need to build our own testing environment to generate a dataset that suits our needs. Section 6.1.3 introduces the reader to the challenges of building an ideal dataset. Section 6.1.4 describes our testing environment, i.e., its architecture and the deployed monitoring systems. Finally, Section 6.1.5 describes the attack scenarios we performed in our testing environment.

### 6.1.1 Describing an Ideal Dataset

Naturally, the attack scenario identification research community needs to have comprehensive datasets to assess the proposed approaches and allow their comparison. Unfortunately, to the best of our knowledge, datasets fitting our needs are not publicly available. The following paragraphs introduce the reader to the concepts surrounding the building of datasets and its challenges.

**Needed Data is Specific to the Proposed Approach.** As we have seen in the first part of this manuscript (Part I), chosen attack scenarios identification methodologies cannot be dissociated from the security monitoring strategy. In fact, the majority of approaches presented in the first part and Chapter 3 only leverage a single type of event. For instance, some approaches presented in Chapter 2 exclusively leverage NIDS alerts, and other approaches, presented in Chapter 3 exclusively leverage system call-related events (Section 3.3.3). On the other hand, our approach aims to include all types of events as they may contain valuable information for attack investigation.

**Contextual Actions and Datasets.** From the perspective of the causal dependency model we defined in Chapter 4, we have seen that contextual events correspond to the observation of contextual actions in the monitored system. A dataset thereby represents a given perspective of the activity, i.e., a perspective of the contextual actions that happened in the monitored system in a given time window. According to the adopted monitoring strategy, this dataset is more or less detailed for the different abstraction layers of the monitored system.

**Ideal Dataset.** The following points describe the general characteristics of an ideal dataset for the assessment of attack scenario identification approaches:

1. The dataset is made up of heterogeneous events emanating from intrusion detection systems as well as general-purpose monitoring systems. In fact, a multi-step attack likely involves steps that leave traces in many places in the monitored system: application-related activity, OS-related activity, or network communication-related activity. Thus, understanding a multi-step attack likely involves the analysis of logs produced at these various levels.

2. The dataset contains the traces of one multi-step attack, at least. The attack ideally involves lateral movements of the attacker throughout the monitored system, i.e., pursuing his attack, the attacker accesses, and exploit different machines of the monitored system.

3. The dataset contains noise, i.e., events generated by the normal activity happening in the monitored system. Such events may correspond to false positive alerts triggered by legitimate activities. Noise makes the assessment more realistic than only having the events corresponding to the multi-step attack.

### 6.1.2 On the Lack of Publicly Available Comprehensive Datasets

**On Public Datasets.** Unfortunately, to the best of our knowledge, a dataset containing all the various types of events we need is not publicly available. The following list presents some of the publicly available datasets and explains why we cannot leverage them to evaluate our methodology and implementation.

DARPA 2000 – This dataset has been produced and released by DARPA in 2000[1]. It contains full packet captures (*network-related* data) and Solaris BSM audit events (*system call-related* data). Additionally, it contains traces of multi-step attacks, as well as an explanation of the attack, i.e., the ground truth, that were performed against the monitored system by a redteam. Even if this dataset is made up of two interesting types of events, recorded system call data does not contain enough information to suit our needs and build the contextual event causal dependency model. More specifically, our approach relies on the capture of all information flow-related system calls, and the BSM audit events do not satisfy this requirement.

LANL – The Los Alamos National Laboratory (LANL) proposes three datasets produced in the context of their internal enterprise network:

The "User-Computer Authentication Associations in Time"[2] dataset, which is made up of authentication events. This dataset is made up of one type of event that does not allow to compute causal dependencies. Thus, we cannot leverage it. Moreover, we do not know if it contains attack traces or not.

The "Comprehensive Multi-Source Cyber-Security Events"[3] dataset. It is made up of five types of events, namely, Windows-based authentication events, process start and stop events from Windows workstations, Domain Name Server (DNS) lookups from the LANL internal DNS servers, netflow events, and attack-related netflow events. The attacks were performed by a redteam. Even if this dataset is made up of heterogeneous types of event, the related monitoring strategy that produce it does not suit our needs to adopt the bottom-up strategy. Recall that we need information flow-related data to build the contextual event causal dependency model for monitored hosts. However, this dataset does not provide information flow-related data in the host-related events, i.e., process start and stop, and authentication. Netflow events can be used to build message exchange related causal dependencies among hosts. However, a contextual event causal dependency model that only relies on netflow events would be too coarse-grained to get insight from it.

The "Unified Host and Network Dataset,"[4] which is made up of 21 types of events, i.e., netflow events, and 20 different types of Windows event logs. Similarly to the previous dataset, selected event types do not allow to apply the bottom-up strategy to compute the contextual event causal dependency model. Additionally, we do not know if it contains attack traces or not. Thus, we cannot leverage it.

---

1 https://www.ll.mit.edu/r-d/datasets/2000-darpa-intrusion-detection-scenario-specific-datasets
2 https://csr.lanl.gov/data/auth/
3 https://csr.lanl.gov/data/cyber1/
4 https://csr.lanl.gov/data/2017.html

**VAST CHALLENGE 2011 MC2 –**  This dataset has been produced for the Visual Analytics Science and Technology (VAST) Challenge of 2011[5]. It represents the activity happening in a four days time window inside a fictional shipping company. This dataset is made up of several types of events, i.e., firewall logs, NIDS alerts, and Windows security event logs. It also contains traces of attacks. However, the issues related to this dataset are the same as the LANL "Comprehensive Multi-Source Cyber-Security Events" dataset. Only network-related events can be leveraged. Thus, computed contextual event causal dependency model would be too coarse-grained to get insights from its analysis.

**DARPA TRANSPARENT COMPUTING –**  This dataset has been produced by DARPA in the context of the Transparent Computing program presented in Chapter 1[6]. It has been released in late 2018. It is made up of system call-related events from the Linux, FreeBSD, and Windows Kernels recorded by COTS monitoring systems, i.e., auditd, DTrace, and Event Windows Tracing, respectively. These events have been produced in DARPA's test environment and contain attack traces, as well as normal activity (i.e., activities corresponding to noise). Even if this dataset only contains one type of events, i.e., information flow-related system call events, this type of events allow us to apply the bottom-up strategy for the computation of the contextual event causal dependency model. The only weakness of this dataset is that it does not contain different types of events, e.g., network-related and application-related events.

As we have started to discuss at the end in Section 2.4.2, we argue that exclusively reasoning on publicly available datasets may constrain researchers' creativity and invite them to propose approaches that only work with the data available in these datasets, e.g., the majority of alert correlation approaches that only rely on NIDS alerts.

**The Need to Build our Own Datasets.**  As we have seen in Chapter 5, our approach adopts a *bottom-up strategy*. It thereby relies on the analysis of specific types of events that enable us to compute contextual events and their causal dependencies. This implementation, presented in Section 5.3, relies on application, system call, netfilter and network-related events. More specifically, it relies on the capability to compute contextual events and their causal dependencies from raw events. Thus, we have to build our own datasets to evaluate the implementation of our model.

### 6.1.3  On the Difficulty to Build a Comprehensive Dataset

Based on the observation that a publicly available dataset that suits our needs is lacking, we need to build our own test environment to produce datasets. Such a testing environment generally corresponds to a classical

---

5 http://www.cs.umd.edu/hcil/varepository/VASTChallenge2011/challenges/MC2-Comput erNetworkingOperations/

6 https://github.com/darpa-i2o/Transparent-Computing

enterprise computer network. However, it is generally built-in the context of a laboratory platform to safely play attack scenarios that may involve the execution of malwares. As a consequence, testing environments rarely involve real users, and generating normal activity and noise is a challenge.

––––––––––––––

Previous sections introduced the reader to one of the main assessment's requirements, namely, datasets. It presented the ideal dataset and explained why we need to build our own testing environment to generate datasets that suit our needs. The following section presents our testing environment.

### 6.1.4  Our Test Environment

Section 6.1.4 describes the environment we have built- to generate the activity and the events needed to assess our implementation and methodology.

#### *Building and Sharing our Test Environment*

**Dataset's Context.**   In order to allow the comparison, or the simultaneous usage, of approaches that leverage different types of events, we argue that the *context* in which datasets are produced (i.e., the activity of the monitored system during a given time window) needs to be shared. In fact, in the light of the relationships we have defined in Chapter 4, the *context* actually corresponds to the *contextual action causal dependency* model.

**Sharing and replaying scenarios with Moirai.**   Sharing the overall activity of a monitored system relies on the capability to exactly replay a given scenario, made up of normal activity and attack-related activity, in an automated way. In order to satisfy this requirement, Brogi et al. propose Moirai [Brogi & Tong, 2017], an attack scenario sharing, and replaying platform.

**Building environments with Vagrant and Ansible.**   Under the hood, Moirai relies on Vagrant[7] and Ansible[8] to build the environment described in the scenario. Vagrant, coupled with Ansible, enables the automatic deployment of virtual machines, services, and network configurations based on files that describe the environment. Thus, anybody can instantiate our test environment using the description files.

**Playing attack scenarios.**   Additionally, it generates the activity described in an input file. Such activity can, for example, correspond to the execution of a shell script or a binary. Based on the capability to replay attack scenarios, users can, therefore, implement their own monitoring strategy to produce the various types of events they need to perform the approach they propose. Thus, we have decided to leverage Moirai to build our test environment. Moirai allows us to describe the architecture of our test environment, instantiate it using virtualization technology, deploy the monitoring systems we need, and play scripted scenarios.

––––––––––––––

7 https://www.vagrantup.com/
8 https://www.ansible.com/
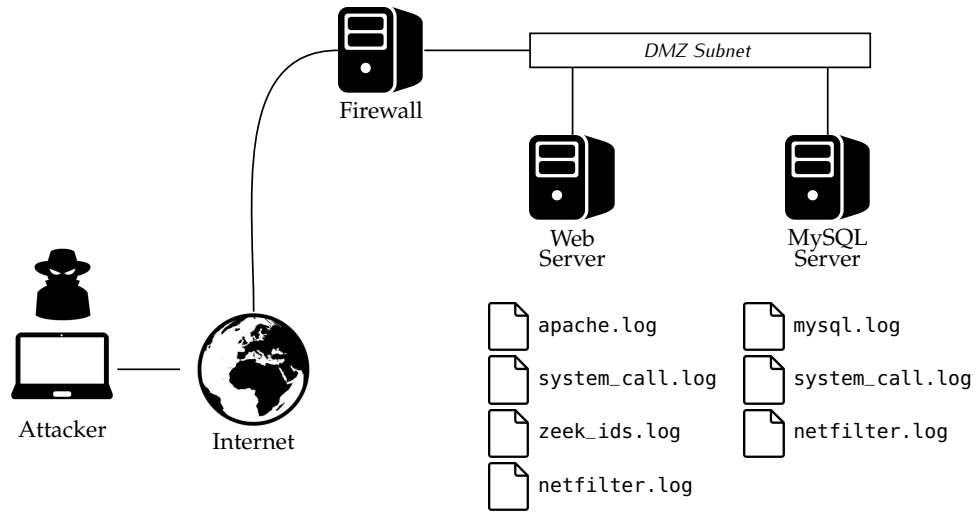
**Figure 6.1:** Network architecture of our test environment.

### Test Environment Description

The current architecture of our test environment is described in Figure 6.1. The current network architecture consists of a demilitarized zone (DMZ) and an internal network. These two subnets are linked by an Ubuntu 14.04 LTS machine, which is in charge of the firewalling, the routing, and giving access to the internet. Two machines are running in the DMZ:

1. An Ubuntu 14.04 LTS running an Apache server version 2.4.7 with the Common Gateway Interface (CGI) module enabled, PHP 5.5 execution environment, and Bash 4.2.37, a version of Bash that is vulnerable to the ShellShock attack (CVE-2014-6271)[9];
2. An Ubuntu 14.04 LTS running a MySQL server version 5.5.62;

On the other hand, the internal network consists in desktop environments. The attack scenarios we have performed to assess our implementation only focus on the machines of the DMZ. Thus, we did not develop the monitoring of desktop environments in the context of this assessment.

### Deployed Monitoring Systems

In order to enable the monitoring of the testing environment we have set up, we have deployed different kinds of monitoring systems that allow the observation of the network, operating system, and application abstraction layers. The monitoring systems we chose produce events that allow us to apply a bottom-up strategy, as described in Section 5.3. Each deployed monitoring system is detailed in the following paragraphs.

**Zeek.** A Zeek NIDS, configured in promiscuous mode, is currently deployed on the $M_{apa}$ host. Ideally, we would have deployed Zeek NIDS on a dedicated monitoring machine that would sniff all the network traffic of the DMZ thanks to the *port mirroring*[10] feature of the DMZ switch. How-

---

9 https://github.com/opsxcq/exploit-CVE-2014-6271

10 A mirror port is a specially configured switch port that gets a copy of all frames that pass through the switch.

ever, the virtualization provider we currently use is Oracle VM VirtualBox[11], and the provided virtual switch implementation does not have the port mirroring feature. That is the reason why we do not have a dedicated network monitoring machine in the DMZ of our test environment. Fortunately, the network monitoring of the $M_{apa}$ host is sufficient to produce the events we need in our scenario.

As we previously mentioned in Section 1.4.3, Zeek produces many types of logs according to the protocols dissected in the analyzed traffic. The base version is already equipped with all the classical protocol dissectors. A few examples of the logs it produces are: `http.log`, which corresponds to the dissection of the HTTP protocol; `mysql.log`, which corresponds to the dissection of the MySQL protocol. Related events contain the invoked command, as well as its result metadata, e.g., whether the command was successfully handled or not, and the number of rows returned; `dns.log`, which corresponds to the dissection of the DNS protocol. Related events contain the requested domain name and the resulting IP addresses; `conn.log`, which corresponds to the dissection of the transport layer. Related events contain network flow statistics, similarly to netflows; and `notice.log`, which corresponds to the alert log file.

Zeek NIDS enables the monitoring of all messages exchanged through the network. When protocol dissection is possible, i.e., when messages are not encrypted, Zeek actually logs some information about the content of the message, as we have presented in the previous paragraph.

In our current implementation, messages are considered to be exchanged through the network between two active objects: a source and a destination network interfaces.

**Netfilter.** Netfilter is configured on all monitored machines to log any "established" connections, either with the TCP or UDP protocols. As we have previously mentioned in Section 5.3.3, produced netfilter events allow to associate network sockets deduced from system calls to their actual related connections (defined by the quadruplet $\{ip_{src}, p_{src}, ip_{dst}, p_{dst}\}$). More specifically, one network socket corresponds to one connection. Netfilter rules are specified in Appendix A (Section A.2).

**Auditd.** Auditd version 3.0 is deployed on all monitored machines. It is configured in a particular manner so that: (1) it enables the monitoring of system calls that produce information flows; and (2) it produces alerts on specifically suspicious actions, e.g., trying to modify auditd configuration, or executing a specified executable such as `whoami`. Auditd rules are specified in Appendix A (Section A.1).

**Apache.** Apache logging system is configured to record any request handling. Additionally, the PID of the process that handled the request is also explicitly recorded. This configuration enables the computation of contextual events from the raw events produced by the logging system. Note that without this specific configuration, it is not possible to deduce the monitored

---

11 https://www.virtualbox.org/

active object from the explicit information contained in the raw event. Apache logging system's configuration is specified in Appendix A (Section A.3).

**MySQL.** Contrary to the monitoring systems previously presented, MySQL logging system cannot be configured to produce events that contain specific information such as the PID of the process that handled the request. Thus, MySQL logging system's events do not provide enough information to compute contextual events, and, as it is, cannot be used for the bottom-up strategy.

### 6.1.5 Attack Scenarios Description

The previous paragraphs presented our testing environment, as well as the deployed monitoring systems that allow us to observe the system and produce heterogeneous events. Section 6.1.5 presents the attack scenarios we have performed in the context of the assessment presented in this manuscript. For each presented attack scenario, we explain how the actions of the attacker are projected on logs and explain how we identify IoCs for the given attack scenario. Two attack scenarios are presented: (1) The first attack scenario corresponds to a sequence of SQL injection against the web application; (2) The second one corresponds to an attack scenario that makes use of the ShellShock vulnerability of the web server.

#### *SQL Injection against the Web Application*

**Attack Scenario Description.** The first one corresponds to the attack scenario we have presented in Chapter 4. It consists in SQL injections that aim to exfiltrate sensitive data contained in the MySQL database. This attack scenario corresponds to the one presented in Chapter 4 (Section 4.1).

**Projection on Logs.** Naturally, any SQL query will be logged by the MySQL server's logging system. However, as we have previously mentioned, MySQL logs cannot be leveraged to compute contextual events. Thus, they are not easily leveraged to start an investigation in the context of the current bottom-up implementation of our approach.

As users perform SQL queries against the web application, SQL queries are embedded in HTTP requests that can be logged by the Apache server's logging system. The Apache server then forwards, over the network, the SQL query to the MySQL server. Of course, these message exchanges (HTTP requests from the internet, and SQL queries between the Apache server and the MySQL server) are sent over the network, which is monitored by the Zeek NIDS. Information flow-related system calls are invoked by the Apache and MySQL applications to handle these messages and reply to the corresponding requests.

**On IoCs Identification.** As we can see in the architecture of our test environment (described in Figure 6.1), the Apache server and the MySQL server corresponds to two separate machines. The Apache server has to issue SQL

requests through the network to reach the MySQL server. These SQL requests can be observed and analyzed by the Zeek NIDS that monitors all the network traffic of the DMZ. Thus, in this context, Zeek NIDS allows the detection of SQL injections.

### ShellShock and RAT Attack Scenario

**Attack Scenario Description.** The second attack scenario consists in several steps. The initial compromise step consists in the exploitation of the ShellShock bash vulnerability (also referenced as CVE-2014-6271). The ShellShock vulnerability consists of the ability to execute arbitrary shell commands by embedding them in bash environment variables. Arbitrary shell commands are then executed when bash environment variables are loaded. As the Apache server relies on bash scripts for CGI, the Apache server allows a remote attacker to execute arbitrary shell commands through the injection of specially crafted bash environment variables embedded in HTTP requests.

Once the attacker has discovered the vulnerability, the second step of the attack scenario consists in making the Apache server download and execute a remote access tool (RAT), written in perl, based on the IRC protocol. Then, this remote access tool connects to its command and control server and starts a reconnaissance step by gathering some information about the web server and its surrounding environment.

**Projection on Logs.** Similarly to the first attack scenario, the actions of the ShellShock and RAT attack scenario can be observed at several abstraction layers. As we have previously mentioned, the ShellShock attack is embedded in an HTTP request. Thus, this request may be observed and recorded at the network, OS, and application layers. More specifically, the connection that contains the HTTP payload can be observed by the Zeek NIDS; the request is then received by the listening network interface, which records seen connections via the netfilter monitoring system; the web server application retrieves the request by invoking system calls, which are recorded by auditd; and, finally, the application logging system records the fact it has handled the request.

The second steps consists in the execution of the bash commands that are embedded in the HTTP request. These bash commands are executed through the CGI module of Apache. In order to execute them, bash has to invoke system calls. These system calls can be captured by auditd.

The RAT has been written to be stealthy. It tries to hide by masquerading its name with a fake one such as `rsyslog`. Similarly to bash, it has to invoke system calls to execute the binaries that allow it to perform reconnaissance actions, e.g., the execution of other binaries such as `/sbin/ip`, which allows a system administrator to list the current configuration of the network interfaces of the machine.

**On IoCs Identification.** Several indicators of compromise can be identified for this attack scenario. The current monitoring systems we have deployed allow detecting different types of actions of the attack scenario.

IoCs related to network traces can be observed by the Zeek NIDS. As the Apache web server is not configured to encrypt its communications, the Zeek NIDS can observe, dissect, and analyze the payload of the HTTP requests. It has thereby the capability to detect patterns corresponding to attempts of ShellShock exploitation.

IoCs related to application traces can be observed in the Apache logs. More specifically, the Apache logging system is configured to record HTTP headers, such as the referrer and the user-agent. ShellShock attacks embedded in HTTP requests often make use of the user-agent header to convey the payload.

IoCs related to the execution of binaries can be observed by auditd. More specifically, auditd is configured to tag specific system call invocation as suspicious, e.g., the execution of the binary `/bin/uname`.

Additionally, the execution of the IRC-based RAT consumes a lot of CPU resources. This abnormal consumption can be noticed by a system administrator that checks the health status of the web server.

### *On Noise*

Unfortunately, we do not satisfy all the requirements needed to build an ideal dataset, as described in Section 6.1.1. More specifically, we currently do not have a means to generate noise activity, such as user interactions, in the system. Such noise activity could be generated by tools such as Locust[12], which simulates users that interact with a targeted web server. Other types of approaches propose to simulate network traffic.

———

This section presented the attack assessment context of the implementation of our model. It described our vision of the ideal dataset, our test environment, our monitoring strategy with the deployed monitoring systems, and the attack scenarios we have played. The following section discusses the results of the assessment of our methodology.

## 6.2 COTS–BASED BOTTOM–UP APPROACH ASSESSMENT

This section presents our assessment of the lightweight approach implementation. More specifically, Section 6.2.1 discusses the relevance of the computed cause and dependence graphs for the two attack scenarios we have presented in the previous section. Section 6.2.4 presents the performance of the current implementation of the ETL pipelines for the lightweight approach.

### 6.2.1 Contextual Event Graphs Relevance

This section presents a qualitative and quantitative assessment, as well as the notions of false positives and false negatives in the context of our

---

approach. As our current visualization tool (presented in Section 5.3.2) does not implement filters, the figures illustrated in this section only show the relevant events from the resulting cause and dependence graphs. Events are described in a JSON format, and some of their fields have been hidden to simplify readability.

### Building Cause and Dependence Graphs

It may be recalled that if a contextual event that is considered as an IoC is given, then the purpose of our approach is to generate the causal dependency graph of events that can be considered as the causes or the consequences of this abnormal event. To achieve this goal, our implementation of the lightweight approach, presented in Section 5.3, computes the best approximation of the contextual event causal dependency model we can deduce from the raw events produced by the deployed monitoring systems. It then stores the contextual events in the ArangoDB graph database.

The cause and dependence graphs are then computed by performing a breadth-first traversal of the contextual event causal dependency graph, starting from an identified suspicious raw event or contextual event. The depth of the breadth-first traversal can be chosen according to the investigation needs. Naturally, the greater the chosen depth is, the bigger cause and dependence graphs are. Their size will be discussed in the different paragraphs of this chapter.

The following paragraphs propose an analysis of the cause and dependence graphs computed for each one of the attack scenarios presented in the previous section, namely, the *SQL Injection against the Web Application* and the *ShellShock and RAT Attack Scenario.*

### Analyzing the SQL Injection against the Web Application Scenario

For the first attack scenario, the cause and dependence graphs are computed from the Zeek NIDS alert triggered by the detection of the SQL injection while dissecting the SQL request from the web server to the database server.

The following table (Table 1) gives the key figures of the second attack scenario. The ratio columns correspond to the comparison between the cause, or dependence, graphs, and the contextual event causal dependency (CECD) graph. We can see that the resulting cause and dependence graphs are small compared to the size of the overall contextual event causal dependency graph. The cause and dependence graphs have been computed with a chosen traversal depth of 50. This depth has been chosen arbitrarily. Choosing a bigger depth would increase the size of the cause graph. However, the dependence graph already contains all the possible contextual events and causal dependencies.

This first attack scenario permits to illustrate the construction of causal dependencies among heterogeneous events produced by different machines, i.e., the web and the mysql servers. Figure 6.2 illustrates the most interesting events of the cause graph of the Zeek NIDS alert. We can notice that the resulting graph contains netfilter and audit events from the two machines, as well as Zeek and apache events from the web server. The Zeek NIDS

Table 1: Key figures of the SQL injection attack scenario

| | CECD Graph | Cause Graph | | Dep Graph | |
|---|---|---|---|---|---|
| | Count | Count | Ratio | Count | Ratio |
| Raw Events | 75939 | 433 | 0.6 % | 145 | 0.2 % |
| Contextual Events | 109178 | 771 | 0.7 % | 265 | 0.2 % |
| Causal Dependencies | 209741 | 1767 | 0.8 % | 568 | 0.3 % |
| Objects | 27387 | 55 | 0.2 % | 31 | 0.1 % |
| – ConnectionObject | 914 | 12 | 1.3 % | 4 | 0.4 % |
| – DirectoryObject | 195 | 0 | 0 % | 0 | 0 % |
| – FileObject | 1283 | 17 | 1.3 % | 0 | 0 % |
| – MemoryObject | 6433 | 0 | 0 % | 15 | 0.2 % |
| – ProcessObject | 382 | 6 | 1.6 % | 4 | 1 % |
| – SysSocketObject | 2002 | 20 | 1 % | 6 | 0.3 % |
| – UnixSocketObject | 2 | 0 | 0 % | 0 | 0 % |
| – UnixSocketPairObject | 9 | 0 | 0 % | 0 | 0 % |
| – UnknownObject | 16987 | 0 | 0 % | 2 | 0.01 % |
| – UnnamedPipeObject | 92 | 0 | 0 % | 0 | 0 % |

alert is represented in red in this figure. In Figure 6.2, we can see that the Zeek event links events from the Apache web server (all the events represented below the Zeek event) and from the MySQL server (event below the Zeek event). In the illustrated cause graph, the events above the Zeek NIDS alert correspond to the execution of the HTTP request, from its reception (`accept()` and `read()` system call events), to the connection (`connect()` and `send()` system call events) to the MySQL server to retrieve information from the database. The event below the Zeek NIDS alert corresponds to a `mysql` process sending the response of the SQL request back to the Apache web server. The cause graph illustrates the connection between the MySQL request, issued from the Apache web server, and the original HTTP request it is coming from, i.e., the Zeek event at the top of Figure 6.2. Information on the HTTP request can also be found in the Apache access log event (the $5^{th}$ event starting from the top).

Similarly to the cause graph, the dependence graph contains events corresponding to the handling of the SQL request by `mysql` processes, from its reception with an `accept()` system call event to the `write()` system call event that corresponds to the sending of the request's response, as illustrated by the bottom event of Figure 6.2.

For the first attack scenario, the computed cause and dependence graphs contain causal dependencies and contextual events that should not be part of the causes or consequences of the Zeek NIDS alert. These causal dependencies correspond to false positives (this notion will be further explained and discussed in Section 6.2.2). These false dependencies have not been illustrated. We will discuss them in the following section.

To the best of our knowledge, the cause and dependence graphs do not have any missing causal dependencies, i.e., false negatives (this notion will be further explained and discussed in Section 6.2.3).

{"__source":"Net_Zeek.log","__type":"NET", "uid":"Cd7ZWP1dJfzbrtFZ88","
    ts":"2020-04-05T16:29:30.609162Z","id.orig_h":"172.16.0.2","id.orig_
    p":57060,"id.resp_h":"192.168.51.100","id.resp_p":80,"method":"GET
    ","host":"172.16.0.1","uri":"/cases/productsCategory.php?category=1
    OR 1=1","user_agent":"curl/7.59.0","status_code":200}

▽

{"__source":"APA_netfilter.log","__type":"NET",
"host_id":"apache","in":"eth1","len":"60","src_ip":"172.16.0.2","dst_ip
    ":"192.168.51.100","protocol":"TCP","src_p":"57060","dst_p":"80","
    dest_mac":"08:00:27:6f:91:e5","source_mac":"08:00:27:0f:fb:67"}

▽

{"__source":"APA_auditd.log","__type":"SYS","exe":"/usr/sbin/apache2","
    pid":"22947","ppid":"22891","syscall":"288","operation":"accept","
    saddr":"0200DEE4AC1000020000000000000000","success":"yes","time
    ":"1586103781.741","egid":"33","euid":"33","sgid":"33","event id
    ":"66694","exit":"11"}

▽

{"__source":"APA_auditd.log","__type":"SYS","exe":"/usr/sbin/apache2","
    pid":"22947","ppid":"22891","syscall":"0","operation":"read","a
    0":"11","size":"122","success":"yes","time":"1586104170.607","egid
    ":"33","euid":"33","sgid":"33","event id":"66699","exit":"122"}

▽

{"__source":"APA_access.log","__type":"APP",
"time_received":"[05/Apr/2020:16:29:30 +0000]","pid":"22947","local_ip
    ":"192.168.51.100","server_port":"80","remote_host":"172.16.0.2","
    server_port_remote":"57060 ","request_method":"GET","request_url":"/
    cases/productsCategory.php?category=1 OR 1=1","request_http_ver
    ":"1.1","status":"200","bytes_tx":"9962","request_header_referer
    ":"-","request_header_user_agent":"curl/7.59.0"}

▽

{"__source":"APA_auditd.log","__type":"SYS","exe":"/usr/sbin/apache2","
    pid":"22947","ppid":"22891","syscall":"42","operation":"connect","
    saddr":"02000CEAC0A8336500000000000000000","a0":"12","success":"no","
    time":"1586104170.611","egid":"33","euid":"33","sgid":"33","event id
    ":"66707","exit":"-115"}

▽

{"__source":"APA_auditd.log","__type":"SYS","exe":"/usr/sbin/apache2","
    pid":"22947","ppid":"22891",syscall":"44","operation":"send","a
    0":"12","size":"65","success":"yes","time":"1586104170.615","egid
    ":"33","euid":"33","sgid":"33","event id":"66711","exit":"65"}

▽

{"__source":"APA_netfilter.log","__type":"NET", "host_id":"apache","out
    ":"eth1","len":"60","src_ip":"192.168.51.100","dst_ip
    ":"192.168.51.101","protocol":"TCP","src_p":"41238","dst_p":"3306"}

▽

{"__source":"Net_Zeek.log","__type":"NET","uid":"CGUJkC1cvTct10IiR4","ts
    ":"2020-04-05T16:29:30.613652Z","id.orig_h":"192.168.51.100","id.
    orig_p":41238,"id.resp_h":"192.168.51.101","id.resp_p":3306,"proto
    ":"tcp","duration":0.0133020878,"orig_bytes":172,"resp_bytes":2437,"
    conn_state":"SF"}

△

{"__source":"MYSQL_auditd.log","__type":"SYS","exe":"/usr/sbin/mysqld",
"pid":"21915","ppid":"1","syscall":"1","operation":"write","a0":"31","
    size":"48","success":"yes","time":"1586104170.613","egid":"112","
    euid":"106","fsgid":"112","fsuid":"106","gid":"112","sgid":"112","
    suid":"106","uid":"106","event id":"35995","exit":"48"}

**Figure 6.2:** Relevant parts of the event cause graph of the SQLi attack scenario.

*Analyzing the ShellShock and RAT Attack Scenario*

For the second attack scenario, the cause and dependence graphs are computed from the Zeek NIDS alert triggered by the detection of the ShellShock attack attempt through an HTTP request coming from the outside of the network. In the following analysis, only the dependence graph is illustrated as the cause graph only contains 9 contextual events. Additionally, in this specific attack scenario, the Zeek NIDS alert actually corresponds to the first event of the scenario. In fact, the related HTTP request is the cause of the attack scenario. Thus, the computation of the cause graph of the alert is less relevant. The following table (Table 2) gives the key figures of the second attack scenario. These numbers will be discussed in the following paragraphs. As this attack scenario only involves the web server, figures of Table 2 only refer to the events of the web server.

**Table 2:** Key figures of the ShellShock and RAT attack scenario

| | CECD Graph | Dep Graph | |
| --- | --- | --- | --- |
| | Count | Count | Ratio |
| Raw Events | 34117 | 1992 | 5.8 % |
| Contextual Events | 68107 | 3610 | 5.3 % |
| Causal Dependencies | 79460 | 4583 | 5.8 % |
| Objects | 23583 | 620 | 2.6 % |
| – ConnectionObject | 109 | 4 | 3.7 % |
| – DirectoryObject | 36 | 0 | 0 % |
| – FileObject | 587 | 1 | 0.2 % |
| – MemoryObject | 4232 | 546 | 12.9 % |
| – ProcessObject | 259 | 51 | 19.7 % |
| – SysSocketObject | 1470 | 5 | 0.3 % |
| – UnixSocketObject | 2 | 0 | 0 % |
| – UnixSocketPairObject | 7 | 0 | 0 % |
| – UnknownObject | 16929 | 2 | 0.01 % |
| – UnnamedPipeObject | 73 | 11 | 15.1 % |

An analysis of the dependence graph reveals that, to the best of our knowledge, all the contextual events, and thus all the 1992 events it contains correspond to traces of the attack scenario. Thus, the resulting dependence graph does not contain any false causal dependencies, i.e., false positives (this notion will be further explained and discussed in Section 6.2.2). Additionally, the computed dependence graph contains all the events that help understand the overall attack scenario.

Looking at these figures, the size of the computed dependence graph seems big compared to the overall contextual event causal dependency graph. This can be explained by the fact that the testing environment does not have a lot of noise activity. The number of retrieved events in the dependence would not increase with additional noise activity in the test environment. The following paragraphs explain the conclusions that can be drawn from the analysis of the dependence graph.

```
{"__source":"Net_Zeek.log","__type":"NET", "uid":"C7SKfD3hbTetA4cQb2",
"ts":"2020-04-03T15:28:45.389087Z", "id.orig_h":"172.16.0.2",
"id.orig_p":33032, "id.resp_h":"192.168.51.100", "id.resp_p":80,
"method":"GET","host":"172.16.0.1","uri":"/cgi-bin/shell.sh",
"user_agent":"() { :;}; /usr/bin/perl -e 'print \"Content-Type: text/
    plain\\r\\n\\r\\nATTACK SUCCESS\\n\"; system(\"/usr/bin/curl http
    ://172.16.0.2/attacker1-irc-bot.pl -o /tmp/core; perl /tmp/core; rm
    /tmp/core;\");'"}
```

▽

```
{"__source":"Net_netfilter.log","__type":"NET", "host_id":"apache","in
    ":"eth1","len":"60","src_ip":"172.16.0.2","dst_ip
    ":"192.168.51.100","protocol":"TCP","src_p":"33032","dst_p":"80","
    dest_mac":"08:00:27:05:d9:f1","source_mac":"08:00:27:b4:c7:20", "raw
    ":"Apr 3 15:28:45 apache kernel: [ 565.137077] [nfconnections] IN=
    eth1 OUT= MAC=08:00:27:05:d9:f1:08:00:27:b4:c7:20:08:00 SRC
    =172.16.0.2 DST=192.168.51.100 LEN=60 TOS=0x00 PREC=0x00 TTL=63 ID
    =44609 DF PROTO=TCP SPT=33032 DPT=80 WINDOW=64240 RES=0x00 SYN URGP
    =0 \n"}
```

▽

```
{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/sbin/apache2","
    pid":"22898","ppid":"22891","syscall":"288","operation":"accept","
    saddr":"02008108AC10000200000000000000000","success":"yes","time
    ":"1585927592.670","type":"SOCKADDR","egid":"33","euid":"33","sgid
    ":"33","event id":"62327","exit":"11"}
```

▽

```
{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/sbin/apache2","
    pid":"22898","ppid":"22891","syscall":"56","operation":"fork","flags
    ":"CLONE_CHILD_CLEARTID|SIGCHLD|CLONE_CHILD_SETTID","success":"yes
    ","time":"1585927725.389","egid":"33","euid":"33","event id
    ":"62346","exit":"22929"}
```

▽

```
{"__source":"apache_access.log","__type":"APP","time_received":"[03/Apr
    /2020:15:28:45 +0000]","pid":"22898","local_ip":"192.168.51.100","
    server_port":"80","remote_host":"172.16.0.2","server_port_remote
    ":"33032 ","request_method":"GET","request_url":"/cgi-bin/shell.sh
    ","request_http_ver":"1.1","status":"200","request_header_referer
    ":"-","request_header_user_agent":"() { :;}; /usr/bin/perl -e 'print
    \\"}
```

**Figure 6.3:** Event dependence graph of the beginning of the ShellShock and RAT attack scenario.

The beginning of the event dependence graph is illustrated in Figure 6.3. The connection corresponding to the Zeek NIDS alert is handled by an apache process, with the PID 22898, with an accept() system call. We can see the causal dependencies between the Zeek NIDS alerts, the netfilter event, and the system call event. The analysis of the Zeek event shows that the ShellShock payload has been embedded in the *user-agent* HTTP header. The event dependence graph also shows that the clone() system call event (i.e. the next to last event, starting from the bottom of the figure) and the apache application event, at the bottom of the figure, are causally dependent, in the sense of "▷,"on the accept() system call event. The clone() system call event actually corresponds to the execution of the CGI module.

{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/sbin/apache2","
    pid":"22898","ppid":"22891","syscall":"56","operation":"fork","flags
    ":"CLONE_CHILD_CLEARTID|SIGCHLD|CLONE_CHILD_SETTID","success":"yes
    ","time":"1585927725.389","egid":"33","euid":"33","event id
    ":"62346","exit":"22929"}

▽

{"__source":"Sys_auditd.log","__type":"SYS","exe":"/bin/bash","pid
    ":"22929","ppid":"22898","syscall":"59","operation":"execve","execve
    _a0":"/bin/bash","execve_a1":"/usr/lib/cgi-bin/shell.sh","execve_
    argc":"2","mode0":"0100755","mode1":"0100755","mode2":"0100755","
    path0":"/usr/lib/cgi-bin/shell.sh","path1":"","path2":"","egid
    ":"33","euid":"33","success":"yes","time":"1585927725.389","event id
    ":"62367","exit":"0"}

▽

{"__source":"Sys_auditd.log","__type":"SYS","exe":"/bin/bash","pid
    ":"22929","ppid":"22898","syscall":"56","operation":"fork","flags":"
    CLONE_CHILD_CLEARTID|SIGCHLD|CLONE_CHILD_SETTID","success":"yes","
    time":"1585927725.397","egid":"33","euid":"33","event id":"62402","
    exit":"22930"}

▽

{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/bin/perl","pid
    ":"22930","ppid":"22929","syscall":"59","operation":"execve","execve
    _a0":"/usr/bin/perl","execve_a1":"-e","execve_a2":"print \"Content-
    Type: text/plain\r\n\r\nATTACK SUCCESS\n\"; system(\"/usr/bin/curl
    http://172.16.0.2/attacker1-irc-bot.pl -o /tmp/core; perl /tmp/core;
     rm /tmp/core;\");","execve_argc":"3","path0":"/usr/bin/perl","path
    1":"","success":"yes","time":"1585927725.397","egid":"33","euid
    ":"33","event id":"62403","exit":"0"}

**Figure 6.4:** Event dependence graph of the execution of the ShellShock payload.

Figure 6.4 illustrates the events that are causally dependent on the `clone()` system call event of Figure 6.3. The child `apache` process starts a shell that in turn clone itself. The child shell process invokes `perl`, which executes the `perl` payload embedded in the attack. We can see that the payload contains commands to download a file from a remote server, save it as `/tmp/core`, execute it with `perl`, and remove it. The events that causally depend on the bottom event of Figure 6.4 show the execution of this payload. The `perl` process clones itself, becomes a shell with an `execve()` system call that, in turn, clone itself three times, i.e., once for each invocation of the `curl`, `perl`, and `rm` binaries. It is worth mentioning that even if the `/tmp/core` file is removed, all the actions related to it are captured by the deployed monitoring systems.

{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/bin/perl","comm
    ":"/sbin/klogd -c ","pid":"22933","ppid":"22931","syscall":"56","
    operation":"fork","flags":"CLONE_CHILD_CLEARTID|SIGCHLD|CLONE_CHILD_
    SETTID","success":"yes","time":"1585927725.497","egid":"33","euid
    ":"33","event id":"63719","exit":"22935"}

▽

{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/bin/perl","comm
    ":"/sbin/klogd -c ","pid":"22935","ppid":"1","syscall":"42","
    operation":"connect","saddr":"020000350A000203000000000000000000","
    success":"yes","time":"1585927725.505","type":"SOCKADDR","egid
    ":"33","euid":"33","event id":"63809","exit":"0"}

▽

{"__source":"Net_netfilter.log","__type":"NET", "host_id":"apache","out
    ":"eth0","len":"36","src_ip":"10.0.2.15","dst_ip":"10.0.2.3","
    protocol":"UDP","src_p":"43940","dst_p":"53","raw":"Apr 3 15:28:45
    apache kernel: [ 565.258817] [nfconnections] IN= OUT=eth0 SRC
    =10.0.2.15 DST=10.0.2.3 LEN=56 TOS=0x00 PREC=0x00 TTL=64 ID=53539 DF
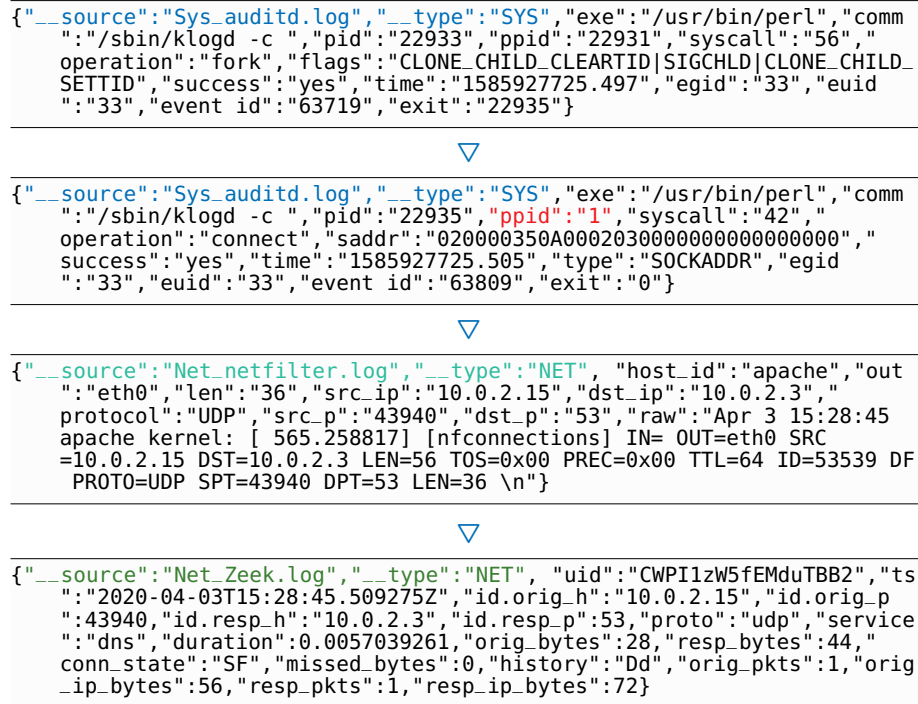     PROTO=UDP SPT=43940 DPT=53 LEN=36 \n"}

▽

{"__source":"Net_Zeek.log","__type":"NET", "uid":"CWPI1zW5fEMduTBB2","ts
    ":"2020-04-03T15:28:45.509275Z","id.orig_h":"10.0.2.15","id.orig_p
    ":43940,"id.resp_h":"10.0.2.3","id.resp_p":53,"proto":"udp","service
    ":"dns","duration":0.0057039261,"orig_bytes":28,"resp_bytes":44,"
    conn_state":"SF","missed_bytes":0,"history":"Dd","orig_pkts":1,"orig
    _ip_bytes":56,"resp_pkts":1,"resp_ip_bytes":72}

**Figure 6.5**: Event dependence graph of the domain name resolution of the RAT.

Figure 6.5 illustrates the events corresponding to the beginning of the execution of the RAT. Following the execution of the /tmp/core file with perl, we can see the start of a new process called /sbin/klogd -c. In fact, this process corresponds to the execution of the /tmp/core file. Additionally, we can notice that the PPID (i.e., the PID of its parent process) has been changed to 1. At this point, we can suspect that the process name is fake, i.e., it attempts to be stealthy. Additionally, we can see that the process issues a DNS request to attempt to retrieve the IP address of its C&C.

Figure 6.6 illustrates the following events of the fake process. These events correspond to the beginning of the communications with the C&C server, which actually correspond to IRC communications. We can see that the process seems to behave like a bot. The overall scenario involves four network connections. So far, we have seen that the three network connections are causally dependent on the network connection corresponding to the ShellShock HTTP request. The three network connections respectively correspond to: (1) the download of a file, through a GET HTTP request, from a remote web server; (2) a DNS request to resolve a domain name. This DNS request actually corresponds to an attempt to get the IP address of the attacker's command and control (C&C) server; (3) IRC communications with the C&C server on its port 6667.

The fake /sbin/klogd -c process further performs many actions: the execution of the uname, id, ip, and "cat /etc/passwd" commands. These actions correspond to the *reconnaissance step* of the attack scenario. Continuing the analysis of the event dependence graph, we can also notice that all child processes of /sbin/klogd -c are linked to the same socket and connection, i.e., the connection to the C&C server. This connection very likely corresponds to a data exfiltration channel.
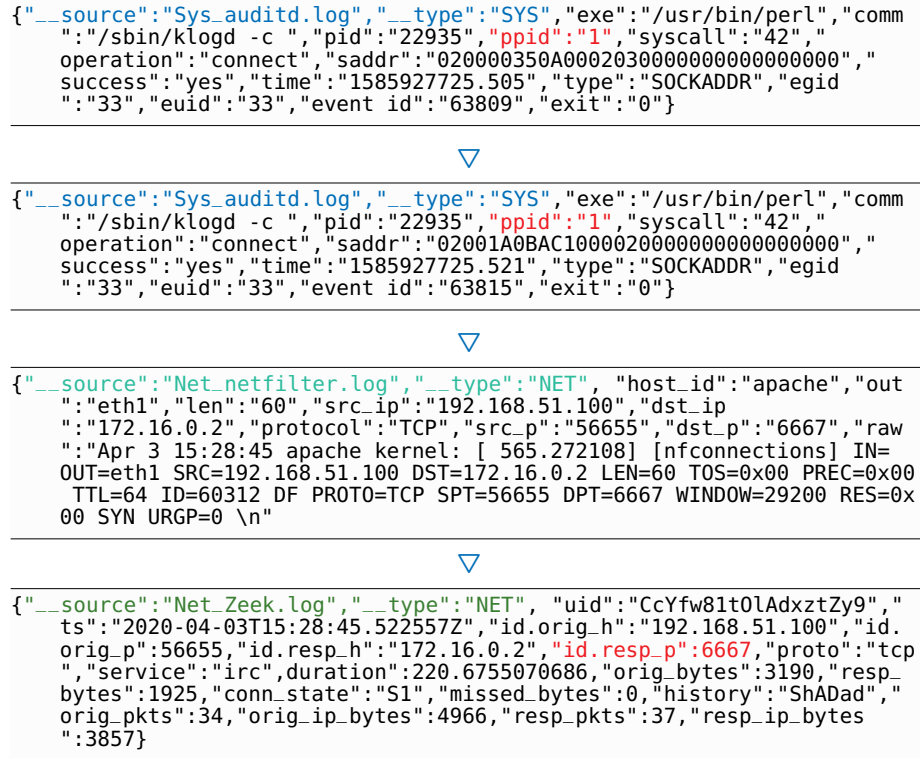
{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/bin/perl","comm":"/sbin/klogd -c ","pid":"22935","ppid":"1","syscall":"42","operation":"connect","saddr":"020000350A0002030000000000000000","success":"yes","time":"1585927725.505","type":"SOCKADDR","egid":"33","euid":"33","event id":"63809","exit":"0"}

▽

{"__source":"Sys_auditd.log","__type":"SYS","exe":"/usr/bin/perl","comm":"/sbin/klogd -c ","pid":"22935","ppid":"1","syscall":"42","operation":"connect","saddr":"02001A0BAC10000200000000000000000","success":"yes","time":"1585927725.521","type":"SOCKADDR","egid":"33","euid":"33","event id":"63815","exit":"0"}

▽

{"__source":"Net_netfilter.log","__type":"NET", "host_id":"apache","out":"eth1","len":"60","src_ip":"192.168.51.100","dst_ip":"172.16.0.2","protocol":"TCP","src_p":"56655","dst_p":"6667","raw":"Apr 3 15:28:45 apache kernel: [ 565.272108] [nfconnections] IN= OUT=eth1 SRC=192.168.51.100 DST=172.16.0.2 LEN=60 TOS=0x00 PREC=0x00 TTL=64 ID=60312 DF PROTO=TCP SPT=56655 DPT=6667 WINDOW=29200 RES=0x 00 SYN URGP=0 \n"}

▽

{"__source":"Net_Zeek.log","__type":"NET", "uid":"CcYfw81tOlAdxztZy9","ts":"2020-04-03T15:28:45.522557Z","id.orig_h":"192.168.51.100","id.orig_p":56655,"id.resp_h":"172.16.0.2","id.resp_p":6667,"proto":"tcp","service":"irc","duration":220.6755070686,"orig_bytes":3190,"resp_bytes":1925,"conn_state":"S1","missed_bytes":0,"history":"ShADad","orig_pkts":34,"orig_ip_bytes":4966,"resp_pkts":37,"resp_ip_bytes":3857}

**Figure 6.6:** Event dependence graph of the IRC communications of the RAT.

The analysis of the dependence graph of the ShellShock and RAT attack scenario shows promising results regarding the relevance of our approach. We have shown that our approach allows retrieving the heterogeneous events corresponding to the traces of the attack scenario on the monitored system and that the different types of events contain information that complements each other.

———————

Based on the hypothesis that all the raw events produced by the deployed monitored system are correctly handled by the *transform components* we implemented (i.e., CEs & CDs, message exchange, and timelines transform components described in Section 5.3.2), generated contextual events and their related causal dependencies accurately represent the *best approximation* of the contextual event causal dependency model we can compute. The computed approximation may contain false causal dependencies, missing causal dependencies, or both, due to the limitations of the deployed monitoring systems, e.g., poor semantic quality of logged information. This corresponds to the notions of false positives and false negatives in the context of the contextual event causal dependency model. The following sections discuss these two notions.

### 6.2.2 False Positives—False Causal Dependencies

In the context of the contextual event causal dependency model we defined in Chapter 4, a *false positive* corresponds to a false causal dependency between two contextual events. Computing false causal dependencies during the *transform* step of raw events handling implies that computed cause and dependence graphs may contain contextual events that are not causally dependent on the event of interest in the sense of "$\rightarrow$," as defined in Chapter 4 (Section 4.4.2). The following paragraphs illustrate the different cases where false positives can be computed.

***On the Ability to Identify or Compute Sessions***

As we have defined in Section 4.3.1 (Definition 4.3), the execution of an object may be made up of several sessions. Thus two contextual actions pertaining to two different sessions might not be causally dependent. This remark also applies to the contextual event causal dependency and event causal dependency models. In fact, it may be recalled that sessions aim to decrease the size of computed cause and dependence graphs. They contribute to the reduction of false positives.

Currently, if we don't have a means to identify and compute sessions, then we adopt a conservative assumption regarding the computation of causal dependencies, i.e., we suppose that all the computed contextual events related to the same object are causally dependent. Naturally, some computed causal dependencies may be false. This assumption is thereby subject to false positive production.

Regarding our current implementation, we deduce sessions for the Apache and MySQL servers. As we mentioned in the last paragraph of Section 5.3.3, we consider that `accept()` system calls start new sessions for $apache$ processes.

As we have previously mentioned in Section 6.2.1, the cause and dependence graphs of the SQL injection attack scenario contain false positives. The dependence graph of the SQL injection attack scenario contains 102 contextual events and 201 causal dependencies that are not part of the execution of the SQL injection request. This corresponds to a false positive (false causal dependencies) rate of 35.4% (201/568) and a rate of irrelevant contextual events of 38.5% (102/265). Regarding the cause graph of the SQL injection attack scenario, the false positive rate is 75.9% (1347/1773), and the rate of irrelevant contextual events is 81.7% (630/771). These high numbers could be lowered by decreasing the depth of traversal. However, the number of relevant events would also decrease.

This is explained by the fact that the mysql process that handles the `accept()` system call is different from the one that actually handles the mysql request. We currently do not have any means to compute sessions for the latter process. Therefore, the conservative assumption implies that all its contextual events are in the same session. The computation of cause or dependence graphs thereby includes them. Naturally, the bigger the depth of traversal, for cause and dependence graph construction, is, the more false dependencies will be included in the traversal. This corresponds to a com-

pound effect which is called the "dependence explosion problem." This notion has been discussed in Section 3.4.4.

### On the Accuracy of the Deployed Information Flow Monitoring Systems

Naturally, the accuracy of the contextual event causal dependency model greatly relies on the monitoring systems' capabilities to observe contextual actions.

Let's consider the OS abstraction layer to illustrate the notion of granularity. The following example illustrates the limitations of observing system call invocations. Let's consider that a process reads from a file $f_1$ and, then, writes to another one called $f_2$. From the point of view of system call invocations, we cannot determine whether the process transferred information from $f_1$ to $f_2$. Thus, we conservatively consider that information has flowed between the two files. In other words, we might consider a false causal dependency, through the *transitivity* property of "$\mapsto$" between the contextual actions of $f_1$ and $f_2$. In order to overcome this issue, we would need to be able to observe the states of objects.

Additionally to the intrinsic limitations of the observing system call invocations, limitations can come from the capabilities of the deployed monitoring system, e.g., RfBlare [Georget, 2017] performs a better information flow tracking than auditd-based information flow trackers.

### Consequences

The consequences of generating false positives are multiple. First, it may lead to the dependency explosion problem, as described in Section 3.4.4, which would greatly impact cause and dependence graphs computation depending on the adopted graph traversal strategy. For instance, the performance impact on a naive breadth-first traversal strategy would be important. Computed cause and dependence graphs would also contain contextual events that do not correspond to the attack traces. Thus false positives may impact cyber defense analysts' analyses.

### 6.2.3 False Negatives—Missing Causal Dependencies

In the context of the contextual event causal dependency model, a *false negative* corresponds to the absence of an actual causal dependency between two contextual events.

To the best of our knowledge, given the events that are generated by the monitoring systems we have deployed, and the analysis of the computed cause and dependence graphs, we don't have any false negatives in our graphs, i.e., missing causal dependencies.

The following paragraphs illustrate the different cases where false negatives can happen.

### On Monitoring System Failure

Assuming that the *transform components* we implement are reliable, false negatives can be explained by the failure of a given monitoring system, e.g.,

a network sniffer that drops packets because the network traffic to analyze is too big, or a system call monitor that has not been configured or designed to handle bursts of system call invocations.

### On ETL Pipelines Reliability

Assuming that deployed monitoring systems are reliable, i.e., any monitored contextual actions are correctly recorded into events, faults might come from the ETL pipelines, either from the extract, transform, or load components. In other words, raw events, or computed contextual events and their causal dependencies, may be accidentally lost in the ETL pipelines. This may happen with a poorly designed system for ETL pipelines.

### On the Lack of Observation Capability

False negatives can also be explained by a monitoring system's inability to observe some contextual actions. For example, auditd might not be configured to capture all information flow-related system calls.

Another observation capability issue is the approximation of timestamps, as discussed in Section 4.4.1. For example, auditd only records the exit of system call invocations. Thus, we cannot know when they started and can only consider them as instantaneous. In practice, we have to rely on timestamps to order contextual events and events locally. However, their corresponding contextual actions might have a different order. This behavior might introduce false positives, as some contextual events might be incorrectly *included* in contextual event causal dependency graph traversals, and false negatives, as some contextual events might be incorrectly *excluded* in contextual event causal dependency graph traversals.

### Attacker Behaviors

Of course, false negatives may be caused by attackers' actions. Such actions could correspond to monitoring systems evasion, or specific actions that target the computation of our model, i.e., the computation of contextual events and their causal dependencies. For example, attackers may leverage the timestamp approximation issue to attack the computation of our model, or even delete log entries.

### Consequences

False negatives greatly impact the overall methodology we propose, i.e., missing causal dependencies hinders the capability to perform graph traversal and compute cause and dependence graphs. All the work previously presented in this manuscript have this weakness. To the best of our knowledge, no approach has been proposed to compensate it in the literature.

---

The previous sections proposed a qualitative and quantitative assessment of the implementation of our methodology. The following section discusses ETL pipelines' performance.

### 6.2.4 ETL Pipelines Performance—Event Handling Rate

This section presents an assessment of the performance of our current ETL pipeline implementation, i.e., the rate of event handling, or in other words, the time needed to compute the contextual event causal dependency model. All the tests are done on a single machine where all the components of the ETL pipelines, i.e., *transform components*, Kafka, and the ArangoDB graph database, run in parallel. Our current implementation of the *transform components*, are written in Python. The testing machine is made up of 16Gb of RAM and an Intel i7-7600U CPU.

As we have mentioned in the "ETL Pipelines" section (Section 5.3.2), *extract components* are currently not implemented in the current version of our implementation. Log files are manually collected and read by *transform components*.

**Transform Component Performance.** As we have previously presented in Section 5.3.2, the architecture of the ETL pipelines is distributed. Each *transform component*'s performance is independent of the other ones' performances. However, the performance of the overall ETL pipelines is limited to the component with the poorest performance.

Table 3: Average event handling rate in seconds of transform components

| Transform Component | Event Handling Rate (No of Events per Seconds) | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| | Mean | Min | Q1 | Median | Q3 | Max |
| Auditd | 2377.82 | 2191.86 | 2357.33 | 2399.07 | 2414.72 | 2441.67 |
| Netfilter | 5960.83 | 3474.55 | 5748.77 | 6141.9 | 6302.01 | 6517.95 |
| Apache | 2591.01 | 2337.31 | 2573.1 | 2597.18 | 2616.72 | 2716.52 |
| Zeek | 17947.03 | 11228.87 | 17034.19 | 18369.6 | 18887.18 | 19388.58 |

| Transform Component | CE Handling Rate (No of CEs per Seconds) | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| | Mean | Min | Q1 | Median | Q3 | Max |
| Auditd Netfilter Match | 2337.17 | 2155.71 | 2312.29 | 2357.58 | 2373.93 | 2401.62 |
| Time Ordering | 2281.18 | 2107.39 | 2257.75 | 2301.41 | 2317.82 | 2344.2 |
| Timelines | 910.81 | 835.42 | 899.35 | 922.37 | 927.93 | 944.24 |
| Message Exchange | 12006.9 | 7652.81 | 11451.24 | 12168.73 | 12616.36 | 13159.26 |
| Overall Pipeline | 663.23 | 608.17 | 653.68 | 671.96 | 675.06 | 686.16 |

| Load Component | Import Rate (No of Documents per Seconds) | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| | Mean | Min | Q1 | Median | Q3 | Max |
| Arango Import | 8222.04 | 8149.22 | 8199.83 | 8215.95 | 8235.74 | 8321.81 |

Table 3 shows the performance figures of our transform components, i.e., the number of events, or contextual events, handled per second. The overall ETL pipelines have been run a hundred times using the ShellShock and RAT attack scenario dataset. The table presents the mean, the minimal, first quartile, median, third quartile, and maximum handling rates for each transform component.

**Database Loading Performance.** Table 3 also shows the performance of the *load* part of our ETL pipelines. The *load component* adopts a batch strategy to index the contextual events and causal dependencies computed by the *transform components* into the ArangoDB graph database.

As we can see, the major bottleneck of our current implementation is the *timelines transform component*. We discuss ways to improve the performance of the ETL pipelines in Section 6.3.2.

---

Section 6.2 presented a qualitative and quantitative assessment of our current implementation of the bottom-up strategy. More specifically, we have discussed the relevance of the contextual event causal dependency graph we compute and have proposed an assessment of the performance of the ETL pipelines.

## 6.3 DISCUSSIONS

Section 6.3 further discusses the current implementation of our model. More specifically, we will discuss the size of the computed contextual event causal dependency model and the weaknesses of our implementation.

### 6.3.1 Graph Size

#### *Analysis of Contextual Events and Causal Dependencies Computation*

Currently, contextual events and causal dependencies are stored as documents, i.e., in the JSON format, inside the ArangoDB graph database. More specifically, one document corresponds to one contextual event or to one causal dependency. As a reminder, in the context of the lightweight implementation we presented in Section 5.3, we potentially transform a raw event into several contextual events and causal dependencies. It may be recalled that our current architecture temporally orders all the events emanating from each host. This property allows us to build timelines easily, i.e., a computed contextual event simply needs to be appended to the timeline of its corresponding object. Several cases are possible:

1. A raw event is transformed into a unique contextual event. Then, in the context of the monitored object, we have two possibilities:
   a) If the contextual event is part of the current session of the object, then a causal dependency is created between this contextual event and its previous one, according to the object's timeline;
   b) If the contextual event pertains to a brand new session of the object, then any causal dependency is created;
2. A raw event is transformed into several contextual events and causal dependencies between them. This is, for example, the case for information flow-related system call events where a first causal dependency is created between the two created contextual events. Then, similarly to the first case with a) and b), we examine whether each contextual event pertains to a running object session, or to a new one.

Following the previous remarks, we can conclude that the number of stored documents is at least twice the number of analyzed events.

*On Attacks' Time Length*

So far, we did not address the issues related to the time length of multi-step attacks. In fact, multi-step attacks can be scattered over time for a period of several days, weeks, or even months. Such attack scenarios are also called Advanced Persistent Threat [Hutchins et al., 2011].

Regarding our approach and methodology, several cases are possible for this type of attack scenarios.

**Time between Consecutive Contextual Actions.** As multi-step attacks can be scattered over time, corresponding contextual actions can also be scattered over time. An interesting property of the contextual action causal dependency relationship we defined is that, given two *consecutive* causally dependent contextual actions $(a_1, (o, t_1))$ and $(a_2, (o, t_2))$ performed by the same object, $(a_2, (o, t_2))$ may be performed long after $(a_1, (o, t_1))$. Thus, even if an object does not have activity for a long time, its next contextual action will be added to its timeline and, if they pertain to the same session, a causal dependency will be created between this contextual action and the previous one. Therefore, time between consecutive contextual actions does not matter.

**A Visualization Problem.** Supposing that we have a good approximation of the contextual event causal dependency model, the cause and dependence graphs would contain all the contextual events that correspond to traces of the attack. The cause and dependence graphs might be too big to be analyzed by cyber defense analysts. In fact, we argue that this issue corresponds to a visualization problem. It may actually be counter-productive to display cause and dependence graphs entirely. However, they ideally contain all the information items available for attack investigation. Thus, we could imagine a solution that would enable the computation of different kinds of perspective, such as only displaying causally dependent network-related events, or causally dependent application-related events.

*On Data Reduction and Storage*

The issues related to the storage of contextual events and their causal dependency are orthogonal to the work presented in this manuscript. They represent a full research subfield and we did not address these issues in our work.

Such studies correspond to the capability to handle a high throughput of events into the database. Depending on the architecture of the extract, transform and load pipelines, proposed solutions greatly differ. For instance, CamQuery [Pasquier et al., 2018], which focuses on the provenance collection in the Linux Kernel, merges the capture and storage layers. On the opposite, our approach separates the collection, transform, and load components. Additionally, it centralizes all computed contextual events and their causal dependencies into a single database (i.e., the ArangoDB graph database).

In order to address the problem of storage, various approaches have been proposed regarding system call-based causal dependency approaches:

GARBAGE COLLECTION – In [Lee et al., 2013b], Lee et al. propose to leverage garbage collection methodologies to discard events that are not considered to be useful for forensics.

EVENT AGGREGATION STRATEGIES – In [Xu et al., 2016], Xu et al. propose strategies to aggregate events by leveraging graph analysis: (1) they aggregate events that have identical contributions to the causal dependency analysis; (2) they identify subgraphs that can be shrunk without too much impact of the causal dependency analysis. Hossain et al. further develop these event aggregation strategies in [Hossain et al., 2018].

GENERATING LESS EVENTS – In [Ma et al., 2016], Ma et al. change their recording strategy by only logging `write()` system calls without losing information for causal dependency analysis.

These approaches reduce the quantity of system call events by fusing them or discarding them. This might be a problem in the context of heterogeneous events as some events might contain valuable information, with higher semantics, for attack investigation.

---

### 6.3.2 Limitations of our Implementation

This section discusses the limitations of our current implementation and its assessment. It goes through the limitations of our test environment, the deployed monitoring systems, as well as our implementation of the ETL pipelines.

#### Test Environment's Limitations

**The Lack of Representative Activity.** As we have previously mentioned, the test environment we have built does not contain activity from real users. Its representativeness of the reality can easily be questioned, and it does not allow us to generate an ideal dataset. However, it may be recalled that the primary goal of our implementation is to illustrate that the implementation of the bottom-up strategy is possible. We argue that the dataset we produce is sufficient to satisfy the prerequisites to attain this goal.

**On Virtualization.** Additionally, the fact that our test environment is fully virtualized can also be criticized as it can be seen as not representative of a real enterprise network. We argue that virtualization is not an issue as it tends to be more and more adopted in enterprise networks with the usage of cloud provider services.

#### Deployed Monitoring Systems' Limitations

It may be recalled that our capability to adopt the bottom-up strategy and compute the contextual event causal dependency model relies on the adopted monitoring strategy. This section discusses the limitations caused by the adopted monitoring strategy.

**On Attack Types Coverage.** Naturally, the capability to detect specific types of attacks depends on the deployed monitoring systems. For instance, our current monitoring strategy does not have a means to detect rootkits. Therefore, some steps of attack scenarios might be undetected. However, other actions related to the rootkits can be captured with our monitoring strategy, e.g., their installations, or their network and system call-related activities. Additionally, as we have previously explained, a multi-step attack involves several steps that are likely to leave traces in the different abstraction layers of the monitored system. Thus, we argue that our implementation of the bottom-up strategy is still relevant as it can help cyber defense analysts to retrieve and investigate some steps of the overall attack scenario.

**On Performance.** Additionally to the fact that we do not, and cannot, cover the detection of all attack types, we are also limited by the performance of the chosen deployed monitoring systems. For instance, auditd has not been designed to record all the system call related to information flows. Thus, the number of system call invocations to record might be too significant for auditd. New monitoring system designs, e.g., KCAL [Ma et al., 2018] or CamQuery [Pasquier et al., 2018], have been proposed to overcome this problem.

**On Deployed Monitoring System's Configuration Capabilities.** COTS monitoring systems might not be able to record all the information we need to compute contextual events and their causal dependencies. This is, for example, the case with the MySQL server's logging system, which does not allow the recording of the PID of the process that actually handled the SQL query. Ideally, the logging systems of application should be configurable enough to apply the bottom-up strategy and compute contextual events and causal dependencies. We argue that this constraint is not an issue as the logging systems of applications often rely on logging frameworks such as Log4J and Log4C, as we have previously mentioned in Section 1.1.4.

### ETL Pipelines' Limitations

As we have previously mentioned, our current implementations of the *transform components* are written in Python and leverage the CPython interpreter. Other languages and optimized designs could be explored to increase the rate of event analysis.

---

Following the discussions of the limitations of our current implementation, the next section presents a comparison of our approach and implementation with other approaches.

### 6.3.3 Comparison of our Implementation with other Approaches

As we have previously mentioned in Chapter 4, the new causality model we define aims to unify prior work on causal dependency. To the best of our knowledge, this new model is the first to address the formal definition of the causal dependency relationship among heterogeneous events. As we have seen in Chapter 5, all the work presented in the first part of this manuscript (Part I) enable the computation of parts of the overall causality model. Additionally, a few approaches focus on the analysis of heterogeneous events.

**Provenance Layering.** The closest work to our implementation corresponds to the approaches presented in the *Layered Provenance* paragraph of Section 3.4.4. In fact, these approaches correspond to various implementations adopting the bottom-up strategy. They propose a monitoring strategy that allows them to compute an approximation of the contextual event causal dependency model from raw events.

**Leveraging Learning Techniques to Detect Multi–Step Attack Scenarios.** HERCULE, presented in the *Detecting Multi-Step Attacks with Heterogeneous Logs Analysis* paragraph of Section 1.4.5 is also close to our work. However, contrary to our implementation of the bottom-up strategy, the authors leverage a supervised learning algorithm to compute a weighted event graph. We argue that, as any methodology that leverages a supervised learning algorithm, a good training data set is needed. Unfortunately, obtaining or building realistic training datasets containing multi-step attacks and benign activity is not an easy task to do. HERCULE's approach assessment has currently been done using a single target machine. In a more complex setting, such as a small or middle size company network, the existence of such a learning dataset is questionable. Indeed, even if they may have high-level commonalities, each multi-step attack scenario is unique as any target network or system is different. Literature has also mentioned that supervised learning *for multi-step attack scenarios* is complex to apply due to the lack of training datasets containing enough complex attack scenarios data [Ourston et al., 2003].

## 6.4  SUMMARY

Chapter 6 presents an assessment of the bottom-up strategy implementation we have made.

**The Need to Build our Own Dataset.**  Section 6.1 covers the description of the ideal dataset regarding our approach, as well as our attempt to generate it with our own test environment.  More specifically, this section explains why publicly available datasets do not suit our needs for our approach.  It then describes how we generate assessment datasets by detailing the test environment, and the monitoring strategy we have designed, as well as the attack scenarios that we have performed against the test environment. These attack scenarios correspond to: (1) SQL injections, as presented in Section 4.1; (2) The exploitation of the ShellShock Vulnerability against the web server of our test environment. This step is followed by the download and installation of a remote access tool that allows the attacker to perform reconnaissance actions such as the listing of the network configuration of the web server.

**Analyzing the Resulting Cause and Dependence Graphs.**  Section 6.2 presents the results of our approach, i.e., computed cause and dependence graphs that can be used by cyber defense analysts to perform attack investigation. We show that our approach allows retrieving heterogeneous events corresponding to an attack scenario, and discuss the relevance of the computed cause and dependence graphs.  More specifically, computed cause and dependence graphs might have false or missing causal dependencies, or both. These two cases respectively correspond to false positives and false negatives in the context of our approach.  The analysis of the cause and dependence graphs yields different results for each attack scenario. In both cases, they contain all the events related to the attack scenarios and, to the best of our knowledge, do not contain any false negatives.  However, the cause and dependence graphs of the SQL injection attack scenario contain many false positives. This is explained by our inability to compute sessions for the MySQL application.  On the other hand, the cause and dependence graphs of the ShellShock and RAT attack scenario contain all the steps of the attack.  Moreover, they do not contain any false positives. We have concluded that our approach shows promising results and could be even more relevant with the adoption of a better monitoring strategy.  Following the discussions surrounding the results of our approach, this section also presents an assessment of the performance of our current implementation of the ETL pipelines.

**On the Limitations of Our Current Implementation.**  Finally, Section 6.3 further discusses concepts surrounding our approach, such as the size of the overall contextual event causal dependency graph, as well as the limitations of our current implementation.  Additionally, it proposes a comparison of our implementation with other approaches.

# CONCLUSION

The purpose of this study is to help cyber defense analysts with the detection and understanding of multi-step attacks through security monitoring and event analysis. Ideally, event analysis would help them ascertain attacks' root causes and impacts, i.e., by identifying all the compromised assets. This study addresses the following observation: a persistent attacker will *eventually* succeed in gaining a foothold inside the targeted network despite all the prevention mechanisms. Thus, the detection of complex attack scenarios, and more generally, security monitoring, remains an active domain of research in computer security.

To meet this objective, we asked ourselves the following questions: How do we observe the system to protect? What do we observe? How do we analyze observations to get insights regarding potential attacks? These questions led us to the review of *monitoring systems* and *alert correlation methodologies*.

**Monitoring Systems.** Detecting attacks relies on the capability to observe the activities that take place in the system to protect. This capability is enabled by monitoring systems. Chapter 1 introduced the reader to monitoring systems and security monitoring. The various types of monitoring systems enable the observation of the monitored system at different abstraction layers, e.g., the application, operating system, or network abstraction layer. Observed activity is recorded as events to keep a history of what happened in the monitored system. In practice, they are often the only information available for attack investigation, digital forensics or, postmortem debugging of production systems failures. Additionally, some monitoring systems are dedicated to the observation and detection of intrusion-related actions, i.e., intrusion detection systems. IDSs produce a particular class of events, called *alerts*, when detecting suspicious behavior such as an intrusion attempt.

**Alert Correlation.** In practice, an event represents the observation of a specific action happening in the monitored system. Events have thereby to be further analyzed to gain insights from them. In the context of security monitoring, this analysis aims to *discover and understand the different steps that make up attacks*. This is the role of alert correlation. Chapter 2 introduced the reader to alert correlation methodologies, which aim at building connections among the events produced by deployed monitoring systems. From the study of the alert correlation research field, we came up with the following conclusions:

1. It is necessary to consider heterogeneous events, i.e., events produced at different abstraction layers, to understand fully complex multi-step attack scenarios;

2. All the alert and event *correlation* techniques dedicated to attack scenario identification *ideally aim to discover causal dependency relationships* among events.

**Towards the Search of Causality.**   Following the lead of alert correlation research, the study of correlation has naturally brought us to study *causality* and its definition regarding the context of computer systems. Our research led us to the study of distributed systems, information flow, and provenance fields. Reviewing the literature, we observed the lack of a clear definition of the *causal dependency relationship among heterogeneous events*. We believe that defining these causal dependencies and enabling their computation would simplify the attack scenario identification process and enhance attack investigation capabilities. The rationale is the following: given an event of interest $e$, e.g., an IoC or an alert, we want to identify all the events that can be considered as the causes or consequences of $e$. This set of events would ideally correspond to the traces left by the attacker on the monitored system. More specifically, if causal dependency relationships among events can be defined and computed, then the discovery of attack scenarios translates to simple graph traversals. Such graphs correspond to causal dependency graphs with events as nodes and causal dependency relationships as directed edges.

## SUMMARY OF OUR CONTRIBUTIONS

**Formal Definition of the Causal Dependency Relationship Among Heterogeneous Events.**   Working towards the definition of the causal dependency relationship among heterogeneous events, we came up with the following insights and models:

1. A system can be modeled as a set of active and passive objects that have states. Active objects have the capability to perform actions;
2. There exist causal dependencies among object states, among object actions, as well as between actions and states;
3. Monitoring systems enable the observation and recording of actions or states in the form of events;
4. If we can determine causal dependencies among actions, among states, and between actions and states, we can propagate this knowledge to the events.

In order to define the causal dependency relationship among heterogeneous events, we have thereby proposed a unified understanding of the causal dependency relationships that can be defined between active objects, passive objects, and event logs. More specifically, inspired from two causality models from the distributed system and the security research areas, i.e., Lamport's [Lamport, 1978] and d'Ausbourg's [d'Ausbourg, 1994] models, we have formally defined three causal dependency relationships that represent different perspectives of the monitored system, namely, the *contextual action causal dependency*, the *contextual event causal dependency*, and the *event causal dependency* relationships. The definitions of these relationships have been laid down in Chapter 4. We believe that our attempt to formalize the notion

of causal dependency among heterogeneous events will help our research community to explore new directions.

**Implementation Strategies of the Causal Dependency Relationships.** Following the presentation of the model we defined, we have shown how this model fits reality in the context of enterprise computer networks in Chapter 5. More specifically, we have introduced two strategies to implement the causal dependency model we defined:

TOP-DOWN STRATEGY – The first strategy is based on the computation and tracking of contextual action causal dependencies. These causal dependencies among contextual actions are then leveraged to compute causal dependencies among contextual events, and events.

BOTTOM-UP STRATEGY – The second strategy consists in leveraging raw events' semantics to compute the contextual event causal dependency model, as well as the contextual action causal dependency model, when possible. More specifically, the bottom-up strategy allows obtaining an approximation of the contextual event causal dependency model, which accuracy depends on the semantic quality of events.

**Lightweight COTS-based Implementation.** Our current implementation of the causal dependency model we have defined adopts the bottom-up strategy. The particularity of our implementation lies in the fact that we have based our monitoring strategy on COTS monitoring systems in order to compute the contextual event causal dependency model and enable the identification of multi-step attack scenarios. We have assessed our implementation in Chapter 6 and have discussed its limitations.

PERSPECTIVES

Several areas of the work presented could be further explored and developed. In particular, the assessment of our implementation may contain more attack scenarios with various levels of complexity. For instance, it would be interesting to explore an attack scenario in which actions are scattered in time, e.g., separated by several days.

Additionally, other implementation strategies could be explored. As we have seen in Figure 5.2, the overview of our architecture implementation is made up of four parts:

1. the deployment of monitoring systems, which focuses on the observation of contextual actions and their recording into events;

2. the extract, transform and load pipelines, which are in charge of the collection of raw events, the building of the contextual event causal dependency graph, and its loading into the database;

3. the graph database, which handles the storage and the query capabilities;

4. the visualization and investigation interface, which aims to give insights to the cyber defense analyst.

Each one of these parts could be further explored. The following paragraphs provide some examples of possible research directions.

**Integrating Better Monitoring Systems.** Deploying monitoring systems is already a challenge as the security monitoring team has to find a compromise between storage, memory, and CPU performance, and observation capabilities. This issue is well illustrated by system call-related monitoring systems as applications may invoke many system calls in a very short period of time. A first avenue of research lies in developing better monitoring systems in order to ease their adoption in the industry. For instance, Ma et al. propose a new architecture for the system call auditing mechanisms of the Linux Kernel in [Ma et al., 2018].

Another research direction consists in exploring static analysis and dynamic analysis to have further grained information flow tracking, as proposed with BEEP in [Lee et al., 2013a].

**Exploring Pattern Recognition.** The work presented in this manuscript focuses on discovering and retrieving events that correspond to the same attack scenarios. These events may have a semantic level that is too low to be easily analyzed by cyber defense analysts. Thus, an interesting avenue of research is the ability to recognize patterns, in the contextual event causal dependency model, that would provide a higher semantic level of information. For example, given sequences of events may be automatically recognized as APT stages, as proposed in [Milajerdi et al., 2019].

**Developing Visualization Strategies.** As we have previously mentioned, we did not explore the visualization part of the architecture. As they are, cause and dependence graphs might not be easily analyzed by a cyber defense analyst as it potentially contains many nodes. Information could be better rearranged to get insight faster. For example, we could compute different kinds of perspectives, such as only displaying causally dependent network-related events or causally dependent application-related events. Additionally, the approach and methodology we propose in this manuscript could be coupled with knowledge databases (i.e., topology, cartography, and vulnerability knowledge) to create a more holistic view of the monitored system and further enable situational awareness.

# A | MONITORING SYSTEMS CONFIGURATION

Appendix A describes how the deployed monitoring systems have been configured to generate the dataset we used to assess our current implementation of the causality model. Section A.1 describes how auditd is configured in order to capture information flow-related system calls. Section A.2 describes how netfilter is configured in order to log any new UDP or TCP communications. Section A.3 describes how Apache is configured in order to use the log format we need.

## A.1 LIST OF MONITORED LINUX SYSTEM CALLS

This section describes how auditd is configured to log of the appendix aims to log information flow-related system calls. The following code snippet corresponds to the auditd configuration file. It is made up of 5 sections:

1. auditd settings;
2. self auditing, which includes audit rules to watch auditd configuration files;
3. filters, which includes rules to avoid the logging of log messages we are not interested in;
4. alert rules, which includes audit rules that correspond to suspicious activity;
5. information flow-related system calls, which includes audit rules to log the system calls of interest that allow us to compute the OS-related part of the contextual event causal dependency model.

```
###### [filepath] /etc/audit/rules.d/custom.rules

# First rule - delete all
-D

# Auditd Settings ----------------------------------------------------


## Increase the backlog buffers to survive stress events.
## Make this bigger for busy systems
-b 100000

## This determine how long to wait in burst of events
# --backlog_wait_time 0

## Set failure mode to syslog
-f 1

## This rule will cause auditctl to continue loading rules when it runs
## across a an unsupported or a rule with a syntax error however it will
```

```
## report an error at exit. The normal action is to report the line and
## issue with the rule and exit immediately with an error to get the
    admin's
## attention to fix the rules.
## Continue through errors in rules
-c


# Self Auditing -----------------------------------------------------------
## Audit the audit logs
### Successful and unsuccessful attempts to read information from the
    audit records
-w /var/log/audit/ -k alert_auditlog


## Auditd configuration
### Modifications to audit configuration that occur while the audit
    collection functions are operating
-w /etc/audit/ -p wa -k alert_auditconfig
-w /etc/libaudit.conf -p wa -k alert_auditconfig
-w /etc/audisp/ -p wa -k alert_audispconfig


## Monitor for use of audit management tools
-w /sbin/auditctl -p x -k alert_audittools
-w /sbin/auditd -p x -k alert_audittools


# Filters ------------------------------------------------------------------
### We put these early because audit is a first match wins system.


## Ignore PROCTITLE messages
-a always,exclude -F msgtype=PROCTITLE


## Ignore SELinux AVC records
-a always,exclude -F msgtype=AVC


## Cron jobs fill the logs with stuff we normally don't want (works with
    SELinux)
-a never,user -F subj_type=crond_t
-a never,exit -F subj_type=crond_t


## This is not very interesting and wastes a lot of space if the server
    is public facing
-a always,exclude -F msgtype=CRYPTO_KEY_USER


##### Exclude /dev/random for apache2 monitoring
-a never,exit -F path=/dev/urandom -k exclude_dev_urandom
-a never,exit -F path=/dev/null -k exclude_dev_null


## More information on how to filter events
### https://access.redhat.com/solutions/2482221



# Alert Rules --------------------------------------------------------------
## 32bit API Exploitation
### If you are on a 64 bit platform, everything _should_ be running
### in 64 bit mode. This rule will detect any use of the 32 bit syscalls
### because this might be a sign of someone exploiting a hole in the 32
```

```
### bit API.
-a always,exit -F arch=b32 -S all -k alert_32bit_api

## Reconnaissance
-w /usr/bin/whoami -p x -k alert_recon
-w /bin/uname -p x -k alert_recon
-w /etc/issue -p r -k alert_recon
-w /etc/hostname -p r -k alert_recon

## Suspicious activity
-w /usr/bin/wget -p x -k alert_susp_activity
-w /usr/bin/curl -p x -k alert_susp_activity
-w /usr/bin/base64 -p x -k alert_susp_activity
-w /bin/nc -p x -k alert_susp_activity
-w /bin/netcat -p x -k alert_susp_activity
-w /usr/bin/ncat -p x -k alert_susp_activity
-w /usr/bin/ssh -p x -k alert_susp_activity
-w /usr/bin/socat -p x -k alert_susp_activity
-w /usr/bin/wireshark -p x -k alert_susp_activity
-w /usr/bin/rawshark -p x -k alert_susp_activity
-w /usr/bin/rdesktop -p x -k alert_sbin_susp

## Sbin suspicious activity
-w /sbin/iptables -p x -k alert_sbin_susp
-w /sbin/ifconfig -p x -k alert_sbin_susp
-w /usr/sbin/tcpdump -p x -k alert_sbin_susp
-w /usr/sbin/traceroute -p x -k alert_sbin_susp

-w /etc/shadow -p rwa -F success!=1 -k alert_susp_activity_shadow

## Injection
### These rules watch for code injection by the ptrace facility.
### This could indicate someone trying to do something bad or just
    debugging
-a always,exit -F arch=b32 -S ptrace -k alert_tracing
-a always,exit -F arch=b64 -S ptrace -k alert_tracing
-a always,exit -F arch=b32 -S ptrace -F a0=0x4 -k alert_code_injection
-a always,exit -F arch=b64 -S ptrace -F a0=0x4 -k alert_code_injection
-a always,exit -F arch=b32 -S ptrace -F a0=0x5 -k alert_data_injection
-a always,exit -F arch=b64 -S ptrace -F a0=0x5 -k alert_data_injection
-a always,exit -F arch=b32 -S ptrace -F a0=0x6 -k alert_register_
    injection
-a always,exit -F arch=b64 -S ptrace -F a0=0x6 -k alert_register_
    injection

## Privilege Abuse
### The purpose of this rule is to detect when an admin may be abusing
    power by looking in user's home dir.
-a always,exit -F dir=/home -F uid=0 -F auid>=1000 -F auid!=4294967295 -
    C auid!=obj_uid -k alert_power_abuse


# Information Flow-related Syscalls ----------------------------------
## Record all the following events, regardless of their exit code
-a always,exit -F arch=b64 -S exit -S exit_group -S connect -S kill
```

```
## Only record these events if they succeed
-a always,exit -F arch=b64 -S read -S readv -S pread -S preadv -S write
    -S writev -S pwrite -S pwritev -S lseek -S sendto -S recvfrom -S
    sendmsg -S recvmsg -S bind -S accept -S accept4 -S socket -S mmap -S
     mprotect -S madvise -S unlink -S unlinkat -S link -S linkat -S
    symlink -S symlinkat -S clone -S fork -S vfork -S execve -S open -S
    close -S creat -S openat -S mknodat -S mknod -S dup -S dup2 -S dup3
    -S fcntl -S rename -S renameat -S setuid -S setreuid -S setgid -S
    setregid -S chmod -S fchmod -S fchmodat -S pipe -S pipe2 -S truncate
     -S ftruncate -S init_module -S finit_module -S tee -S splice -S
    vmsplice -S socketpair -S ptrace -F success=1
```

## A.2   NETFILTER CONFIGURATION

This section describes how netfilter is configured in order to log any new UDP or TCP communications. The following code snippets correspond to the `iptables` commands to enable the logging of new UDP and TCP communications.

```
# iptables -I INPUT -p tcp -m state --state NEW -j LOG
    --log-prefix='[nfconnections] '
# iptables -I OUTPUT -p tcp -m state --state NEW -j LOG
    --log-prefix='[nfconnections] '
# iptables -I INPUT -p udp -m state --state NEW -j LOG
    --log-prefix='[nfconnections] '
# iptables -I OUTPUT -p udp -m state --state NEW -j LOG
    --log-prefix='[nfconnections] '
```

In order to log netfilter events in a dedicated file, we also configure `rsyslog` with the following directive:

```
# [filepath] /etc/rsyslog.d/00-nfconnections.conf
:msg,contains,"[nfconnections] " /var/log/nfconnections.log
```

## A.3   APACHE CONFIGURATION

The following code snippet corresponds to the `LogFormat` configuration of the Apache server. More specifically, we have kept the original Apache configuration and only changed the `LogFormat` part by replacing it with our configuration.

```
# [filepath] /etc/apache2/apache2.conf
LogFormat "%t %{usec}t %{UNIQUE_ID}e %P %A %p %h %{remote}p \"%r\" %>s %
    O \"%{Referer}i\" \"%{User-Agent}i\"" combined
```

Additionally, the Apacher server is configured to enable the use of CGI. This is done by executing the following command with root privileges:

```
# a2enmod cgi
```

# PUBLICATIONS

Xosanavongsa, C., Totel, E., & Bettan, O. (2019a). Discovering correlations: A formal definition of causal dependency among heterogeneous events. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, (pp. 340–355). IEEE.

Xosanavongsa, C., Totel, E., & Bettan, O. (2019b). Définition formelle de la relation de dépendance causale entre Événements journalisés. In *2019 Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI)*. INRIA.

Xosanavongsa, C., Totel, E., & Kheir, N. (2018). Corrélation d'événements et découverte de scénarios d'attaque multi-étapes. In *2018 Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI)*. INRIA.

# BIBLIOGRAPHY

(2007). The intrusion detection message exchange format (idmef).

(2016). The intrusion detection message exchange format v2 (idmef v2).

Abad, C., Li, Y., Lakkaraju, K., Yin, X., & Yurcik, W. (2004). Correlation between netflow system and network views for intrusion detection. In *Workshop on Link Analysis, Counter-terrorism, and Privacy held in conjunction with SDM 2004*.

Abad, C., Taylor, J., Sengul, C., Yurcik, W., Zhou, Y., & Rowe, K. (2003). Log correlation for intrusion detection: A proof of concept. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, (pp. 255–264). IEEE.

Abadi, M., Budiu, M., Erlingsson, Ú., & Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, *13*(1), 4.

Almgren, M. & Lindqvist, U. (2001). Application-integrated data collection for security monitoring. In *International Workshop on Recent Advances in Intrusion Detection*, (pp. 22–36). Springer.

Anderson, J. P. (1980). Computer security threat monitoring and surveillance. *Technical Report, James P. Anderson Company*.

Andersson, D., Fong, M., & Valdes, A. (2002). Heterogeneous sensor correlation: A case study of live traffic analysis. In *IEEE Information Assurance Workshop*.

ANSSI (2013). Recommandations de sécurité pour la mise en œuvre d'un système de journalisation.

ANSSI (2015). Les métiers de la sécurité du numérique.

Australian Cyber Security Center (2019). Windows event logging and forwarding.

Axelsson, S. (1999). The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, (pp. 1–7). ACM.

Axelsson, S. (2000). Intrusion Detection Systems: A Survey and Taxonomy.

Baláž, A., Ádám, N., Pietriková, E., & Madoš, B. (2018). Modsecurity idmef module. In *2018 IEEE 16th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, (pp. 000043–000048). IEEE.

Balliu, M., Schoepe, D., & Sabelfeld, A. (2017). We are family: Relating information-flow trackers. In *European Symposium on Research in Computer Security*, (pp. 124–145). Springer.

Baquero, C. & Preguiça, N. (2016). Why logical clocks are easy. *acmqueue*.

Bass, T. (2000). Intrusion detection systems and multisensor data fusion: Creating cyberspace situational awareness. *Communications of the ACM*, *43*(4), 99–105.

Bates, A. & Hassan, W. U. (2019). Can data provenance put an end to the data breach? *IEEE Security & Privacy*, *17*(4), 88–93.

Bates, A., Hassan, W. U., Butler, K., Dobra, A., Reaves, B., Cable, P., Moyer, T., & Schear, N. (2017). Transparent web service auditing via network provenance functions. In *Proceedings of the 26th International Conference on World Wide Web*, (pp. 887–895). International World Wide Web Conferences Steering Committee.

Bates, A., Tian, D. J., Butler, K. R., & Moyer, T. (2015). Trustworthy whole-system provenance for the linux kernel. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, (pp. 319–334).

Bejtlich, R. (2013). *The practice of network security monitoring: understanding incident detection and response*. No Starch Press.

Benferhat, S., Autrel, F., & Cuppens, F. (2003). Enhanced correlation in an intrusion detection process. In *International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, (pp. 157–170). Springer.

Beschastnikh, I., Brun, Y., Ernst, M. D., & Krishnamurthy, A. (2014). Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, (pp. 468–479). ACM.

Beschastnikh, I., Brun, Y., Ernst, M. D., Krishnamurthy, A., & Anderson, T. E. (2011). Mining temporal invariants from partially ordered logs. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques* (pp. 1–10).

Brogi, G. & Tong, V. V. T. (2017). Sharing and replaying attack scenarios with moirai. In *2017 Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI)*. INRIA.

Chabot, Y., Bertaux, A., Kechadi, T., & Nicolle, C. (2015). Event reconstruction: A state of the art. In *Handbook of Research on Digital Crime, Cyberspace Security, and Information Assurance* (pp. 231–245). IGI Global.

Chabot, Y., Bertaux, A., Nicolle, C., & Kechadi, T. (2015). An ontology-based approach for the reconstruction and analysis of digital incidents timelines. *Digital Investigation*, *15*, 83–100.

Chari, S. N. & Cheng, P.-C. (2003). Bluebox: A policy-driven, host-based intrusion detection system. *ACM Transactions on Information and System Security (TISSEC)*, *6*(2), 173–200.

Chen, B., Lee, J., & Wu, A. S. (2006). Active event correlation in bro ids to detect multi-stage attacks. In *Fourth IEEE International Workshop on Information Assurance (IWIA'06)*, (pp. 16–pp). IEEE.

Chen, L., Sultana, S., & Sahita, R. (2018). Henet: A deep learning approach on intel® processor trace for effective exploit detection. In *2018 IEEE Security and Privacy Workshops (SPW)*, (pp. 109–115). IEEE.

Cheney, J., Chong, S., Foster, N., Seltzer, M., & Vansummeren, S. (2009). Provenance: a future history. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, (pp. 957–964). ACM.

Chow, J., Garfinkel, T., & Chen, P. M. (2008). Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, (pp. 1–14).

Chuvakin, A., Schmidt, K., & Phillips, C. (2012). *Logging and log management: the authoritative guide to understanding the concepts surrounding logging and log management*. Newnes.

Chuvakin, W. J. H. A., Marty, J. T. J. R., & Mcquaid, R. M. (2008). Common event expression.

Chyssler, T., Burschka, S., Semling, M., Lingvall, T., & Burbeck, K. (2004). Alarm reduction and correlation in intrusion detection systems. In *DIMVA*, (pp. 9–24).

CISSP, S. H., CISSP, J. B., & Hare, C. (2003). *Official (ISC) 2 guide to the CISSP exam*. Auerbach Publications.

Clause, J., Li, W., & Orso, A. (2007). Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, (pp. 196–206). ACM.

Creech, G. (2014). *Developing a high-accuracy cross platform Host-Based Intrusion Detection System capable of reliably detecting zero-day attacks.* PhD thesis, University of New South Wales, Canberra, Australia.

Cuppens, F. & Miege, A. (2002). Alert correlation in a cooperative intrusion detection framework. In *Proceedings 2002 IEEE symposium on security and privacy*, (pp. 202–215). IEEE.

Cuppens, F. & Ortalo, R. (2000). Lambda: A language to model a database for detection of attacks. In *International Workshop on Recent Advances in Intrusion Detection*, (pp. 197–216). Springer.

Dain, O. & Cunningham, R. K. (2002). Fusing a heterogeneous alert stream into scenarios. In *Applications of Data Mining in Computer Security* (pp. 103–122). Springer.

Dain, O. M. & Cunningham, R. K. (2001). Building scenarios from a heterogeneous alert stream. In *Proceedings of the 2001 IEEE workshop on Information Assurance and Security*, volume 6. United States Military Academy, West Point, NY.

d'Ausbourg, B. (1994). Implementing secure dependencies over a network by designing a distributed security subsystem. In *European Symposium on Research in Computer Security*, (pp. 247–266). Springer.

de Alvarenga, S. C., Barbon, S., Miani, R. S., Cukier, M., & Zarpelão, B. B. (2018). Process mining and hierarchical clustering to help intrusion alert visualization. *Computers & Security*, *73*, 474 – 491.

Debar, H., Dacier, M., & Wespi, A. (1999). Towards a Taxonomy of Intrusion-detection Systems. *Computer Networks*, *31*(8), 805–822.

Debar, H. & Wespi, A. (2001). Aggregation and correlation of intrusion-detection alerts. In *International Workshop on Recent Advances in Intrusion Detection*, (pp. 85–103). Springer.

Defense Advanced Research Projects Agency (2014). Transparent computing.

Denning, D. E. (1976). A lattice model of secure information flow. *Communications of the ACM*, *19*(5), 236–243.

Denning, D. E. (1987). An intrusion-detection model. *IEEE Transactions on software engineering*, (2), 222–232.

Devecsery, D., Chow, M., Dou, X., Flinn, J., & Chen, P. M. (2014). Eidetic systems. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, (pp. 525–540).

Dreger, H., Kreibich, C., Paxson, V., & Sommer, R. (2005). Enhancing the accuracy of network-based intrusion detection with host-based context. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, (pp. 206–221). Springer.

Du, H., Liu, D. F., Holsopple, J., & Yang, S. J. (2010). Toward ensemble characterization and projection of multistage cyber attacks. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, (pp. 1–8). IEEE.

Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, (pp. 1285–1298). ACM.

Eckmann, S. T., Vigna, G., & Kemmerer, R. A. (2002). Statl: An attack language for state-based intrusion detection. *Journal of computer security*, *10*(1-2), 71–103.

European Commission (2010). Standard on logging and monitoring.

European Union Computer Emergency Response Team (2017). Detecting lateral movementsin windows infrastructure.

Fidge, C. J. (1988). Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, (pp. 55–66)., University of Queensland, Australia.

Forrest, S., Hofmeyr, S. A., Somayaji, A., & Longstaff, T. A. (1996). A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, (pp. 120–128).

Gagnon, F., Massicotte, F., & Esfandiari, B. (2009). Using contextual information for ids alarm classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, (pp. 147–156). Springer.

Gao, P., Xiao, X., Li, Z., Xu, F., Kulkarni, S. R., & Mittal, P. (2018). {AIQL}: Enabling efficient attack investigation from system monitoring data. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, (pp. 113–126).

Gardner, R. D. & Harle, D. A. (1996). Methods and systems for alarm correlation. In *Proceedings of GLOBECOM'96. 1996 IEEE Global Telecommunications Conference*, volume 1, (pp. 136–140). IEEE.

Gehani, A. & Tariq, D. (2012). Spade: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference*, (pp. 101–120). Springer-Verlag New York, Inc.

Georget, L. (2017). *Suivi de flux d'information correct pour les systèmes d'exploitation Linux*. PhD thesis.

Georget, L., Jaume, M., Tronel, F., Piolle, G., & Tong, V. V. T. (2017). Verifying the reliability of operating system-level information flow control systems in linux. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, (pp. 10–16). IEEE.

Godefroy, E. (2016). *Définition et évaluation d'un mécanisme de génération de règles de corrélation liées à l'environnement*. PhD thesis, CentraleSupélec.

Godefroy, E., Totel, E., Hurfin, M., & Majorczyk, F. (2015a). Generation and assessment of correlation rules to detect complex attack scenarios. In *2015 IEEE Conference on Communications and Network Security (CNS)*, (pp. 707–708). IEEE.

Godefroy, E., Totel, E., Hurfin, M., & Majorczyk, F. (2015b). Generation and assessment of correlation rules to detect complex attack scenarios. In *2015 IEEE Conference on Communications and Network Security (CNS)*, (pp. 707–708).

Goldman, R. P., Heimerdinger, W., Harp, S. A., Geib, C. W., Thomas, V., & Carter, R. L. (2001). Information modeling for intrusion report aggregation. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 1, (pp. 329–342). IEEE.

Goubault-Larrecq, J. & Olivain, J. (2008). A smell of orchids. In *International Workshop on Runtime Verification*, (pp. 1–20). Springer.

Gu, G., Cárdenas, A. A., & Lee, W. (2008). Principled reasoning and practical applications of alert fusion in intrusion detection systems. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, (pp. 136–147). ACM.

Halpern, J. Y. (2006). Causality, responsibility, and blame: a structural-model approach. In *Third International Conference on the Quantitative Evaluation of Systems-(QEST'06)*, (pp. 3–8). IEEE.

Hassan, W. U., Guo, S., Li, D., Chen, Z., Jee, K., Li, Z., & Bates, A. (2019). Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*.

Hauser, C., Tronel, F., Reid, J., & Fidge, C. (2012). A taint marking approach to confidentiality violation detection. In *Proceedings of the Tenth Australasian Information Security Conference-Volume 125*, (pp. 83–90). Australian Computer Society, Inc.

Heady, R., Luger, G., Maccabe, A., & Servilla, M. (1990). The architecture of a network level intrusion detection system. Technical report, Los Alamos National Lab., NM (United States); New Mexico Univ., Albuquerque . . . .

Herley, C. & Van Oorschot, P. C. (2017). Sok: Science, security and the elusive goal of security as a scientific pursuit. In *2017 IEEE Symposium on Security and Privacy (SP)*, (pp. 99–120). IEEE.

Hiet, G., Tong, V. V. T., Me, L., & Morin, B. (2008). Policy-based intrusion detection in web applications by monitoring java information flows. In *2008 Third International Conference on Risks and Security of Internet and Systems*, (pp. 53–60). IEEE.

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hossain, M. N., Milajerdi, S. M., Wang, J., Eshete, B., Gjomemo, R., Sekar, R., Stoller, S., & Venkatakrishnan, V. (2017). {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, (pp. 487–504).

Hossain, M. N., Wang, J., Weisse, O., Sekar, R., Genkin, D., He, B., Stoller, S. D., Fang, G., Piessens, F., Downing, E., et al. (2018). Dependence-preserving data compaction for scalable forensic analysis. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, (pp. 1723–1740).

Huang, M.-Y., Jasper, R. J., & Wicks, T. M. (1999). A large scale distributed intrusion detection framework based on attack strategy analysis. *Computer Networks*, *31*(23-24), 2465–2475.

Hutchins, E. M., Cloppert, M. J., & Amin, R. M. (2011). Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, *1*(1), 80.

Jajodia, S. & Noel, S. (2010). Advanced cyber attack modeling analysis and visualization. Technical report, GEORGE MASON UNIV FAIRFAX VA.

Jajodia, S., Noel, S., & O'berry, B. (2005). Topological analysis of network attack vulnerability. In *Managing Cyber Threats* (pp. 247–266). Springer.

Jakobson, G. & Weissman, M. (1993). Alarm correlation. *IEEE network*, 7(6), 52–59.

Japan Computer Emergency Response Team Coordination Center (2017). Detecting lateral movement through tracking event logs.

Ji, Y., Lee, S., Downing, E., Wang, W., Fazzini, M., Kim, T., Orso, A., & Lee, W. (2017). Rain: Refinable attack investigation with on-demand interprocess information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, (pp. 377–390). ACM.

Ji, Y., Lee, S., Fazzini, M., Allen, J., Downing, E., Kim, T., Orso, A., & Lee, W. (2018). Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, (pp. 1705–1722).

Josephson, J. R. & Josephson, S. G. (1996). *Abductive inference: Computation, philosophy, technology*. Cambridge University Press.

Julisch, K. (2003). Clustering intrusion detection alarms to support root cause analysis. *ACM transactions on information and system security (TISSEC)*, 6(4), 443–471.

Julisch, K. & Dacier, M. (2002). Mining intrusion detection alarms for actionable knowledge. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, (pp. 366–375). ACM.

Kannan, H., Dalton, M., & Kozyrakis, C. (2009). Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, (pp. 105–114). IEEE.

Kayser, D. & Lévy, F. (2009). Causality: Purposes, core notions, properties. In *Extended abstracts of the International Multidisciplinary Workshop on Causality*, (pp.9).

Kemerlis, V. P., Portokalidis, G., Jee, K., & Keromytis, A. D. (2012). libdft: Practical dynamic data flow tracking for commodity systems. In *Acm Sigplan Notices*, volume 47, (pp. 121–132). ACM.

Kent, K. & Souppaya, M. (2006). Guide to computer security log management. *NIST special publication*, 92.

King, S. T. & Chen, P. M. (2003). Backtracking intrusions. In *ACM SIGOPS Operating Systems Review*, volume 37, (pp. 223–236). ACM.

Kordy, B., Piètre-Cambacédès, L., & Schweitzer, P. (2014). Dag-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer science review*, 13, 1–38.

Kruegel, C., Robertson, W., & Vigna, G. (2004). Using alert verification to identify successful intrusion attempts. *Praxis der Informationsverarbeitung und Kommunikation*, 27(4), 219–227.

Kwon, Y., Wang, F., Wang, W., Lee, K. H., Lee, W.-C., Ma, S., Zhang, X., Xu, D., Jha, S., Ciocarlie, G. F., et al. (2018). Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, *21*(7), 558–565.

Lamprakis, P., Dargenio, R., Gugelmann, D., Lenders, V., Happe, M., & Vanbever, L. (2017). Unsupervised detection of apt c&c channels using web request graphs. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, (pp. 366–387). Springer.

Lanoë, D., Hurfin, M., & Totel, E. (2018). A scalable and efficient correlation engine to detect multi-step attacks in distributed systems. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, (pp. 31–40). IEEE.

Lanoë, D., Hurfin, M., Totel, E., & Maziero, C. (2019). An Efficient and Scalable Intrusion Detection System on Logs of Distributed Applications. In Dhillon, G., Karlsson, F., Hedström, K., & Zúquete, A. (Eds.), *ICT Systems Security and Privacy Protection*, IFIP Advances in Information and Communication Technology, (pp. 49–63). Springer International Publishing.

Lee, K. H., Zhang, X., & Xu, D. (2013a). High accuracy attack provenance via binary-based execution partition. In *NDSS*.

Lee, K. H., Zhang, X., & Xu, D. (2013b). Loggc: garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, (pp. 1005–1016). ACM.

Leichtnam, L., Totel, E., Prigent, N., & Mé, L. (2017). Starlord: Linked security data exploration in a 3d graph. In *2017 IEEE Symposium on Visualization for Cyber Security (VizSec)*, (pp. 1–4). IEEE.

Lewis, D. (2013). *Counterfactuals*. John Wiley & Sons.

Li, B., Vadrevu, P., Lee, K. H., & Perdisci, R. (2018). Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *NDSS*.

Li, Z., Taylor, J., Partridge, E., Zhou, Y., Yurcik, W., Abad, C., Barlow, J. J., & Rosendale, J. (2004). Uclog: A unified, correlated logging architecture for intrusion detection. In *the 12th International Conference on Telecommunication Systems-Modeling and Analysis (ICTSM)*.

Liu, P., Jajodia, S., & Wang, C. (2017). *Theory and Models for Cyber Situation Awareness*, volume 10030. Springer.

Liu, X., Xiao, D., & Peng, X. (2008). Towards a collaborative and systematic approach to alert verification. *JSW*, *3*(9), 77–84.

Liu, Y., Zhang, M., Li, D., Jee, K., Li, Z., Wu, Z., Rhee, J., & Mittal, P. (2018). Towards a timely causality analysis for enterprise security. In *NDSS*.

Luh, R., Marschalek, S., Kaiser, M., Janicke, H., & Schrittwieser, S. (2017). Semantics-aware detection of targeted attacks: a survey. *Journal of Computer Virology and Hacking Techniques*, *13*(1), 47–85.

Ma, S., Lee, K. H., Kim, C. H., Rhee, J., Zhang, X., & Xu, D. (2015). Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference*, (pp. 401–410). ACM.

Ma, S., Zhai, J., Kwon, Y., Lee, K. H., Zhang, X., Ciocarlie, G., Gehani, A., Yegneswaran, V., Xu, D., & Jha, S. (2018). Kernel-supported cost-effective audit logging for causality tracking. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, (pp. 241–254).

Ma, S., Zhai, J., Wang, F., Lee, K. H., Zhang, X., & Xu, D. (2017). {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, (pp. 1111–1128).

Ma, S., Zhang, X., & Xu, D. (2016). Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*.

Massicotte, F., Labiche, Y., & Briand, L. C. (2008). Toward automatic generation of intrusion detection verification rules. In *2008 Annual Computer Security Applications Conference (ACSAC)*, (pp. 279–288). IEEE.

Mathew, S. & Upadhyaya, S. (2009). Attack scenario recognition through heterogeneous event stream analysis. In *MILCOM 2009-2009 IEEE Military Communications Conference*, (pp. 1–7). IEEE.

Mattern, F. (1989). Virtual time and global states of distributed systems. In *Proc. Int. Workshop on Parallel and Distributed Algorithms*.

Milajerdi, S. M., Gjomemo, R., Eshete, B., Sekar, R., & Venkatakrishnan, V. (2019). Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, (pp. 1137–1152). IEEE.

Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., et al. (2011). The open provenance model core specification (v1. 1). *Future generation computer systems*, *27*(6), 743–756.

Morin, B. (2004). *Corrélation d'alertes issues de sondes de détection d'intrusions avec prise en compte d'informations sur le système surveillé*. PhD thesis, Rennes, INSA.

Morin, B., Mé, L., Debar, H., & Ducassé, M. (2002). M2d2: A formal data model for ids alert correlation. In *International Workshop on Recent Advances in Intrusion Detection*, (pp. 115–137). Springer.

Morin, B., Mé, L., Debar, H., & Ducassé, M. (2009). A logic-based model to support alert correlation in intrusion detection. *Information Fusion*, *10*(4), 285–299.

Muniswamy-Reddy, K.-K., Braun, U. J., Holland, D. A., Macko, P., Maclean, D., Margo, D. W., Seltzer, M. I., & Smogor, R. (2009). Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX'09)*. USENIX Association.

Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., & Seltzer, M. I. (2006). Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, (pp. 43–56).

Mustapha, Y. B., Débar, H., & Jacob, G. (2012). Limitation of honeypot/honeynet databases to enhance alert correlation. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, (pp. 203–217). Springer.

Myers, A. C. & Myers, A. C. (1999). Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (pp. 228–241). ACM.

Navarro, J., Deruyver, A., & Parrend, P. (2018). A systematic survey on multi-step attack detection. *Computers & Security*, *76*, 214–249.

Navarro, J., Legrand, V., Deruyver, A., & Parrend, P. (2018). Omma: open architecture for operator-guided monitoring of multi-step attacks. *EURASIP Journal on Information Security*, *2018*(1), 6.

Newhouse, W., Keith, S., Scribner, B., & Witte, G. (2017). National initiative for cybersecurity education (nice) cybersecurity workforce framework. *NIST Special Publication*, *800*, 181.

Newsome, J. & Song, D. X. (2005). Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, (pp. 3–4). Citeseer.

Nicolett, M. & Kavanagh, K. M. (2011). Magic quadrant for security information and event management. *Gartner RAS Core Reasearch Note (May 2009)*.

Ning, P., Cui, Y., & Reeves, D. S. (2002). Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, (pp. 245–254). ACM.

Ning, P. & Xu, D. (2010). Toward automated intrusion alert analysis. In *Network Security* (pp. 175–205). Springer.

Noel, S., Robertson, E., & Jajodia, S. (2004). Correlating intrusion events and building attack scenarios through attack graph distances. In *20th Annual Computer Security Applications Conference*, (pp. 350–359). IEEE.

Ourston, D., Matzner, S., Stump, W., & Hopkins, B. (2003). Applications of hidden markov models to detecting multi-stage network attacks. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, (pp. 10–pp). IEEE.

OWASP (2014). Security logging project.

Pasquier, T., Han, X., Goldstein, M., Moyer, T., Eyers, D., Seltzer, M., & Bacon, J. (2017). Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, (pp. 405–418). ACM.

Pasquier, T., Han, X., Moyer, T., Bates, A., Hermant, O., Eyers, D., Bacon, J., & Seltzer, M. (2018). Runtime analysis of whole-system provenance. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, (pp. 1601–1616). ACM.

Pearl, J. (2000). *Causality: models, reasoning and inference*, volume 29. Springer.

Pei, K., Gu, Z., Saltaformaggio, B., Ma, S., Wang, F., Zhang, Z., Si, L., Zhang, X., & Xu, D. (2016). Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, (pp. 583–595). ACM.

Pérez, M. G., Tapiador, J. E., Clark, J. A., Pérez, G. M., & Gómez, A. F. S. (2014). Trustworthy placements: Improving quality and resilience in collaborative attack detection. *Computer Networks*, *58*, 70–86.

Pohly, D. J., McLaughlin, S., McDaniel, P., & Butler, K. (2012). Hi-fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*, (pp. 259–268). ACM.

Porras, P. A., Fong, M. W., & Valdes, A. (2002). A mission-impact-based approach to infosec alarm correlation. In *International Workshop on Recent Advances in Intrusion Detection*, (pp. 95–114). Springer.

Pouget, F. & Dacier, M. (2003). Alert correlation: Review of the state of the art. *TechnicalReport EURECOM*, *1271*.

Qin, X. & Lee, W. (2004a). Attack plan recognition and prediction using causal networks. In *20th Annual Computer Security Applications Conference*, (pp. 370–379). IEEE.

Qin, X. & Lee, W. (2004b). Discovering novel attack strategies from infosec alerts. In *European Symposium on Research in Computer Security*, (pp. 439–456). Springer.

Raynal, M. (2013). *Distributed algorithms for message-passing systems*, volume 500. Springer.

Ren, H., Stakhanova, N., & Ghorbani, A. A. (2010). An online adaptive approach to alert correlation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, (pp. 153–172). Springer.

Sadighian, A., Fernandez, J. M., Lemay, A., & Zargar, S. T. (2013). Ontids: A highly flexible context-aware and ontology-based alert correlation framework. In *International Symposium on Foundations and Practice of Security*, (pp. 161–177). Springer.

Sadoddin, R. & Ghorbani, A. (2006). Alert correlation survey: framework and techniques. In *Proceedings of the 2006 international conference on privacy, security and trust: bridge the gap between PST technologies and business services*, (pp.37). ACM.

Salah, S., Maciá-Fernández, G., & DíAz-Verdejo, J. E. (2013). A model-based survey of alert correlation techniques. *Computer Networks*, *57*(5), 1289–1317.

Schwarz, R. & Mattern, F. (1994). Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, *7*(3), 149–174.

Sheyner, O., Haines, J., Jha, S., Lippmann, R., & Wing, J. M. (2002). Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, (pp. 273–284). IEEE.

Shittu, R. O. (2016). *Mining intrusion detection alert logs to minimise false positives & gain attack insight*. PhD thesis, City University London.

Sommer, R. & Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy*, (pp. 305–316). IEEE.

Strnad, A., Messiter, Q., Watson, R., Carata, L., Anderson, J., & Kidney, B. (2019). Causal, adaptive, distributed, and efficient tracing system (cadets). Technical report, BAE Systems Burlington United States.

Stroeh, K., Madeira, E. R. M., & Goldenstein, S. K. (2013). An approach to the correlation of security events based on machine learning techniques. *Journal of Internet Services and Applications*, *4*(1), 7.

Suh, G. E., Lee, J. W., Zhang, D., & Devadas, S. (2004). Secure program execution via dynamic information flow tracking. In *ACM Sigplan Notices*, volume 39, (pp. 85–96). ACM.

Sundaramurthy, S. C., Zomlot, L., & Ou, X. (2011). Practical ids alert correlation in the face of dynamic threats. In *Proceedings of the International Conference on Security and Management (SAM)*, (pp. ĩ). Citeseer.

Tan, J., Pan, X., Kavulya, S., Gandhi, R., & Narasimhan, P. (2008). SALSA: Analyzing Logs As State Machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs*, WASL'08, (pp. 6–6)., Berkeley, CA, USA. USENIX Association.

Templeton, S. J. & Levitt, K. (2001). A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, (pp. 31–38). ACM.

Totel, E., Hkimi, M., Hurfin, M., Leslous, M., & Labiche, Y. (2016). Inferring a Distributed Application Behavior Model for Anomaly Based Intrusion Detection. In *Dependable Computing Conference (EDCC), 2016 12th European*, (pp. 53–64). IEEE.

Totel, E., Vivinis, B., & Mé, L. (2004). A language driven intrusion detection system for event and alert correlation. In *IFIP International Information Security Conference*, (pp. 209–224). Springer.

Vadrevu, P., Liu, J., Li, B., Rahbarinia, B., Lee, K. H., & Perdisci, R. (2017). Enabling reconstruction of attacks on users via efficient browsing snapshots.

Valdes, A. & Skinner, K. (2001). Probabilistic alert correlation. In *International Workshop on Recent Advances in Intrusion Detection*, (pp. 54–68). Springer.

Valeur, F., Vigna, G., Kruegel, C., & Kemmerer, R. A. (2004). Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on dependable and secure computing*, *1*(3), 146–169.

Viinikka, J., Debar, H., Mé, L., Lehikoinen, A., & Tarvainen, M. (2009). Processing intrusion detection alert aggregates with time series modeling. *Information Fusion*, *10*(4), 312–324.

Wang, F., Kwon, Y., Ma, S., Zhang, X., & Xu, D. (2018). Lprov: Practical library-aware provenance tracing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, (pp. 605–617). ACM.

Welz, M. & Hutchison, A. (2001). Interfacing trusted applications with intrusion detection systems. In *International Workshop on Recent Advances in Intrusion Detection*, (pp. 37–53). Springer.

Xu, Z., Wu, Z., Li, Z., Jee, K., Rhee, J., Xiao, X., Xu, F., Wang, H., & Jiang, G. (2016). High fidelity data reduction for big data security dependency analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, (pp. 504–516). ACM.

Yen, T.-F., Oprea, A., Onarlioglu, K., Leetham, T., Robertson, W., Juels, A., & Kirda, E. (2013). Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *Proceedings of the 29th Annual Computer Security Applications Conference*, (pp. 199–208). ACM.

Yin, H., Song, D., Egele, M., Kruegel, C., & Kirda, E. (2007). Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, (pp. 116–127). ACM.

Yu, J., Reddy, Y. R., Selliah, S., Kankanahalli, S., Reddy, S., & Bharadwaj, V. (2004). Trinetr: an intrusion detection alert management systems. In *13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, (pp. 235–240). IEEE.

Yu, X., Joshi, P., Xu, J., Jin, G., Zhang, H., & Jiang, G. (2016). CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16*, (pp. 489–502)., Atlanta, Georgia, USA. ACM Press.

Yuan, D., Park, S., & Zhou, Y. (2012). Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, (pp. 102–112). IEEE Press.

Yuan, D., Zheng, J., Park, S., Zhou, Y., & Savage, S. (2012). Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS), 30*(1), 4.

Yurcik, W., Abad, C., Hasan, R., Saleem, M., & Sridharan, S. (2006). Uclog+: a security data management system for correlating alerts, incidents, and raw data from remote logs. *arXiv preprint cs/0607111*.

Yusof, R., Selamat, S. R., & Sahib, S. (2008). Intrusion alert correlation technique analysis for heterogeneous log. *International Journal of Computer Science and Network Security, 8*(9), 132–138.

Zand, A., Houmansadr, A., Vigna, G., Kemmerer, R., & Kruegel, C. (2015). Know your achilles' heel: Automatic detection of network critical services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, (pp. 41–50). ACM.

Zaraska, K. (2003). Prelude ids: current state and development perspectives.

Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., & Zhou, Y. (2017). Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (pp. 565–581)., New York, NY, USA. ACM.

Zhu, B. & Ghorbani, A. A. (2006). Alert correlation for extracting attack strategies. *IJ Network Security, 3*(3), 244–258.

Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., & Zhang, D. (2015). Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, (pp. 415–425). IEEE Press.

Zimmermann, J., Mé, L., & Bidan, C. (2003). An improved reference flow control model for policy-based intrusion detection. In *European Symposium on Research in Computer Security*, (pp. 291–308). Springer.

**Titre :** **Définition de la Relation de Dépendance Causale entre Événements Hétérogènes pour la Détection et l'Explication de Scénarios d'Attaque Multi-Étapes**

**Mot clés :** corrélation d'alertes et d'événements, découverte de scénarios d'attaque multi-étapes, modèle formel, dépendances causales

**Resumé :** Partant du constat qu'un attaquant motivé finit par réussir à s'infiltrer dans un réseau malgré les moyens de prévention déployés, la mise en place d'une supervision de sécurité du système est indispensable. L'objectif de cette thèse est de permettre la découverte de scénarios d'attaque multi-étapes à travers l'analyse d'événements de sécurité. Pour atteindre cet objectif, les précédentes approches, issues du domaine de la corrélation d'alertes, visent à construire des liens entre les événements et entre les étapes d'une attaque. En pratique, ces liens sont difficiles à définir et découvrir, notamment lorsque l'on considère l'analyse d'événements hétérogènes (c.-à-d. produits par différents types de systèmes de supervision déployés dans différentes couches d'abstraction). De plus, la littérature ne propose pas de définition formelle de ce lien. Selon nous, ce lien correspond à une relation de dépendance causale. Inspirés de deux modèles de causalité précédemment définis dans les domaines des systèmes distribués (modèle de Lamport) et de la sécurité (modèle de d'Ausbourg), nous avons donc proposé une définition formelle de cette relation dénommée *event causal dependency*. Cette relation permet la découverte de tous les événements pouvant être considérés comme les causes, ou les effets, d'un événement d'intérêt telle qu'une alerte par exemple. À notre connaissance, nos travaux sont les premiers à proposer une définition formelle de la relation de dépendance causale entre événements hétérogènes. Actuellement, les méthodes proposées dans la littérature ne permettent de construire qu'une approximation de notre modèle. Notre implémentation a la particularité de se baser uniquement sur l'analyse d'événements issus de *COTS* tels que Zeek NIDS et auditd. Cette dernière permet d'obtenir une bonne approximation de notre modèle.

**Title :** **Heterogeneous Event Causal Dependency Definition for the Detection and Explanation of Multi-Step Attacks**

**Keywords :** alert and event correlation, multi-step attack discovery, formal model, causal dependencies

**Abstract :** Knowing that a persistent attacker will eventually succeed in gaining a foothold inside the targeted network despite prevention mechanisms, it is mandatory to perform security monitoring on the system. The purpose of this thesis is to enable the discovery of multi-step attacks through logged events analysis. To that end, previous alert correlation work has aimed at building connections among events and between attack steps. In practice, this type of link is not trivial to define and discover, especially when considering heterogeneous events (i.e., events emanating from monitoring systems deployed in different abstraction layers of the monitored system), and the literature lacks a formal definition of these connections. We argue that the connections among heterogeneous events correspond to causal dependency relationships among events. Inspired from two causality models from the distributed system and the security research areas, i.e., Lamport's and d'Ausbourg's models, we have thereby proposed a formal definition of this relationship called *event causal dependency*. The relationship enables the discovery of all events, which can be considered as the cause or the effect of an event of interest (e.g., an alert produced by an attacker action). To the best of our knowledge, our work is the first one to propose a formal definition of the causal dependency relationship among heterogeneous events. We present how existing work permits the computation of parts of the overall model, and detail our implementation, which exclusively leverages existing monitoring facilities (e.g., auditd, or Zeek NIDS) to produce events. We show that our implementation already yields a good approximation of our model.