Titre: Title:	Langage dédié et analyse automatisée pour la détection de patrons au sein de traces d'exécution
Auteur: Author:	Kadjo Gwandy Kouamé
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Kouamé, K. G. (2015). Langage dédié et analyse automatisée pour la détection de patrons au sein de traces d'exécution [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1900/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1900/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

LANGAGE DÉDIÉ ET ANALYSE AUTOMATISÉE POUR LA DÉTECTION DE PATRONS AU SEIN DE TRACES D'EXÉCUTION

KADJO GWANDY KOUAMÉ

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

AOÛT 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

LANGAGE DÉDIÉ ET ANALYSE AUTOMATISÉE POUR LA DÉTECTION DE PATRONS AU SEIN DE TRACES D'EXÉCUTION

présenté par : KOUAMÉ Kadjo Gwandy

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme BOUCHENEB Hanifa, Doctorat, présidente

M. DAGENAIS Michel, Ph. D., membre et directeur de recherche

M. KHOMH Foutse, Ph. D., membre

DÉDICACE

À mes parents

dont le soutien inébranlable

a été pour moi une inspiration et

une source de motivation

REMERCIEMENTS

J'aimerais remercier mes proches et amis qui m'ont toujours soutenu. Plus particulièrement, je remercie mes très chers parents, Affiba et Kadjo, pour leur précieux soutien et amour. Je remercie mes amis, qui constituent une seconde famille pour moi, qui m'écoutent et me conseillent tout le temps.

Je souhaiterais remercier aussi mon directeur de recherche, le professeur Michel Dagenais, de m'avoir honoré en m'acceptant au sein de l'équipe DORSAL et de m'avoir confié un projet. Son encadrement fut indispensable pour la bonne conduite de ce projet.

Je souhaiterais également remercier Naser Ezzati-Jivan pour sa disponibilité et son encadrement qui ont été particulièrement précieux. Un merci particulier à tous mes autres collègues du laboratoire DORSAL pour leur accueil.

Je remercie également Matthew Khouzam dont les conseils et le soutien m'ont mené là où je suis.

Enfin, merci à Ericsson, dont le soutien à permis la réalisation de ce projet. Merci encore à toute l'équipe de traçage pour leur disponibilité et leurs conseils.

RÉSUMÉ

La complexité des systèmes informatiques distribués et à plusieurs unités de calcul a introduit de nouvelles classes de problèmes. Ces problèmes sont difficiles à reproduire et leur complexité accrue a suggéré l'introduction de nouvelles méthodes de compréhension des systèmes.

L'analyse dynamique, à l'aide de traces d'exécution, permet de comprendre les systèmes à partir de leurs données d'exécution. Les traces d'exécution enregistrent, sous forme d'événements, les informations précises et détaillées de l'exécution du système instrumenté.

Pour des systèmes comme le noyau d'exploitation de Linux, le traçage fournit des événements de bas niveau (appels systèmes, fautes de pages). De plus, en quelques secondes, le fichier de trace peut enregistrer des millions d'événements. Des visionneuses de trace, telle que Trace Compass, ont été développées dans le but de fournir des analyses de la trace sous différents angles de vue tels que l'allocation des ressources ou l'usage des unités de calcul, et à un haut niveau d'abstraction.

Cependant, au-delà des analyses déjà fournies par les visionneuses, les utilisateurs souhaiteraient pouvoir créer des analyses personnalisées qui représenteraient mieux leurs besoins. Par exemple, un utilisateur pourrait tenter de vérifier si le système a subi une attaque particulière. Il faudrait dans ce cas précis pouvoir appliquer à la trace une analyse spécialisée qui permettrait de vérifier la présence d'une séquence d'événements ou d'informations qui décrit l'attaque recherchée. Il existe donc un besoin quant à la nécessité d'identifier des formes particulières ou de retrouver des séquences d'intérêts au sein de la trace.

Ce travail propose l'introduction d'un langage déclaratif utilisant les automates finis pour la description de patrons d'intérêts. Les patrons décrits sont ensuite passés à un analyseur élaboré afin de vérifier et repérer leurs présences au sein de traces d'exécution. L'utilisation de machines à états pour la description des patrons permet de décrire des patrons complexes. Ainsi, la mise en place d'un mécanisme de suivi de l'exécution des patrons a été réalisée.

Le langage déclaratif proposé est conçu de façon déclarative en XML. Il permet de représenter avec succès tous les types de patrons rencontrés dans la littérature (patrons de détection d'attaques, patrons de test de programmes, patrons d'évaluation de performance, patrons d'agrégations des événements...). La spécification du langage proposé permet de créer des

événements synthétiques qui peuvent être traités par l'analyseur au fur et à mesure qu'ils sont créés.

La solution proposée dans ce mémoire devrait être capable de traiter les traces de grandes tailles (500MB et plus); la performance en terme de temps d'extraction des données est donc importante. Nous nous assurons qu'elle est au moins aussi bonne que celle des travaux antérieurs du même genre et que la déviation par rapport à la méthode d'extraction standard de Trace Compass reste acceptable. La solution proposée écrit directement les données sur le disque. Elle n'accumule donc pas d'informations en mémoire. L'analyse en terme d'espace en mémoire est donc négligeable.

De plus, nous démontrons l'utilité de l'approche proposée à l'aide d'exemples concrets de cas d'utilisation. Une tentative de découverte de la source d'un défaut de requête Web est présentée ainsi qu'un exemple de détection d'attaque dans le système.

Enfin, nous proposons à la fin de notre étude une liste de suggestions pour des améliorations possibles à la solution en termes de description des patrons et de réduction du temps de l'analyse.

ABSTRACT

The complexity of distributed and multi-core systems introduced new classes of problems. These problems could be difficult to reproduce and their increasing complexity has suggested the introduction of new methods of systems comprehension.

Dynamic analysis, through execution traces, allows to understand the system behavior from its execution data. The execution traces record, in the form of events, accurate and detailed information about the execution of an instrumented system.

For systems like the Linux kernel, tracing provides low level events such as system calls and page faults. Moreover, in a few seconds, the trace file can record millions of events. Visualizers, like Trace Compass, were developed to provide analysis of the trace from different points of view such as ressources allocation and CPU usage, and with a high-level of abstraction.

However, beyond the analyses that have been already provided by the visualizers, users would like to be able to create custom analyses that better represent their needs. For example, a user could attempt to verify if the system is under a particular attack. It should then be possible to apply to the trace a specialized analysis that would verify the presence of a sequence of events or information that describes the intended attack. Thus, there is a need to be able to identify patterns or to find predefined sequences of events within the trace.

This work proposes the introduction of a declarative automata-based pattern description language. The described patterns are then passed to an analyzer designed to efficiently verify and detect their presence within the execution trace. The use of state machines allows to describe complex patterns. Thus, a mechanism to follow the execution of the patterns has been implemented.

The proposed language is designed declaratively in XML. It has successfully represented all the types of pattern found in the literature (security attack patterns, testing patterns, system performance patterns, events aggregation patterns, ...). The language specification allows to create synthetic events that can be processed by the analyzer as they are created.

This proposal should be able to process large trace files (1GB or more). Thus, performance in terms of time of data extraction is important. We ensure that this solution is at least as good as previous ones of the same kind and that the deviation from the standard extraction method of

Trace Compass remains acceptable. The proposed solution writes the data directly to disk. It therefore does not accumulate information in memory. The analysis in terms of memory space is negligible.

In addition, we try to demonstrate the usefulness of our approach with some application test cases. An example of an attempt to find the root cause of a web request defect and an example of attack detection in a system are presented.

Finally, we propose at the end of this study a list of suggestions for possible improvements of the solution in terms of the description of patterns and reduction of the time of the analysis.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ASAX	Advanced security audit-trail analysis on uniX
BPF	Berkeley packet filter
CPU	Central processing unit
DIF	Differential item functioning
FSM	Finite state machine
GB	Gigabyte
GiB	Gibibyte
IDS	Intrusion detection system
LTTNG	Linux trace toolkit next generation
LTTV	Linux trace toolkit viewer
MB	Megabyte
MiB	Mebibyte
RUSSEL	RUle-baSed sequence evaluation language
STAT	State transition analysis technique
STATL	State transition analysis technique language
TMF	Tracing monitoring framework
UML	Unified modeling language
XML	Extensible markup language

CHAPITRE 1 INTRODUCTION

La maintenance des systèmes informatiques est une activité délicate de nos jours principalement à cause de l'augmentation des capacités de calculs des machines et aux possibilités qu'elles offrent. Il est de plus en plus difficile de trouver les défauts et d'améliorer la performance des systèmes. Par exemple, pour comprendre un comportement inatendu survenu dans un système distribué, il faut les informations d'exécution de chacun des composants du système alors que le système ne peut être arrêté. Plus encore, ce comportement peut être difficile à reproduire et se produit rarement dans le système.

Les traces d'exécution fournissent des informations détaillées sur l'exécution du système qui permettent par exemple aux développeurs de valider la justesse du programme ou d'expliquer les comportements inattendus.

Par contre, la quantité de données présentes dans la trace est relativement importante. L'exploration de la trace pour l'extraction d'informations significatives reste une tâche difficile qui soulève de nombreux enjeux. Plus particulièrement, l'ensemble de la recherche exposé dans ce document porte sur la vérification de la présence de patron dans la trace.

Nous présentons dans ce mémoire une approche flexible pour identifier des séquences de données enregistrées au sein de la trace. Plus précisément, nous laissons le soin aux développeurs de décrire eux-mêmes les patrons des scénarios qu'ils souhaitent retrouver dans la trace.

Les travaux présentés dans ce document introduisent un langage de description de patrons conçu de façon déclarative en XML. Les patrons décrits vont permettre de créer des analyses automatisées qui vont parcourir la trace à la recherche de leur présence.

1.1 Définitions et concepts de base

1.1.1 Événement

Un événement est un enregistrement ponctuel qui décrit une action qui se produit dans le système. Il est représenté par une estampille de temps (*t*) qui précise à quel moment l'action s'est produite et possède un type qui donne plus de précision sur sa nature: appel système, faute de page,…. L'événement peut contenir un ensemble d'informations qui vont permettre d'avoir plus

de détails sur l'état du système. Sa structure varie en fonction de sa nature et du traceur utilisé pour l'enregistrer.

L'étude présentée dans ce document ne concerne que les traces dont les informations sont enregistrées sous forme d'événements.

1.1.2 État

Un état, en comparaison avec un événement, possède un temps de début et un temps de fin. Il est donc représenté par un intervalle de temps et une valeur. Un système est représenté par un ensemble d'états qui illustrent son comportement dans le temps

1.1.3 Attribut

Un attribut est la plus petite fraction du modèle dont la valeur représente un état. À tout moment dans la trace, chaque attribut ne peut avoir qu'un seul état qui peut évoluer dans le temps.

1.1.4 Patron

Un patron est la représentation d'une séquence d'informations que l'on souhaite repérer au sein de la trace.

1.2 Problématique

Avec l'avènement des systèmes informatiques distribués, à plusieurs processeurs et à plusieurs fils d'exécution, il est de plus en plus difficile de maintenir ces systèmes et d'en assurer le bon fonctionnement. Dans les systèmes distribués par exemple, le système est composé de plusieurs composants qui communiquent entre eux. Les problèmes qui surviennent dans ces systèmes avancés sont complexes. Il est dans la plupart des cas nécessaire de disposer d'informations provenant de plusieurs composantes du système pour pouvoir comprendre l'origine du problème et déterminer des pistes de solution. Les techniques de débogage conventionnelles s'avèrent inefficaces pour ces systèmes puisqu'on ne peut pas en suspendre l'exécution, le temps d'en examiner l'état. Les développeurs doivent donc se tourner vers d'autres outils offrant une couverture plus vaste et des données plus précises.

Le traçage est une technique qui enregistre sous forme de fichier journal, les informations sur l'exécution d'un programme. C'est une technique de plus en plus utilisée pour attaquer le genre de problème décrit plus haut.

Un fichier de trace contient une suite d'événements qui décrivent l'évolution de l'état du programme dans le temps, pendant son exécution. Il est possible de tracer plusieurs types de systèmes allant du noyau des systèmes d'exploitation aux applications en espace utilisateur. Les fichiers de trace contiennent de l'information précise sur l'exécution d'un programme. Pour des programmes tels que les noyaux de systèmes d'exploitation comme Linux, la quantité d'informations recueillies sur une courte période de temps est énorme. En quelques secondes, le fichier de trace peut rapidement atteindre des centaines de mégaoctets et contenir des milliers d'événements. L'analyse d'une trace permet de mieux comprendre le comportement d'un programme lors de son exécution. Cependant, l'utilisateur peut s'intéresser à un ensemble d'analyses en particulier, par exemple la détection d'une intrusion ou la génération de données statistiques , et souhaiterait possiblement appliquer des filtres personnalisés à la trace sans avoir à modifier le code de l'analyseur de trace. De plus, ces analyses reposent sur seulement une faible quantité d'informations contenues dans la trace et la séquence d'intérêt peut être éparpillée dans la trace. Considérer toutes les informations contenues dans la trace n'est donc pas requis; d'où la nécessité d'un outil adapté pour filtrer la trace et extraire des séquences d'intérêts.

Question de recherche

À partir de la problématique, nous pouvons faire ressortir la question de recherche suivante :

> Peut-on définir simplement, trouver et montrer efficacement des patrons sophistiqués qui correspondent à des situations réelles d'intérêt dans les traces d'exécution?

Objectif de recherche

De la question de recherche énoncée dans la section précédente découle les objectifs spécifiques de recherche suivants :

1. Élaborer un langage de description de patron générique pour représenter les patrons des filtres que l'on souhaiterait appliquer sur la trace.
2. Développer un analyseur de traces capable de filtrer la trace à partir des patrons décrits avec notre langage.

3. Élaborer un mécanisme de suivi de l’évolution des patrons dans la trace.
4. Représenter de façon efficace les patrons repérés au sein de la trace.
5. Valider que la solution ne dégrade pas les critères de performance des analyses et que cette solution soit au moins aussi performante que les travaux antérieurs qui lui sont semblables.

Dans le chapitre suivant, nous présentons une revue de littérature qui fait le point sur l’état de l’art dans les domaines du traçage, de la représentation de l'état du système, de l'abstraction de la trace et des langages de description.

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE

Ce chapitre est une revue critique de l'état de l'art des techniques de traçage, de représentation de l'état du système, d'abstraction de la trace et de langage de description de patrons.

2.1 Traçage

L'analyse dynamique de programmes est une technique de plus en plus populaire afin d'assurer le maintien des systèmes informatiques. C'est une technique qui fournit une image précise du logiciel, car elle expose le comportement réel de celui-ci [6]. La compréhension du système par l'analyse dynamique est utilisée dans la littérature afin de réaliser plusieurs objectifs comme la découverte d'erreurs, la compréhension de certaines fonctionnalités du système, la résolution de problèmes de performance et la surveillance de systèmes durant leur exécution pour la détection d'anomalies [6, 21, 15, 24, 20].

Le traçage est un moyen d'assurer l'analyse dynamique d'un logiciel. Cette technique fournit des informations explicites sur le système en exécution. Elle permet d'observer, à faible coût, le comportement d'un programme durant son exécution. Le système tracé possède des points de trace qui se comportent comme des points d'arrêt, au niveau desquels le traceur enregistre des données avant de laisser le programme reprendre son cours d'exécution.

Les systèmes informatiques peuvent être instrumentés de deux façons : l'instrumentation statique et l'instrumentation dynamique [16].

Instrumentation statique

Les points de traces statiques sont insérés à l'intérieur du code source de l'application avant la compilation. Ils peuvent être activés ou désactivés. Des données sont recueillies à chaque à fois qu'un point de trace est validé. Il n'est pas nécessaire pour l'utilisateur de posséder une connaissance approfondie du logiciel avec cette technique de traçage. Les points de trace sont disposés à des endroits stratégiques dans le code, habituellement au moment même de la conception du programme. Par contre, il y a un coût additionnel pour chaque point de trace, même s'il est désactivé.

Instrumentation dynamique

Avec l'instrumentation dynamique, les points de trace dynamiques sont insérés au temps d'exécution du programme. L'utilisateur n'ajoute des points de trace que lorsque cela est nécessaire. Par conséquent, l'impact sur le temps d'exécution dépend du nombre de points de trace qui sont activés. Par contre, l'utilisation de ce type de traçage nécessite une compréhension avancée du programme puisque l'utilisateur choisit à quel endroit recueillir les données. Le coût additionnel de ce type de traçage, comparé à celui de l'instrumentation statique, est généralement légèrement moins important (puisque nul) que celui d'un point de trace statique désactivé, mais sensiblement plus élevé que celui d'un point de trace statique activé.

Le traçage a bénéficié d'un important intérêt dans la communauté des développeurs au cours des dernières années. Nous effectuons, dans cette section, une description de quelques-uns des traceurs sur Linux les plus populaires dans le domaine.

2.1.1 DTrace

DTrace est un traceur dynamique incorporé originellement dans Solaris en 2005. La première version de DTrace pour Linux est apparue en 2011. C'est un traceur réputé pour sa performance. En effet, le coût additionnel des points de trace lorsqu'ils sont désactivés est quasiment inexistant. Il utilise des scripts décrits avec le langage de programmation '*D*' afin de définir les sondes et les actions aux points de sonde. Il se sert d'un ensemble de fournisseurs qui lui permettent d'observer des détails intéressants sur le système tel que les événements sur le CPU, la suite d'appel de fonction, le temps écoulé…

2.1.2 Ftrace

Ftrace ou function tracer est un traceur inclus dans les distributions de Linux à partir de la version 2.6.27 du noyau. C'est un traceur de système en temps réel. Il enregistre les événements du noyau tels que les appels système, les gestionnaires d'interruption ou les fonctions d'ordonnancement [36]. Il utilise le système de fichier DebugFS comme interface afin de sélectionner les événements à enregistrer dans le noyau. Il provient originellement du correctif -rt [33] de traceur de latence. C'est un puissant traceur pour traquer les problèmes de latence et découvrir leurs origines [32]. Il permet de déterminer si les problèmes de latence proviennent du

noyau de Linux ou d'une application tiers. Il offre la possibilité de suivre la latence des appels systèmes, des changements de contexte et des interruptions.

2.1.3 LTTng

Linux Trace Toolkit Next Generation ou LTTng est un traceur libre pour Linux [23]. C'est un traceur orienté sur la minimisation du coût additionnel engendré par la collecte de données dans le noyau de Linux. Pour ce faire, il utilise par exemple des opérations atomiques plutôt que des verrouillages pour la mise à jour des variables de contrôle afin d'éviter la latence causée par ces opérations au niveau de la synchronisation. Ce traceur utilise des tampons circulaires sur lesquels il écrit les événements enregistrés. Un processus externe est chargé, par la suite, de vider les tampons et d'écrire les données sur le disque dur; ce qui évite les temps d'attentes. Le traceur est contrôlé par une session daemon qui reçoit des commandes à partir de l'interface de commande *lttng*.

Les événements sont enregistrés sous le format *Common Trace Format (CTF)*. Il s'agit d'un format très compact qui sauvegarde les événements sous forme binaire avant de les écrire sur le disque. Il est possible, par la suite, d'utiliser le fichier de métadonnées pour reconstruire les données et procéder à des analyses.

LTTng offre également, avec la version LTTng-UST, la possibilité de tracer des applications en ajoutant des points de trace de façon adéquate, directement dans le code source de l'application.

2.1.4 Perf

Perf est inclus dans le noyau de Linux depuis 2009. Il utilise la macro TRACE_EVENT() pour recueillir les informations au niveau des points de trace. Il utilise également des tampons circulaires. Les informations enregistrées permettent d'obtenir des statistiques sur certains événements et d'entretenir des compteurs de performance [11]. On peut obtenir des statistiques sur les changements de contexte, les fautes de page, les fautes de caches et les cycles des processeurs. Lorsque la valeur d'un compteur de performance dépasse un certain seuil, une interruption est générée afin de sauvegarder la valeur du compteur. Cette façon de procéder réduit l'impact du traceur sur le système.

2.1.5 SystemTap

SystemTap est un traceur assez semblable à DTrace, adapté aux administrateurs système. Il leur permet de surveiller le comportement du système. C'est un traceur à code source libre issu d'une contribution originelle de Red Hat en 2005. Il positionne des points de trace dynamique en s'appuyant sur Kprobes au niveau de Linux et Uprobes dans les programmes en espaces utilisateurs. C'est un traceur axé beaucoup plus sur la flexibilité que sur la performance. Il utilise un mécanisme coûteux de verrouillage pour la synchronisation et peut facilement emmagasiner une énorme quantité de données.

2.2 Représentation de l'état du système

2.2.1 L'état du système

Les visualiseurs de trace noyau tels que LTTV et Trace Compass, développés par le laboratoire DORSAL et ses partenaires, ont besoin de pouvoir recréer l'état complet du système à n'importe quel point dans la trace. Pour ce faire, ces derniers utilisent une approche par états. Cette approche offre la possibilité d'exploiter un diagramme de Gantt pour représenter les informations d'ordonnancement des processeurs et des processus.

Wininger et al. [38] résument le fonctionnement du gestionnaire d'états. C'est pendant l'indexation de la trace que la machine à états représentant le système est créée, en parcourant la trace du début jusqu'à la fin sans jamais revenir en arrière. De ce fait, le gestionnaire d'état est initialisé avec des attributs vides jusqu'à ce que l'on rencontre un événement pertinent qui va fournir une première valeur à l'attribut correspondant. La figure 2.1, tirée de [38], est un exemple d'état, créé à partir d'événements qui montre l'évolution du statut d'un fil d'exécution depuis son initialisation. Avant la rencontre du premier événement *sched_switch*, l'attribut n'existe pas et son état est *Null*. L'état du fil d'exécution évolue par la suite dans le temps en fonction des événements qui lui sont liés.

En cas de perte d'un événement, il est possible de rester dans un état incohérent. Par exemple, dans le cas du fil d'exécution de la figure 2.1, si l'événement *sys_read* est perdu, le *Thread 1* restera à l'état *User Space* tant qu'un autre événement déterminant ne sera pas rencontré. La figure 2.2, aussi tiré de [38], illustre l'état du fil d'exécution quand l'événement *sys_read* est

perdu. L'information sur le nombre d'événements perdus est disponible dans la trace. Toutefois, il n'est pour l'instant pas facilement possible d'avertir l'utilisateur de la possibilité d'états incohérents, en raison de la perte d'événements dans la trace.

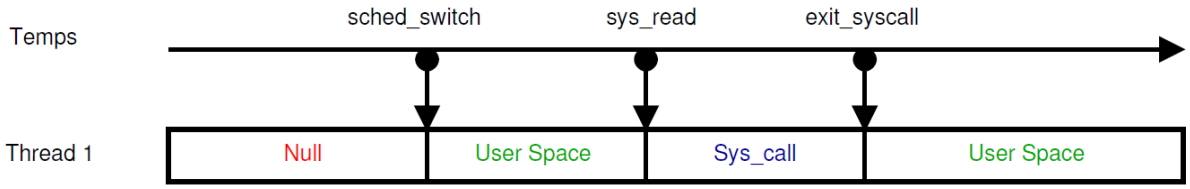


Figure 2.1: Évolution de l'état d'un fil d'exécution dans le système

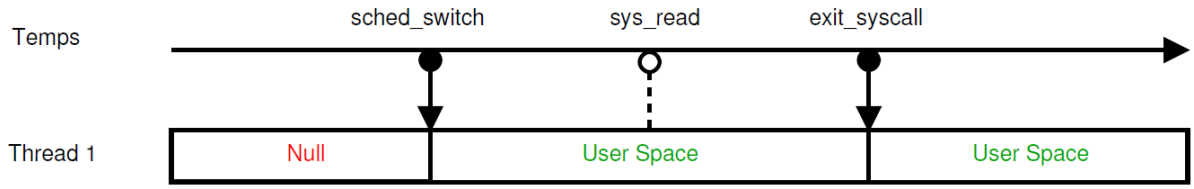


Figure 2.2: Évolution de l'état d'un fil d'exécution avec perte d'un événement

Les traceurs ETW et LTTng propagent des événements au démarrage et à la fin de la trace afin d'initialiser l'état des processus dans le gestionnaire d'état. Ils ont aussi la possibilité d'indiquer le nombre d'événements perdus afin d'assurer la fiabilité des données de la trace.

2.2.2 L'arbre à historique

Pour des systèmes de grande taille, la construction de la machine à états pourrait avoir un coût très élevé. La méthode classique des clichés pour naviguer dans l'historique s'avère être inefficace. Cette méthode consiste à disposer d'une série de clichés à intervalle régulier. Chaque cliché contient l'état complet du système pour l'intervalle qui le concerne. En effet, lorsqu'une requête est effectuée à un instant *t*, le système recharge le cliché le plus proche puis relit les événements de la trace depuis le point du cliché jusqu'à l'instant recherché. Avec un système de grande taille, il y a beaucoup plus d'événements à relire et le cliché contient beaucoup plus d'informations, dont une grande partie se répète d'un cliché à l'autre.

Montplaisir et al. [26] propose de stocker l'état du système dans une nouvelle structure de données dédiée appelé *arbre à historique*. Cette structure de données se base sur l'hypothèse que les intervalles d'état sont triés par ordre croissant de temps de fin lors de leur arrivée. Les recherches selon le temps sont ainsi garanties et il n'y a pas besoin de rééquilibrer l'arbre pendant sa création. L'arbre à historique a donc un avantage certain comparé au *R-tree* ou aux bases de données indexées qui ont constamment besoin d'être rebalancés pour garder un état cohérent.

L'écriture de l'arbre se fait directement sur le disque et de façon continue; ce qui permet de supporter des traces de très grande taille. Le temps de réponse des requêtes est logarithmique. De plus, aucune information redondante n'est stockée.

La construction de l'arbre se fait de manière incrémentale dans le temps. Ainsi, il est possible d'arrêter la construction de l'arbre et de la reprendre à n'importe quel moment.

L'arbre à historique est une structure de donnée dédiée pour la sauvegarde d'intervalle sur le disque. Elle utilise des nœuds de tailles constantes pour regrouper les nœuds. Ce sont ces nœuds qui sont arrangés de façon à former un arbre.

Il existe néanmoins la possibilité de sauvegarder l'information d'état directement dans la trace. Chan et al. [5] propose l'utilisation de traces qui contiennent à la fois des événements ponctuels et des intervalles d'états. Toutefois, pour un visualiseur de trace tel que Trace Compass, il est nécessaire de pouvoir supporter tout format de trace. Cependant, l'approche de stockage proposé par Chan et al. apporte très peu de flexibilité.

2.2.3 Analyse flexible

Le visualiseur Trace Compass souhaite pouvoir supporter tous les formats de trace. C'est dans cette tentative de généralisation du gestionnaire d'états que Wininger et al. [38] propose un outil, basé sur les machines à états, pour concevoir des analyses flexibles. Les auteurs proposent un langage déclaratif dédié, utilisant le XML, pour accéder aux fonctionnalités du gestionnaire d'états de Trace Compass.

Il est possible de décrire, avec le langage proposé, un modèle qui fonctionne comme un gestionnaire d'événements. Le modèle définit comment évolue l'état d'un attribut selon les événements qui surviennent dans la trace.

La description du modèle qui compose l'analyse est réalisée en dehors de l'environnement du visualiseur et est indépendante de son implémentation. L'outil offre ainsi la possibilité de générer des analyses plus riches et variées. De plus, le modèle décrit est également indépendant du système de traçage. Chaque analyse décrite avec le langage pourrait être effectuée avec des systèmes de traçages différents.

En comparaison avec les analyses conventionnelles de Trace Compass, ce nouveau type d'analyse ne produit aucun coût additionnel en terme de temps de construction de la machine à états. De plus, l'outil dispose d'une forte extensibilité liée à l'utilisation du XML pour la description des modèles.

2.3 Abstraction de la trace

Les fichiers de trace fournissent des données cruciales collectées pendant l'exécution du système. L'analyse de ces données d'exécution permet de comprendre le comportement du système, de déceler des problèmes et trouver leurs origines, de surveiller le système. Cependant, la quantité des données enregistrées rend difficile l'atteinte de ces objectifs. L'abstraction de la trace constitue un moyen de remédier à ce problème et réduit la complexité de la trace et de son analyse. Dans [13], les auteurs discutent des différentes techniques d'abstraction à plusieurs niveaux. Ils proposent une taxonomie des techniques d'abstraction des traces basée sur leurs usages possibles dans un outil de visualisation de trace à plusieurs niveaux. Dans cette section, nous résumons les différentes catégories d'abstraction sur lesquelles les auteurs s'appuient pour classer les techniques d'abstraction de traces existantes.

2.3.1 Abstraction basée sur le contenu

Il s'agit d'utiliser les informations contenues à l'intérieur des événements pour réaliser l'abstraction. Ce type d'abstraction est utilisé pour atteindre différents objectifs tels que la réduction de la taille de la trace, la généralisation de la représentation des données, et l'agrégation de la trace.

2.3.1.1 Réduction de la taille des données

La réduction de la taille de la trace a été au centre de bon nombre de travaux dans la littérature. Plusieurs techniques ont été élaborées : traçage sélectif, échantillonnage, filtrage, compression.

- Le traçage sélectif consiste à tracer certains modules et processus en particulier et non le système entier lui-même.

- Il est possible d'échantillonner la trace en analysant seulement une variété précise d'événements au lieu de tous les événements dans la trace.

- Le filtrage de la trace permet d'éliminer tous les événements redondants ou inutiles de la trace et de mettre en exergue ceux qui ont de l'importance. Le filtrage est réalisé en se basant sur des critères variés tels que : l'estampille de temps, le type de l'événement, les arguments de l'événement, le nom d'un processus. Le filtrage au niveau noyau [15] consiste à retirer, des données originales, celles sans intérêts; c'est à dire les événements de gestion de mémoire, et le bruit (comme les fautes de pages).

- Au lieu de sauvegarder les événements sous leur forme de base, il est possible de les sauvegarder sous une forme plus compacte, on réalise ainsi une compression de la trace. On obtient une forme plus compacte des événements en découvrant des similitudes entre les événements et en effaçant les redondances entre elles.

2.3.1.2 Généralisation de la trace

La généralisation permet d'éviter d'être dépendant d'une version particulière du système ou du traceur. Il s'agit du processus par lequel on extrait des concepts communs à un groupe d'événements afin de les combiner et créer un événement généralisé. Par exemple, les appels systèmes *read, readv, pread64,* dans Linux, pourraient être utilisés pour lire un fichier. Cependant, à un niveau d'abstraction élevé, ils peuvent être assimilés à une opération singulière d'ouverture de fichier [13].

2.3.1.3 Agrégation de la trace

L'agrégation de la trace consiste à associer un ensemble d'événements, participants à la réalisation d'une opération, et à former un événement plus large en utilisant l'appariement de patron, l'extraction de patrons ou encore la reconnaissance de patron. [13]

Dans un système à plusieurs niveaux d'abstraction, les événements abstraits de haut niveau représentent un comportement plus général alors que ceux de bas niveau révèlent plus d'informations détaillées.

Appariement de patron

Fadel et al. [15] profitent du fait que les événements, dans le noyau du système d'exploitation, possèdent une entrée et une sortie pour les grouper et faire des agrégations d'événements en utilisant des techniques d'appariement de patron. Waly et al. [37] se servent des techniques d'agrégation et de groupement afin de détecter des fautes et anomalies dans le noyau du système d'exploitation. Cependant, leurs travaux se concentrent plus sur la mise en place d'un langage pour la description des modèles d'agrégation. Ces deux travaux majeurs dans ce domaine ne possèdent malheureusement pas l'évolutivité suffisante pour fonctionner adéquatement avec des fichiers de trace de grande taille; d'où la nécessité de développer de meilleurs outils capables de supporter les larges traces et qui proposent des moteurs de gestion de patron dans la trace plus efficaces.

Matni et al. [25] proposent une approche basée sur les machines à états pour détecter les patrons d'exécution à problème au sein d'une trace. Cette approche permet de détecter des intrusions telles que l'attaque SYN flood, de s'assurer du bon fonctionnement du système et de détecter des problèmes de performance. Cependant, l'analyseur n'est pas optimisé et ne considère pas le partage d'information commune entre patrons [13].

L'extraction de patrons

L'extraction de patrons utilise les techniques d'agrégation et de groupement d'événements afin de trouver les patrons qui surviennent fréquemment dans la trace. L'extraction de patrons fréquents est utile pour découvrir des associations intéressantes entre différents éléments. L'extraction de patrons peut aussi servir à trouver des erreurs et problèmes dans le système. LaRosa et al., dans [22], ont développé une plateforme de fouille afin de détecter des problèmes de communications interprocessus dans la trace. Leur solution s'appuie sur la présence de patrons de données fréquents dans la trace. Les auteurs utilisent des fenêtres de pliage et de tranchage afin de grouper les événements en ensemble d'éléments. Cette technique leur a permis de détecter

les patrons de communication inter-processus excessifs qui ont un impact sur la performance du système.

Toutes ces méthodes s'appuient sur la possibilité de définir des patrons pour la trace. Cependant, cette méthode possède des limitations. Des patrons concurrents concernant le même scénario possèdent certaines données semblables à un certain point; d'où l'introduction du concept de partage d'états et d'espace de stockage.

2.3.2 Abstraction basée sur les métriques

La mise en place de certaines métriques permet d'extraire des mesures et des valeurs cumulées à partir des événements de la trace. Ces mesures (charge du CPU, nombre de tentatives d'attaque…) peuvent aider à avoir une vue globale de la trace. Avec les statistiques générées à partir des paramètres du système, il est possible de fouiller à l'intérieur de celui-ci et de déceler des problèmes importants. Il est possible d'obtenir les statistiques du système en agrégeant les événements des traces de noyau de système d'exploitation. Il est possible d'extraire des statistiques telles que :

- L'utilisation du CPU pour chaque processus, la proportion du temps pour laquelle le processus est occupé ou au repos.
- Le nombre d'octets lus ou écrits pour chaque fichier ou opération sur le réseau ainsi que le nombre d'accès au fichier.
- Quel disque est le plus utilisé, quelle est la latence des opérations sur le disque et quelle est la distribution des distances de positionnement.
- Quel est le débit d'entrée sortie (I/O) sur le réseau, quel est le nombre de tentatives de connexion échouées.
- Quelle est la quantité de mémoire utilisée par un processus? Quels numéros (proportion) de page de mémoire sont (le plus souvent) utilisés.

Pour effectuer l'abstraction basée sur les métriques, un patron des événements pour chaque métrique doit être enregistré. Chaque patron contient un ensemble d'événements et une

agrégation sous-jacente qui spécifie comment calculer les valeurs des métriques à partir des événements qui correspondaient au patron [13].

2.3.3 Abstraction au niveau visualisation

La manière avec laquelle les données sont représentées, et même après l'abstraction de la trace basée sur le contenu, n'a pas forcément de sens dans un environnement de visualisation de la trace. Il est parfois nécessaire de procéder à une abstraction visuelle des données pour les regrouper et les afficher une partie à la fois au moyen de vues plus simples et plus pertinentes. Procéder ainsi permet donc de réduire la complexité de la trace et d'améliorer sa lisibilité et sa clarté, à l'opposé de l'abstraction des données qui analyse et manipule le contenu, l'abstraction au niveau visuel se concentre sur la représentation des données. L'utilisation de couleurs et de formes particulières pour la représentation des données se fait de plus en plus dans les outils de visualisation tels que LTTV ou Trace Compass. Il est aussi possible d'abstraire les éléments de trace à l'aide d'annotations telles que les étiquettes afin de représenter des groupes d'événements sous un même nom de fonction, nom d'appel système ou de valeur de retour.

2.3.4 Abstraction au niveau des ressources

Les événements à l'intérieur de la trace incluent des interactions entre les différentes ressources du système. Par exemple, un événement pour l'ouverture d'un fichier contient des informations sur le processus en cours, le fichier ouvert, le CPU qui réalise l'opération ainsi que la valeur de retour de l'opération. Ces ressources fournissent de l'information sur l'état du système au moment de l'opération. L'abstraction des ressources et de leur représentation permet de bien les organiser afin de réduire la complexité de la trace. Plusieurs travaux dans la littérature ont porté sur l'abstraction des ressources. Par exemple, Montplaisir et al. [26] exploitent une représentation sous forme d'arbre pour organiser les ressources extraites des événements des traces. Schnorr et al. [34] regroupent les fils d'exécution en se basant sur les processus auxquels ils sont associés ainsi que les noms des machines...

2.4 Langages de description de patrons

La mise en place de langage de description est une pratique courante pour l'analyse de trace et la détection de patrons d'exécution au sein de la trace. Les langages sont variés et possèdent des

syntaxes et paramètres différents. Dans ce chapitre, nous présentons d'abord les attributs principaux dont devrait disposer un langage de description capable de définir les patrons d'exécution ou scénarios auxquels nous souhaitons faire face. Nous décrirons, ensuite, certains langages de description existants selon leur domaine et leur utilité et fournirons un exemple de script écrit avec le langage.

2.4.1 Critère du langage de description

Selon le type de scénarios que nous souhaitons traiter dans notre recherche, nous avons dressé une liste de critères sur lesquelles se baser afin d'analyser l'efficacité d'un langage de description de scénario. Dans cette section, nous énumérons chacun de ces critères.

1. La simplicité : Le langage de description doit être clair, facile à comprendre de façon à ne laisser aucune ambiguïté sur le sens. Décrire un scénario à partir d'un langage simple doit être une tâche à réaliser sans trop d'effort. Pour atteindre ces objectifs, le langage descriptif doit posséder une liste restreinte d'opérateurs et de mots clés utiles pour la description des scénarios. Les mots clés et opérateurs doivent être choisis de façon à faciliter la compréhension du scénario décrit.
2. Expressivité : Le langage devrait être en mesure de décrire tous les types de scénarios connus au moment de notre étude. Il devrait être possible de créer de nouveaux scénarios plus tard.
3. Généricité : La syntaxe du langage ne doit pas dépendre du type de traceur utilisé. Elle doit être générique. Un scénario X décrit avec le langage et appliqué à des traces provenant de traceurs différents ne devrait avoir pour seule différence que quelques informations particulières telles que le nom des événements selon chaque traceur.
4. Généralité : Le langage ne devrait pas être lié à un type d'application en particulier. Il devrait être général et en mesure de décrire à la fois des scénarios pour la détection de faute, la détection d'intrusion, l'agrégation d'événements, la collecte de données statistique.
5. Scénario basé sur des événements multiples : Le langage devrait être en mesure de supporter des scénarios qui sont en fait des séquences d'événements. Il devrait supporter la possibilité de gérer plusieurs types d'événements dans un scénario.

6. Scénario basé sur le temps : Le langage devrait offrir la possibilité de définir des conditions basées sur le temps entre les événements ou entre les états des scénarios.

2.4.2 Langages de description existants

2.4.2.1 Langages déclaratifs

Les langages déclaratifs décrivent qu'est ce que le programme doit faire et non comment il se comporte. Ce type de langage se concentre sur la logique de l'application plutôt que sur le flot d'exécution du programme. Dans cette section, nous décrirons 2 langages déclaratifs : Snort et DejaVu. Les deux langages sont spécifiques à des domaines particuliers.

2.4.2.1.1 SNORT

SNORT [35] est un système populaire de détection d'intrusions. Il s'agit d'un des plus importants logiciels libres dans le domaine de la sécurité. Il est capable de fonctionner en trois modes : le mode renifleur, le mode enregistreur de paquet et le mode de système de détection d'intrusion de réseau.

En mode renifleur ("*sniffer mode*"), le système lit les paquets sur le réseau et les affiche sous forme de flux de données. Dans ce mode, il est possible d'afficher des données de base telles que l'en-tête de paquets ou les données en transit.

En mode enregistreur de paquets ("*packet logger mode*"), SNORT enregistre les paquets sur le disque. Il est possible d'enregistrer les paquets sous format binaire afin de disposer d'un format plus compact des données. L'interface BPF permet de filtrer les informations à afficher lors d'une lecture ultérieure des données.

En mode système de détection d'intrusion de réseau ("*Network Intrusion Detection System mode*" ou *NIDS mode*), SNORT procède à la prévention et la détection d'intrusion ainsi qu'à l'analyse du trafic sur le réseau. Il détecte tous les paquets malicieux sur le réseau qui concordent avec un ensemble de règles préalablement décrites par l'utilisateur. L'utilisateur définit également un ensemble d'action à appliquer pour chacun des paquets détectés. Les sections suivantes, sur SNORT, s'intéresseront principalement à ce mode de fonctionnement.

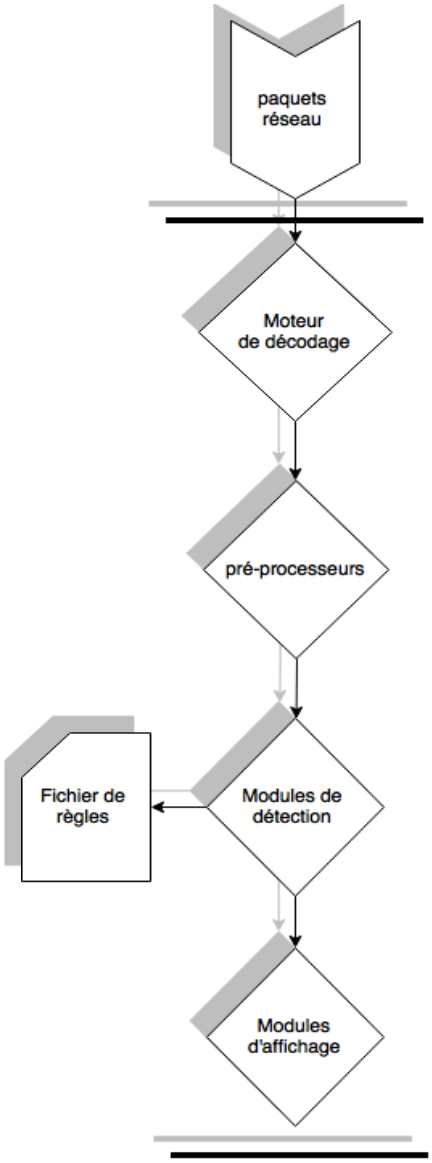


Figure 2.3: Architecture de SNORT

Architecture de SNORT : La figure 2.3 est une représentation de l'architecture de SNORT. L'architecture de SNORT pour la détection d'intrusion est composée principalement de 4 modules.

- Le moteur de décodage de paquet : Il utilise la librairie de *libpcap* pour récupérer les paquets. Les paquets sont décodés à la fois pour des protocoles à bas niveau et aussi pour les protocoles de haut niveau.

- Les préprocesseurs : Ce sont des modules d'extensions. Ils reçoivent les paquets décodés; ils les examinent et les manipulent pour retirer toutes les informations inutiles ou générer des alertes.
- Le moteur de détection : Ce module compare les paquets traités avec un ensemble de règles définies par l'utilisateur. Les règles sont décrites dans un fichier avec lequel le module interagit. SNORT offre la possibilité de rajouter des modules d'extensions pour la détection.
- Les modules d'affichage : C'est l'ensemble des modules nécessaires pour présenter les informations à l'utilisateur ainsi que les résultats de la détection (alertes, en-têtes…)

Syntaxe pour la définition des règles : SNORT utilise un langage déclaratif pour décrire les règles. La figure 2.4 présente 2 exemples de règles simples. Les règles décrites permettent de détecter des informations dans le flux de paquets. Ces règles décrites utilisent les informations disponibles dans les paquets réseau, tels que le protocole, l'adresse de destination, l'adresse source, le masque, le port…

Une règle SNORT est divisée en 2 parties : les en-têtes de règles et les options de règles

- L'en-tête de règles : Cette partie constitue normalement la première partie, à gauche, d'une règle SNORT. Elle est elle-même séparée en 3 parties : les actions de règles, le protocole et les adresses IP et numéros de port. Les actions de règles spécifient quelle action réaliser lorsqu'un paquet est détecté. Les actions possibles sont : "*alert*", "*log*", et "*pass*". TCP, UDP et ICMP sont les 3 protocoles supportés. Le symbole "->" permet de différencier l'adresse source de l'adresse destination.

- Les options de règles : Il s'agit de la partie de la règle à l'intérieur des parenthèses. C'est ici que les critères de la règle sont précisés. SNORT supporte 15 types de critères qui permettent de vérifier différents aspects des paquets réseau et de rechercher des scénarios parmi les paquets.

```
alert udp any any -> 129.210.18.0 / 24 31335 \ (msg: "trinoo port"; logto "DDoS")
alert tcp !10.1.1.0/24 any -> 10.1.1.0/24 any (flags: SF; msg: "SYN-FIN scan")
```

Figure 2.4: Exemple de règle SNORT

Discussion : SNORT propose une excellente façon de décrire des patrons d'attaques à un haut niveau d'abstraction. L'utilisateur n'a pas à se soucier de l'implémentation du système. Cependant, SNORT n'offre pas la possibilité à une règle d'en déclencher une autre. Ce qui limite la complexité des analyses possible vu qu'il n'est pas possible qu'il y ait de dépendance entre les règles.

2.4.2.1.2 DejaVu

DejaVu [8] est un système de traitement d'événement utilisant MySQL. C'est un système déclaratif de reconnaissance de modèle à travers un flux d'événements en direct ou précédemment enregistrés. Dans cette section, nous décrivons certains aspects du langage de requête utilisé par DejaVu et de son architecture. Ce système se concentre sur la mise en place d'un système de reconnaissance de patrons général pour des flux d'événements. En d'autres termes, DejaVu utilise un langage de reconnaissance de modèle qui n'est pas spécifique à un domaine d'application en particulier.

Architecture de DejaVu: Le système est bâti comme une couche au-dessus du système de base de données libre MySQL. DejaVu exploite la possibilité d'échanger le moteur de sauvegarde de MySQL pour ajouter ses propres moteurs de sauvegarde. Il possède deux types de sauvegarde :

- La sauvegarde de flux en direct : Il s'agit d'une sauvegarde en mémoire qui est structuré comme une queue. Chaque événement à sauvegarder est empilé dans la queue de façon a garder l'ordre d'arrivée des événements dans la mémoire. Le système possède deux modes de requête parmi lesquels il est possible de commuter : le mode tirer et le mode pousser. Sous haute charge, on passe en mode tirer et la sauvegarde gère le problème.

- La sauvegarde de flux archivés : Dans ce mode, le stockage d'archives respecte l'ordre prédéfini des événements et permet des mises à jour sous forme d'ajouts. Ce mode est adapté pour les compressions de données et les requêtes sur des scénarios dans le temps.

Le système dispose d'un mécanisme de suivi de l'évolution des requêtes. Cela consiste en fait au suivi des étapes des requêtes à l'aide de machines à états.

Syntaxe de DejaVu : DejaVu utilise le langage MySQL pour effectuer ses requêtes. Les requêtes sont effectuées par l'intermédiaire d'une clause MySQL "MATCH_RECOGNIZE" reliée à la fois aux tables de flux directs et aux tables de flux archivés.

La requête suivante est un exemple de scénario issu du domaine des marchés financiers et décrit avec DejaVu. La description complète de ce cas d'utilisation de DejaVu est disponible dans [21]. La clause "MATCH_RECOGNIZE" n'est pas explicitement déclarée dans cet exemple, mais est la clause de base utilisée pour réaliser n'importe quelle requête DejaVu :

```
SELECT min_tstamp_l, symbol_l, min_price_l,
       initial_price_a, min_price_a, max_price_a
       FROM LiveStock MATCH_RECOGNIZE(
       PARTITION BY Symbol
       MEASURES A.Symbol AS symbol_l,
       MIN(B.Tstamp) AS min_tstamp_l,
       MIN(B.Price) AS min_price_l
       ONE ROW PER MATCH
       AFTER MATCH SKIP PAST LAST ROW
       INCREMENTAL MATCH
       PATTERN (A B+)
       DEFINE /* A matches any row */
       B AS (B.Price<A.Price AND B.Price<=PREV(B.Price))
       ), ArchivedStock MATCH_RECOGNIZE(
       PARTITION BY Symbol
       MEASURES A.Symbol AS symbol_a,
       A.Price AS initial_price_a,
       MIN(B.Price) AS min_price_a,
       LAST(D.Price) AS max_price_a
       ONE ROW PER MATCH
       AFTER MATCH SKIP PAST LAST ROW
       MAXIMAL MATCH
       PATTERN (A B+ C* D+)
       DEFINE /* A matches any row */
       B AS (B.Price<A.Price AND B.Price<=PREV(B.Price))
       C AS (C.Price>=PREV(C.Price) AND C.Price<=A.Price)
       D AS (D.Price>PREV(D.Price) AND D.Price>A.Price)
       )
       WHERE symbol_l = symbol_a;
```

Discussion : DejaVu est un langage déclaratif simple pour la description de scénarios. Il permet de décrire des scénarios variés pour différents types d'applications sous forme de requête MySQL. C'est un système de traitement d'événements capable de fonctionner en mode en ligne et hors ligne. Il pourrait très bien être appliqué aux traces d'exécution de noyau. De plus, il utilise une machine à état pour évaluer les patrons.

2.4.2.2 Langages impératifs

Les langages impératifs sont orientés sur comment le programme se comporte et non sur les résultats désirés, comme c'est le cas avec les langages déclaratifs. Il s'agit de citer toutes les instructions qui modifient la dynamique du programme. C'est un type de langage très répandu. Les langages procéduraux comme le C++ ou JAVA sont des langages impératifs à usage général. Ce type de langage est populaire dans les systèmes de détection d'intrusion ("IDS") ainsi que pour les systèmes d'analyse de trace. Dans cette section, nous décrivons RUSSEL, un langage impératif utilisé dans le système de détection d'intrusion ASAX, ainsi que DTrace pour l'analyse de trace.

2.4.2.2.1 RUSSEL

RUSSEL ("RUle-baSed Sequence Evaluation Language") [1] est un langage impératif basé sur les règles et spécialement dédié à l'analyse de traces de transactions dans ASAX. Il est possible de détecter des intrusions dans ASAX à partir de règles définies avec RUSSEL. À l'opposé de SNORT, RUSSEL offre la possibilité qu'une règle puisse être déclenchée par une autre. Dans cette section, nous décrivons le fonctionnement du mécanisme de déclenchement de règles de RUSSEL et présentons un exemple de règle de détection d'intrusion avec RUSSEL.

Mécanisme de déclenchement de règles : La structure de contrôle de RUSSEL est basée sur un mécanisme de déclenchement de règle qui se résume à ceci [1] :

1. Le cheminement d'information est analysé, enregistrement par enregistrement, par un ensemble de règles. Ainsi, à un instant donné, il y a un nombre précis de règles qui sont actives.

2. Les règles actives possèdent des informations sur des analyses précédentes qui pourraient être utilisées pour traiter l'enregistrement en cours d'analyse.

3. Du point de vue du programmeur, une règle est une sorte de procédure paramétrisée qui peut avoir des instructions qui réalisent des actions.

4. Déclencher une règle implique de fournir les paramètres nécessaires et de préciser à quel moment la règle doit être déclenchée.

5. Le processus est initialisé par un ensemble de règles qui est activé par le premier enregistrement.

Les figures 2.5 et 2.6 tirées de [1] présentent un exemple d'ensemble d'instructions écrites avec le langage RUSSEL. Cet exemple décrit 2 règles utilisées afin de détecter des tentatives abusives de pénétration du système par un utilisateur. La première règle décrit le scénario qui décèle une première tentative échouée, ce qui déclenche la seconde règle qui compte le nombre de tentatives échouées sur une certaine période par le même utilisateur.

```
rule Abusive_user          (maxtimes , duration : integer)
# This rule detects a first failed command and triggers off an accounting rule;
begin
if (        (evt='chown' and res='failure' ) or (evt='chmode' and res='failure' )
     or     (evt='cd' and res='failure' )        or (evt='open' and res='failure') or … )
   —>
   Trigger off for next Count_rule2 (maxtimes-1, timestp+duration, userid)
fi ;
Trigger off for next  Abusive_user (maxtimes , duration)
end
```

Figure 2.5: Première règle décrite avec RUSSEL

```
rule      Count_rule2   (countdown , expiration , suspectid : integer)

# This rule counts the other failed commands of the suspected user,
# it remains active until its expiration time or until the countdown becomes 0

if (      (evt='chown' and res='failure' ) or (evt='chmode' and res='failure' )
    or    (evt='cd' and res='failure' )      or (evt='open' and res='failure') or ... )
  and     userid =suspectid
  and     timestp < expiration
  —>
  if      countdown>1
          —>  Trigger off for next
                      Count_rule2  (countdown-1, expiration, suspectid);
          countdown=1
          —> SendMessage  ("too much access break attempts for:", suspectid)
  fi

  timestp ≥ expiration
  —>  Skip  ;

  true
  —> Trigger off for next Count_rule2(countdown,  expiration , suspectid)

fi
```

Figure 2.6: Seconde règle décrite avec RUSSEL

Discussion : RUSSEL est un exemple intéressant de langage impératif. Il supporte la fonctionnalité de déclenchement d'une règle par une autre contrairement à SNORT. Par contre, la description de scénarios complexes impliquant plusieurs ressources est difficile à réaliser. De même, le système ne permet pas de gérer plus d'une instance d'un même scénario simultanément. Ce qui constitue une limitation importante pour les systèmes à multi-processeurs.

2.4.2.2.2 DTrace

Dtrace [9] [4] est un traceur dynamique réputé pour sa performance. Une description de DTrace a été couverte dans la section 2.1.1. Dans cette section, nous nous intéressons plus au langage 'D' utilisé pour décrire comment instrumenter les systèmes.

Architecture des programmes D : D est un langage de description similaire à C qui supporte tous les opérateurs ANSI de base. Le langage offre la possibilité d'accéder aux types natifs du noyau du système d'exploitation ainsi qu'aux variables globales. Les programmes écrits avec le langage D sont compilés en DIF par un compilateur intégré de DTrace. Le DIF est ensuite

empaqueté sous forme de fichier mémoire et envoyé dans le noyau interne de DTrace pour la validation et l'activation des sondes.

Syntaxe de D : La commande *dtrace* (1M) fournit une interface frontale générique pour le compilateur D et DTrace. Un programme D se compose d'une ou plusieurs clauses qui décrivent l'instrumentation qui doit être activée par DTrace. La figure 2.7, tirée de [4], présente le format générique des clauses pour chaque sonde.

```
probe-descriptions
/predicate/
{
        action-statements
}
```

Figure 2.7: format générique des clauses avec le langage D

La description de sonde ("probe-description") dans la figure 2.7 est spécifiée sous la forme *fournisseur:module:fonction:nom*. Les prédicats et les actions de D sont similaires à celles du langage C. La figure 3.6 est un exemple de script simple écrit avec le langage D. Cet exemple affiche le temps pris par chaque fil d'exécution pour exécuter l'appel système *read(2)*. Une description plus détaillée de cet exemple est disponible dans [4].

```
syscall::read:entry
{
        self->t = timestamp;
}

syscall::read:return
/self->t/
{
        printf("%d/%d spent %d nsecs in read\n",
            pid, tid, timestamp - self->t);
}
```

Figure 2.8: Exemple de script décrit avec le langage D

Discussion : La syntaxe du langage D est similaire à celle du langage C. Ce qui facilite sa prise en main par les développeurs. Par contre, le langage n'offre pas la possibilité de mettre en place des scripts basés sur des machines à états. Les patrons complexes sont donc difficiles à décrire. De plus, l'utilisateur est souvent obligé de coder manuellement les patrons dans les gestionnaires de sondes.

2.4.2.3 Langages à automates

Il s'agit des langages de description basés sur les machines à états. Dans cette section, nous décrivons STATL, un langage dédié à la description de scénarios d'intrusion.

2.4.2.3.1 STATL

STATL [10] (State Transition Analysis Technique Language) est un langage à transition/état pour la description d'intrusion. La description de scénarios d'attaque avec STATL peut être utilisée sur des flux d'événements dans les systèmes de détection d'intrusion (« IDS ») afin de détecter des attaques en cours. STATL peut être porté sur différents environnements et offre la possibilité d'être modifié afin de s'adapter à l'environnement dans lequel il est utilisé. Par exemple :

- USTAT est une implémentation de STATL pour détecter les intrusions dans le système d'exploitation UNIX.
- NetSTAT permet de détecter les intrusions sur le réseau en temps réel.
- WinSTAT permet de détecter les intrusions dans l'environnement Windows NT.

Il est capable de décrire parfaitement aussi bien les attaques sur le réseau que les attaques sur des hôtes.

Architecture de STATL : STATL est le langage utilisé pour décrire les scénarios d'attaques dans STAT (State Transition Analysis Technique). Le système étant portable sur différentes plates-formes, STATL offre donc une interface commune pour la description de scénarios. Le système possède un module noyau afin de détecter les scénarios décrits, dans un flux d'événements. Il est possible d'ajouter des extensions à l'architecture du système.

Syntaxe de STATL : STATL est un langage de description d'événements qui permet de décrire les scénarios d'intrusion dans un système. Le langage supporte la plupart des types de données

standards : *int, u_int, bool*… Il utilise le mot clé *scenario* pour déclarer chaque scénario d'attaque. Chaque scénario contient tous les états et transitions qui définissent la signature de l'attaque. Le langage offre aussi la possibilité de définir des fonctions à l'intérieur de la description d'un bloc de scénario. La figure 2.10 donne une description complète de la signature de l'attaque "*ftp-write"* pour une implémentation USTAT de STATL. Une description complète du schéma de l'attaque est décrite dans [10]. La figure 2.9 offre une vue graphique des états et transitions de l'attaque.

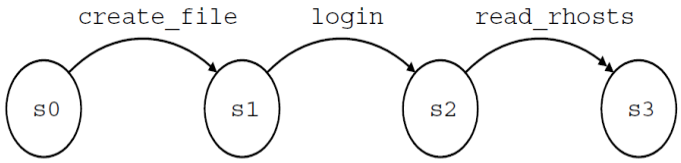


Figure 2.9: Diagramme de transition d'état pour l'attaque *ftp write*

Discussion : STATL est un langage extensible qui permet de décrire des patrons d'attaques. C'est un langage portable qui peut être adapté à l'environnement dans lequel on l'utilise. C'est un très bon exemple de langage descriptif à machine à états. Il gère très bien la description de scénarios complexes faisant interagir plusieurs ressources. Il permet aussi de générer des événements synthétiques et ainsi de réaliser de l'abstraction.

```
use ustat;
scenario ftp_write
{
  int user;
  int pid;
  int inode;

  initial state s0 { }

  transition create_file (s0 -> s1)
    nonconsuming
  {
    [WRITE w] : (w.euid != 0) &&
                    (w.owner != w.rui
    { inode = w.inode; }
  }

  state s1 { }

  transition login (s1 -> s2)
    nonconsuming
  {
    [EXEC e] :
      match_name(e.objname, "login")
    {
      user = e.ruid;
      pid = e.pid;
    }
  }

  state s2 { }

  transition read_rhosts (s2 -> s3)
    consuming
  {
    [READ r] : (r.pid == pid) &&
                   (r.inode == inode)
  }

  state s3
  {
    {
       string username;
       userid2name(user, username);
       log("remote user %s \
         gained local access",
         username);
    }
  }
}
```

Figure 2.10: Description de l'attaque *ftp-write* avec STATL

2.5 Conclusion de la revue de littérature

Il existe de multiples traceurs : Dtrace, LTTng, Perf, SystemTap. Ces traceurs permettent de récolter les données d'exécution du système. Cependant, les données recueillies sont

volumineuses et difficiles à comprendre. De nombreuses méthodes d'abstraction basées sur le contenu de la trace, sur les métriques, sur la visualisation ou encore sur les ressources, ont été développées dans le but de réduire la taille des données et faciliter la recherche d'informations dans la trace. Toutefois, ces méthodes présentent des faiblesses telles que la difficulté à gérer les traces volumineuses et la possibilité de partager les informations. De même, il existe plusieurs langages de description de patrons : SNORT, DejaVu, RUSSEL, Dtrace, STATL. Néanmoins, ils présentent quelques lacunes telles que la difficulté de décrire des patrons complexes, la difficulté de supporter plusieurs instances de patron et le problème de dépendance entre les patrons. Ce travail de recherche s'inspire des lacunes et faiblesses énumérées dans cette revue de littérature pour concevoir un langage de description de patrons ainsi qu'un analyseur de trace pour filtrer et rechercher des patrons préalablement décrits dans la trace. La démarche suivie pour concevoir notre solution est présentée au chapitre suivant.

CHAPITRE 3 MÉTHODOLOGIE

Dans ce chapitre, nous décrivons comment les critiques soulignées dans la revue de littérature ont motivé les besoins. Nous présentons la démarche de l'ensemble du travail ainsi que l'organisation générale de ce document.

3.1 Identification des besoins

La revue de littérature a permis d'identifier une multitude d'outils de traçage. Dépendamment des besoins, le choix du développeur peut se porter sur l'un ou l'autre de ces outils. Il est donc nécessaire que notre solution soit capable de fonctionner avec chacun des formats de trace identifiés.

Ensuite l'examen des techniques d'abstraction a soulevé plusieurs fonctionnalités dont doit disposer notre solution. L'approche que nous proposons doit fonctionner adéquatement avec de larges traces. Il y a également un besoin de partage d'informations entre les patrons. Cette approche doit également permettre de générer, jusqu'à une certaine mesure, des métriques dont la description en elle-même pourrait constituer des patrons.

Enfin, les limites ainsi que les atouts des langages de description de patron étudiés dans le chapitre 2 permettent de définir certains critères auxquels notre solution doit répondre. En effet, la solution proposée doit offrir la possibilité à un patron d'en déclencher un autre. De plus, plusieurs instances d'un même scénario devraient être possibles. Exploiter les machines devrait permettre également de définir des scénarios complexes qui rencontrent les besoins des traces d'exécution. De plus, le langage proposé devrait être suffisamment complet pour ne pas que les utilisateurs aient à modifier manuellement le code de l'analyseur de trace.

3.2 Démarche de l'ensemble du travail

L’approche envisagée est de se baser sur le module d’analyse XML de Trace Compass et de l’étendre pour ajouter un outil de filtrage flexible de la trace. Premièrement, nous allons identifier un ensemble de patrons de base que l’on souhaiterait appliquer comme filtre sur la trace. Ces patrons vont nous permettre d’identifier les caractéristiques des filtres. Ensuite, le module d’analyse XML de Trace Compass permet au visualiseur de supporter des analyses de la trace décrite en XML. Le langage XML étant extensible, nous allons nous concentrer sur la conception

d’un langage de description de patrons basé sur le XML pour décrire les filtres qu'on appliquera à la trace. Pour ce faire, nous allons définir un ensemble de balises et attributs capables de représenter les caractéristiques des patrons que nous aurons déjà identifiés.

Puis, nous allons étendre le module d’analyse XML de Trace Compass pour supporter les filtres décrits en XML. Nous allons ajouter au module la capacité de charger les patrons décrits avec le langage proposé, dans l’environnement du visualiseur Trace Compass et de créer des modèles d’analyses qui vont filtrer la trace.

Ensuite, nous allons développer un moyen de représenter efficacement le résultat des filtres. Pour ce faire, nous allons implémenter des vues qui vont fournir plus de détails sur l’exécution des filtres et synchroniser le résultat de ceux-ci avec les autres analyses de Trace Compass. Nous avons choisi de rassembler les patrons dans une vue tabulaire qui donnera les informations détaillées sur le patron. Chacun des patrons présents dans la vue aura son intervalle de temps synchronisé avec les autres analyses de Trace Compass de façon à pouvoir le repérer. De plus, on utilisera un diagramme de Gantt temporel pour suivre l’évolution des patrons dans le temps ainsi que les données qui leur sont reliées.

De plus, nous allons évaluer l’efficacité de notre approche par rapport aux analyses habituelles de Trace Compass et aussi par rapport à d’anciens travaux semblables à notre recherche. Cette efficacité sera déterminée en fonction du temps pris pour filtrer la trace et détecter les patrons.

Finalement, nous allons continuellement décrire de nouveaux patrons, plus complexes, avec le langage descriptif que nous aurons établi. Cela permettra d’améliorer sa structure et les fonctionnalités supportées. Pour ce faire, nous allons parcourir les travaux antérieurs; à la recherche d’intéressants exemples de patrons.

3.3 Organisation générale du document

La suite de ce document est organisée comme suit : le chapitre 4 est sous forme d’article et présente l’essentiel des travaux de recherche ainsi que quelques exemples concrets de l’utilité de l’approche développée et les tests de performance de l'approche. Ensuite, le chapitre 5 est une discussion générale qui revient sur les résultats obtenus, possiblement pour confirmer que les objectifs identifiés ont été atteints. Enfin, le chapitre 6 fait une synthèse des travaux et émet des suggestions pour de possibles améliorations.

CHAPITRE 4 ARTICLE 1 : A DOMAIN SPECIFIC LANGUAGE FOR PATTERN MATCHING, FILTERING AND ANALYSIS OF EXECUTION TRACES

Authors

Kadjo Kouame
Ecole Polytechnique Montreal
kadjo.kouame@polymtl.ca

Naser Ezzati-jivan
Ecole Polytechnique Montreal
n.ezzati@polymtl.ca

Michel R. Dagenais
Ecole Polytechnique Montreal
michel.dagenais@polymtl.ca

Submitted to Software : Practice and Experience

Keywords Performance analysis, Tracing, System state analysis, Declarative language, pattern matching, Filtering.

4.1 Abstract

Dynamic analysis generates execution traces that can be used to study the run-time behavior of systems. However, the huge amount of data in an execution trace may complexify its analysis. Moreover, users are not interested in all the events from the trace and may want to find some pattern in the data. For instance, one may need to study the interprocess communications or detect security attacks, hence the interest for a flexible abstraction approach. Abstraction is used to generate an enhanced trace, with a reduced size and complexity, or to discover some specific sequences of data. The approach proposed in this paper allows to define custom filtering patterns, declaratively in XML, to abstract

the trace and check for the presence of patterns. The proposed solution is a stateful approach covering various analysis patterns with different levels of complexity. The paper is an improvement from our previous work. It provides more details on this data-driven filtering approach, more performance results and some interesting use cases for the trace events generated by the LTTng Linux kernel tracer.

4.2 Introduction

The Performance of distributed systems is continuously increasing with the integration of multiple cores and several computer nodes. Developers face challenges such as concurrency control, data synchronization and security. Problems happening in these systems could be difficult to detect and understand. It becomes hard to find and locate flaws in these systems because they are continuously running, in parallel with other clients and servers, and cannot be stopped for debugging. Furthermore, some of the suspected problems are timing related, occur rarely and are not easily reproducible. Dynamic analysis, through tracing, seems the best solution for testing and evaluating distributed systems and todays other complex systems. It records the system execution data into a trace file.

Multiple tracers exist and provide different ways to record and store the data efficiently, according to the needs. The trace file is a log-like file with all the information that could help the developers to understand the behavior of the system. The data is arranged in the form of events which are necessary to model the evolving state of the system. Multiple tools have been developed in the last years to analyze the data. Visualizers, such as Trace Compass[1], propose a set of useful analyses that display details on many aspects of the trace. Likewise, several other techniques exist to extract data from the trace.

However, analyzing the trace may be difficult because of the huge amount of data recorded. Thousands of events may be reported in a few seconds and only a very small subset is significant

[1] https ://projects.eclipse.org/projects/tools.tracecompass

to the users. Thus, it becomes necessary to reduce the size of the trace. Abstraction is a proper solution for trace reduction. One of the main abstraction techniques requires to match some patterns within the trace to be fully helpful and effective. There are many proposals in the literature that use pattern based techniques to abstract or filter out the trace data [3] [30] [22] [2]. However, they usually offer a constant set of patterns, mostly hard-coded in the system, which reduces the flexibility of the approach. Users are only limited to the predefined patterns of abstraction. Also, the solutions proposed are mostly designed for a specific purpose such as the understanding of concurrency bugs. Our solution solves these problems by allowing users to define their own analysis patterns using a declarative pattern description language, with XML syntax, and import them dynamically at analysis time. This enhances the flexibility and usability of the approach.

The remainder of the paper is organized as follows: after discussing the related work, the details of our solution are presented. Then are presented some concrete use-cases and a performance evaluation of the approach.

4.3 Literature Review

Trace Compass (previously “TM”) is a trace visualizer which is an Eclipse plug-in developed by the DORSAL laboratory and its partners. The visualizer represents the data extracted from the trace using a state machine. Montplaisir et al. [26] proposed an efficient solution to model the state at any point within the trace. Then, Wininger et al [38] improved the capacity of Trace Compass to support all trace types through a flexible XML-based analysis, and suggested the possibility to filter the trace with his flexible analysis core. In this section, we first present their solutions. Then, we describe the abstraction techniques used so far to reduce the complexity of the trace. Finally, we present some of the existing description languages for data-driven analysis.

4.3.1 Modeling the state system

4.3.1.1 State history tree

The solution proposed by Montplaisir et al. [26, 27, 28] uses a new data structure called ‘*state history tree’* instead of a previous snapshot-based method. The snapshot method stores several

snapshots of the trace at regular intervals. When a request is made at a timestamp *t*, the system loads the closest snapshot and starts to read all the events in the trace from the snapshot timestamp until it reaches the desired timestamp. This method is inefficient when the trace is very large, because there are so many more events and there is more data to read for each snapshot.

The history tree is an optimized data structure for intervals. It writes the data directly on disk. All the state intervals produced by the state system are sorted in ascending order of interval end time, before being inserted into the data structure. There is no need to rebalance the tree during construction. The structure uses nodes of constant size as container for the state intervals. These nodes are arranged to form a tree. The structure allows fast queries for any timestamp in the trace range, with a logarithmic search time. It is possible to stop the construction of the tree at any time and to restart later.

4.3.1.2 State system

In this section, we describe the main components of the state system and how the flexible analysis of Wininger et al. [38] accesses them.

4.3.1.2.1 Attribute tree

Montplaisir et al. [26] described an attribute as a single unit in the model that can only have one state at a time. For example, a thread could have an attribute *status* and this attribute can have the following possible values (state): unknown, blocked, user mode, system call, interrupted, waiting for CPU, etc.

The attributes may be related to one another. In fact, they could be complementary properties of the system or its resources. A thread, for example, can have other attribute like its PPID, the name of the running program, the name of the system call, the priority, etc.

The attributes are arranged into a tree called the *attribute tree*. All the childs of a node, in the tree, should be related and represent an attribute describing the state of their parent. Figure 4.1 shows a representation of the attribute tree for a specific thread. A request to know the value of an attribute can be made using its path in the tree.

4.3.1.2.2 Event handler

It is that component that turns events into state changes. Each state change has three properties: 1- The timestamp of the event, 2- an attribute, 3- a state value generated from the event.

The event handler receives the events from the trace and generates state changes. There are actually five supported operations in the tool to describe the state changes: modify, remove, push, pop, increment. It is possible, for an event, to generate several state changes at a time. For example, a scheduling change event affects the state of the outgoing and incoming threads.

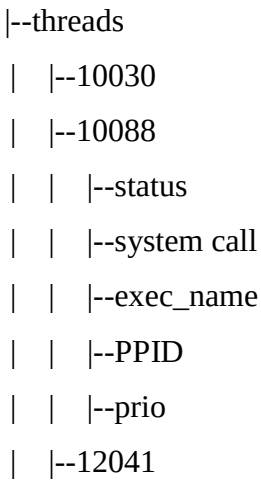


Figure 4.1: Example of attributes disposition in the attribute tree

It is possible to add new analyses in Trace Compass. Each of them defines its own way to model the system and create states. Thus, an event handler is necessary to specify how states will be created from events. The event handler had to be manually added into Trace Compass. Wininger et al. [38] offered the possibility to define external event handlers without adding any source code to Trace Compass. The handlers could then be designed using a declarative XML-based description language. The user can describe how to generate state changes outside of the Trace Compass environment, regardless of the implementation. They also added the possibility to generate views whose behavior and data are described directly with their proposed declarative language.

4.3.2 Trace abstraction

Many abstraction techniques exist in the literature to reduce the complexity of the trace and the size of the data. Ezzati et al.[13] presented a taxonomy of the trace abstraction techniques based on their usage in a multi-level vizualisation tool. They identified three categories of abstraction : 1- Content-based (data-based) abstraction techniques, the techniques that use the content of the events to abstract the trace. 2- Visual abstraction, those techniques mostly used to ease the vizualisation process of the trace. 3- Resource abstraction techniques, the techniques to organize and represent the resources information available in the events.

The tool described in this paper belongs to the first category. In this section, we will describe some of the interesting content-based abstraction techniques in the literature. Many content-based abstraction techniques have been developed in order to reduce the size of the traces. Most of them are based on events removal, aggregation and trace grouping.

Trace filtering allows to remove irrelevant and noisy events (for the intended application) from the trace and to highlight those which are not. The trace can be filtered using criteria like the timestamp, the event type, the thread id, the running CPU, etc. Fadel et al. [15] filtered kernel trace data in order to remove unimportant data, such as the memory management events, and noise like page faults. Also, most of the vizualisation tools like Trace Compass propose ways to filter the trace data based on the events content.

Frequent mining techniques combine aggregation and trace generalization to reduce the trace complexity. It is used to achieved several goals. In [3], Befrouei et al. used a sequential pattern mining algorithm with traces in order to detect and expose critical concurrency. They aggregate the events by mapping sequences of concrete events to single abstract events. They removed the *spurious* and *misleading* patterns from the refined trace, so they only processed the relevant ones.

Pirzadeh et al. [30] exploited text mining techniques to find the most relevant events in the trace. They read the trace as if it is a series of execution phases. They used two gravity schemes to create the phases at runtime. The first scheme, of the similarity gravity, is used to move and group similar events to form dense groups that could constitute execution phases. The similarity between two events is determined based on criteria like the timestamp or the thread. The second

scheme, of continuity of gravity, groups the data within the trace based on the nesting level of the routine calls. They also remove the irrelevant events from the original data using a relevant threshold.

LaRosa et al. [22] used frequent itemset pattern mining to detect inter-process communication problems. They employed window slices to divide the trace into sets of items. Similarly, Alawneh et al. [2] used pattern recognition techniques, based on two algorithms, to detect inter-process communications. The first algorithm detects frequent patterns in the trace and the second algorithm is used to match predefined patterns in the trace. They divided the trace into N sub-traces; each of them corresponding to a process. They used the N-gram extraction technique in both algorithms and aggregate the contiguous events to reduce the size of the sub-traces.

Waly et al. [13] detected suspicious behavior within kernel traces using trace grouping and aggregation techniques. They proposed a well-defined declarative pattern description language to describe problematic behaviors. They designed a detection engine that receives the compiled patterns and analyzes the trace.

Matni et al. [25] used predefined patterns to discover faulty behavior within kernel traces. They chose a state machine approach to describe the patterns to match within the trace. They expressed the benefits of this choice based on the simplicity and expressiveness of the state transition language, the domain independence advantage and the ease to generate synthetic events.

Fadel in [15] and also Naser Ezzati et al. in [14] generated synthetic events to realize trace abstraction, using state machines to describe patterns. The patterns are then stored in a library that will help to simplify the analysis and reduce the trace size.

All the abstraction techniques described above try to reduce the size of the trace but they are mostly domain-specific and not extensible. Moreover, proposals from Waly and Fadel “do not offer the needed scalability to meet the demands of large trace sizes”[14]. Also, the approach proposed by Matni et al. does not offer the possibility for the patterns to share information, which limits the possibilities of the tool. Thus, there is a need for a domain independent, and trace format independent, abstraction tool that could easily support large trace files (1GB or more), and where patterns could share information between themselves.

4.3.3 Description languages

The pattern description languages can be separated into several categories. In this section, we present three categories of description languages: the declarative languages, the imperative languages and the automata-based languages.

Declarative languages describe what the program is supposed to do. It focuses on the logical instead of the execution flow. SNORT and DejaVu are two examples of declarative languages. SNORT [35] is a very popular intrusion detection system. It can be used in three modes: sniffer mode, packet logger mode and Network Intrusion Detection System mode (NIDS mode). The sniffer mode reads the packets from the network and presents them like an information stream. In the second mode, SNORT saves the packets on disk in binary format. The data can be subsequently read using the BPF interface for filtering. With the NIDS mode, SNORT prevents and detects intrusions, and analyzes the network traffic. Packets are identified as malicious if they match the user-specified rules. However, there is no possible dependency between the rules since a rule cannot trigger another one.

DejaVu [8] is built on top of MySQL and uses the “MATCH_RECOGNIZE” clause to define its requests. This declarative system recognizes the pattern through an event stream in both online and offline modes. The proposed language is not necessarily specific to an application domain and uses a state machine to drive the requests.

By opposition to declarative languages, imperative languages show how the program behaves. All the instructions that modify the program's dynamics are quoted. The RUle-based Sequence Evaluation Language (RUSSEL) [1] and the D language are two examples. RUSSEL is rule-based and is especially dedicated to the transaction traces in the detection system called ASAX. In fact, intrusions in ASAX are detected thanks to the rules defined with RUSSEL. Unlike SNORT, RUSSEL offers the possibility for one rule to be triggered by another. On the other hand, it is not possible to have more than one instance of a pattern simultaneously.

The D language [4] is used with the DTrace tracer to define how to instrument systems. It uses the *dtrace* command as frontend interface for the D compiler. It has access to the native types of the operating system core and to the global variables. Dtrace is a very effective tracer and the D

language can support all the basic ANSI operators. However, users sometimes have to manually encode the patterns into the probes using the C language, since complex patterns are difficult to describe with the language.

The third category of description languages presented here is the automata-based languages. STATL (State Transition Analysis Technique Language) [10] is one of them. It is a transition/state language used for intrusion detection patterns. It is used on the events stream inside intrusion detection systems (IDS). Moreover, STATL can be exported to different environments and can be modified to adapt to its targeted environment. The language is well suited to deal with execution traces and allows to realize abstraction by generating synthetic events. The stateful approach enables to define complex scenarios.

4.4 Architecture

As mentioned in the previous section, we discussed the existing techniques for trace abstraction based on data. We faced the limits of these techniques: most are application specific, some do not support well large trace files, and there is a need to share information between the patterns.

The tool proposed in this paper is a data-driven approach for generic trace filtering. The tool allows users to define declaratively in XML their own filters through a stateful pattern description language. The language developed allows the definition of a variety of patterns : system monitoring patterns, security attack patterns, program testing patterns, system performance patterns, etc. It uses the state system and the state history tree of Trace Compass, whose data arrangement allows the sharing of information between patterns. In the following sections, we describe the architecture of our declarative solution. First, we explain how the data is processed in the system. Then, we describe how it is stored. Finally, we present how we used the visualization tool to support our work.

4.4.1 Data processing

There exist in the literature many advanced tools that aim to abstract and filter data within traces using a pattern description approach. However, most of them are hardly extensible and have difficulties to process large trace files. In the data-driven approach proposed, users can define

their own custom patterns to abstract and filter information out of the trace data (at different granularity levels) and output only the interesting (aggregated) part of the trace data. The general architecture of the tool is displayed in Figure 4.2. This generic data-driven trace filtering method works in two main steps:

- Model creation : It is the first data processing step. Users define their own patterns using the proposed declarative pattern description language. The patterns are described using any text editor, outside of the Trace Compass environment. Then, the event handler inside the filter analyzer parses the pattern and creates JAVA models from that. All the entities (finite state machines, transition, actions, …) useful to define a scenario are transformed into JAVA models that will be used for the event filtering process in the second step.
- Event filtering: The generated models are used to filter the data within the trace and generate the desired output. The trace analyzer receives the events from the trace. It is responsible for creating scenarios when needed. A scenario is an instance of a pattern modeled at the creation step. The event handler, of the filter analyzer, processes each event one at the time. Depending on the information available in the ongoing event, it is passed to the relevant finite state machine (FSM). Then, the FSM transfers the event to its scenarios. Finally the scenarios handle the event according to their current state and have the possibility to generate outputs. Not all events are relevant for the pattern described and some may be ignored.

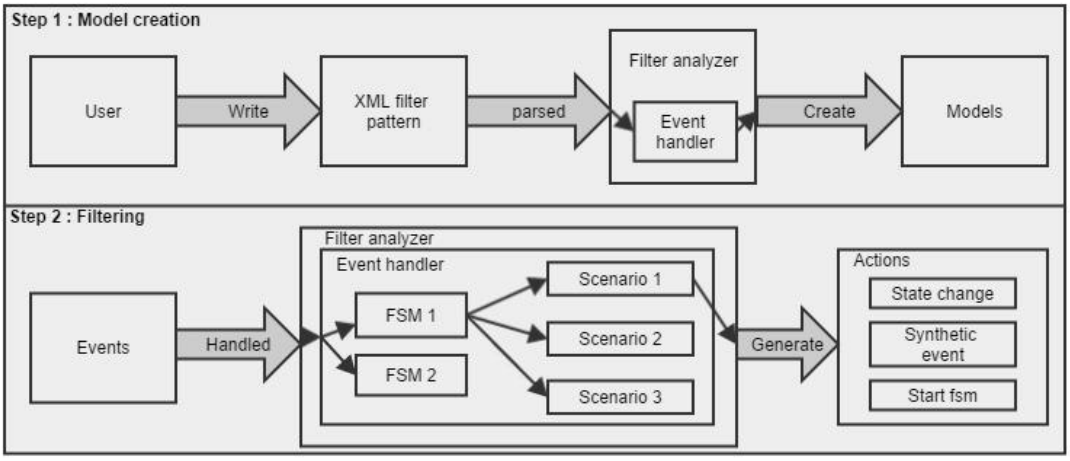


Figure 4.2: Data processing in the filter analyzer

4.4.2 Data container

The tool described in this paper uses the state system architecture, explained in the literature review, to generate and store the data. The attribute tree is a generic component that acts like a temporal database. In our design, there is no limitation on how users can define their models. The model is built and stored using the state system and exploited later for analysis and visualization of the information.

A state value is created for each data that we want to save from the pattern. We use the state system and the history tree to store both internal data and external data. The internal data contains details about the state machine during its execution, while the external data is defined by the users. This makes the FSM state visible in order to easily verify and understand their operation, and fix eventual errors in their definition.

Various information can be saved into the state system: timestamp, number of matched scenarios, fields, status, states, etc.… These fields can be stored into the history tree in a way that patterns can share information using the suitable path for the attribute. This allows sharing some of the generated state and creating the description of more complex patterns.

4.4.3 Visualization

We have built a series of views in Trace Compass to be able to support our tool. The views are analyses that will help the users to have a better hand on the tool.

4.4.3.1 Filter View

The filter view is a tabular view to display the output of the pattern details. It shows all the instances for each pattern executed on the trace data. For each instance, it gives the start timestamp, the end timestamp, the identifier, the scenario name and other additional information. Figure 4.3 depicts this view. This view is synchronized with all the other analyses of Trace Compass, so that we can clearly locate each pattern execution in the trace, regardless of whether it failed or not.

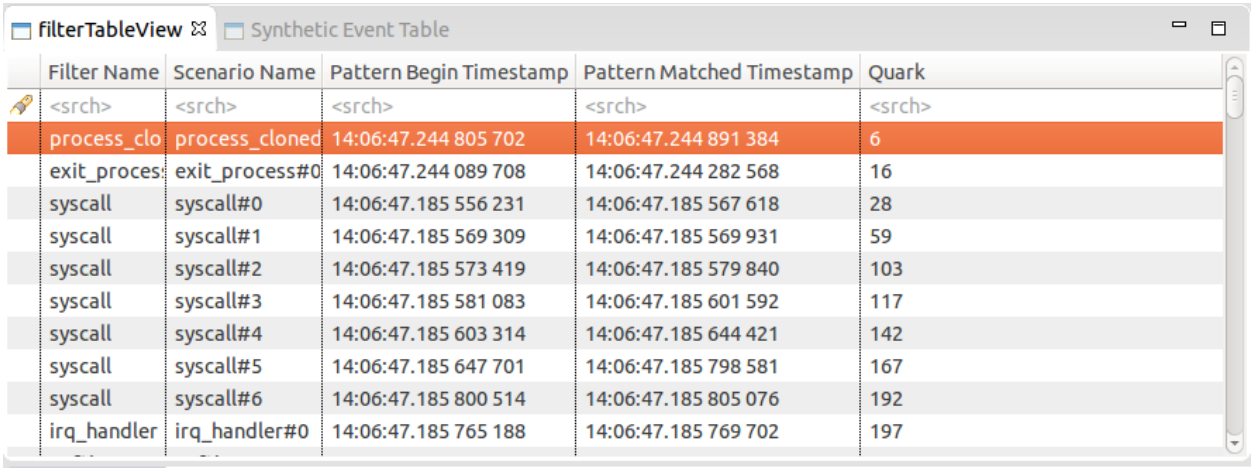


Figure 4.3: A filter table view to display the instances of patterns

4.4.3.2 Synthetic Events View

The pattern matching process described in this paper has the capability to generate synthetic events from the trace data. The synthetic events describe high-level aspects of the trace and can be used as input to the filtering process. Since this type of event may represent a sequence of other events and has two timestamps, a dedicated tabular view has been developed to represent them. Figure 4.4 displays the view. Like the Filter View, the entries of this view are synchronized with the rest of the Trace Compass environment.

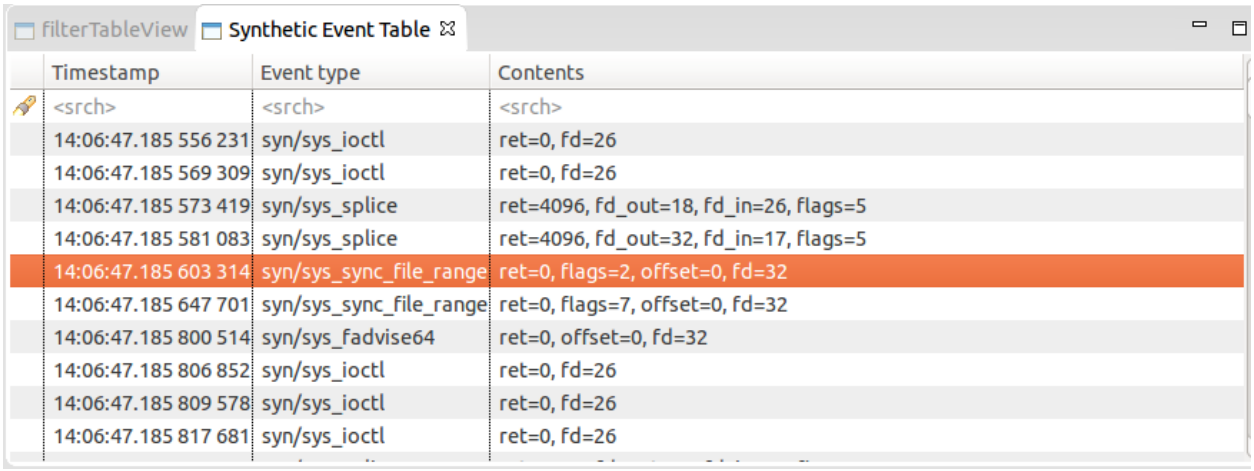


Figure 4.4: A synthetic table view to display the resulting abstract events

4.4.3.3 Filter Status (Debug) View

Storing the internal state in the state system and state history tree brings unique capabilities. We introduced in this work a time graph view that gives the execution details of each pattern. The filter status view enables users to go back and forth in the trace and display the different internal states of the pattern matching processes. It can be used to follow and dig into a pattern to see how and why it matched (or did not). Figure 4.5 depicts this filter status (debug) view.

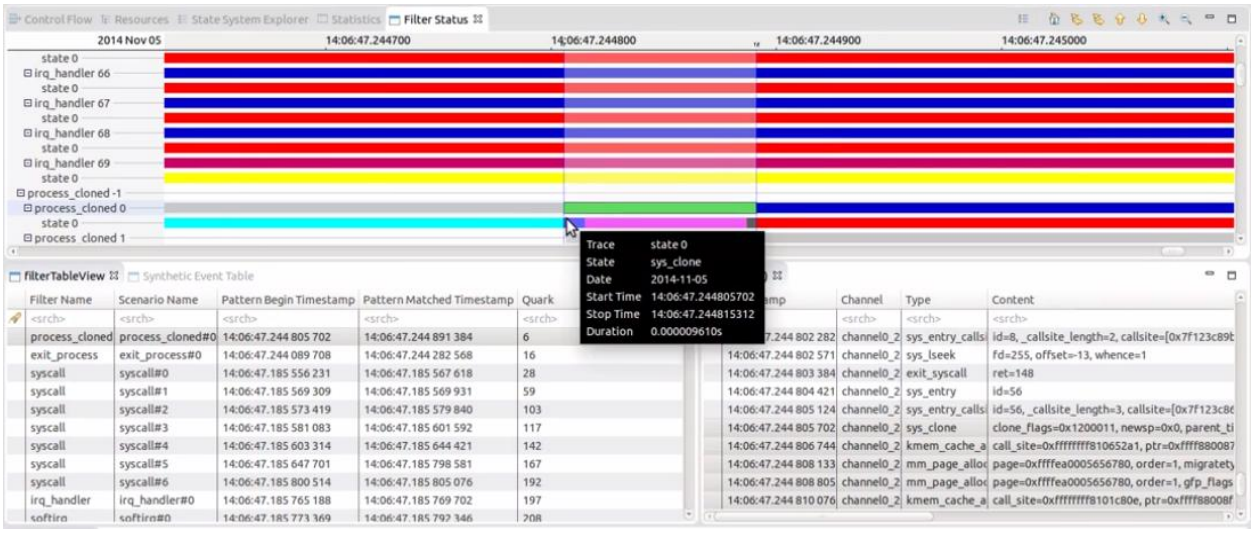


Figure 4.5: A filter status view to display the status of the pattern

The view shown in Figure 4.5 displays the status of the matched pattern (i.e., process cloned) for the given time value (i.e., state 0). Users can go back and forth to follow other possible states of this pattern.

4.5 Language Specification

Our solution allows users to define their own custom patterns. In this section, the details of the general specification of the proposed declarative language are provided. Our approach uses an XML-based declarative pattern language. In this section, we detail how the patterns are defined and modeled. We first expose the motivation for selecting XML to describe the patterns:

- **Simplicity:** Pattern descriptions are accessible since the XML syntax is fairly easy to understand. Users can define their own scenarios. Adding new patterns is then easy.
- **Flexible patterns:** Since the XML language is extensible, we designed our tags such that it is possible to compose patterns for several purposes. The description language proposed is domain independent and supports custom analysis.

The behavior of the event handler is described along with the proposed declarative language. It relies on three main entities to determine how the data within the trace should be processed. We describe here each of these entities.

1) Transition input

The transition inputs are the first elements described in the event handler. Each transition input can be identified in the file through the tag *transitionInput*. They are listed at the beginning of the file, so they can be used by all FSMs described in the file. It defines a (group of) condition(s) that will be used later to design an FSM. Two kind of conditions are supported:

- Conditions based on the event data, these conditions are based on the event type and the data it contains, such as CPU, threads, events fields,...
- Conditions based on the time, these conditions are based on the timestamp of the ongoing event. The timestamp of the ongoing event helps to know the elapsed time since a specific state in the pattern, or can be used to know whether we are within a certain time range. These conditions allow to process things like timeouts, or constrain to within a window of duration X.

The conditions support basic logic operators such as AND, OR, NOT, … Each transition input has an attribute *id* used to reference it.

2) Action

Actions are listed after the transition inputs in the file structure. The *action* tag is used to describe an action. As for the transition inputs, each action also has an attribute *id.* As the name suggests, these are the possible actions to execute. Three types of actions are supported:

- State change: state changes can be generated. It is possible to modify, increment, push, pop,... a state value with this statement. The state values computed can be used to validate conditions or to get more details about the system.
- Synthetic event creation: it is possible to generate a synthetic event in the execution of the state machine. It can then be used either to realize abstraction and reduce the size and complexity of the trace, or also as condition for transition's inputs for later analysis, since the synthetic events are processed just like regular events.
- Start of a new scenario: a new instance of a state machine can be generated by an action. The ID of the FSM must be specified.

3) Finite state machine (FSM)

The finite state machine is the most important entity of the declarative language. It describes the behavior of the pattern. In other words, it defines all possible steps for a scenario. The *fsm* tag allows to define a FSM. Each FSM has two attributes *id* and *multiple*. *multiple* can be true or false and specifies whether several instances of a pattern are allowed or not. An FSM contains a state table that defines all the possible states of the state machine. The FSM also supports pre-conditions and pre-actions in its description. Several FSMs can be described within a file. This allows to process more than one independent scenario simultaneously or even different scenarios with the same FSM. Thus, it is necessary to explicitly specify the FSMs that will be activated at the start of the matching process. The FSMs use the transition inputs and the actions by referencing their IDs.

state definition: It is the tag used to describe one state of the state table of an FSM. Each state definition has a mandatory unique name to label the state. Each state owns a list of *transitions* that define how to switch between the states from this current state.

transition: It is the tag used to describe how to go to the next state from the current one. It has an *input* attribute that is a reference to the transition input id that will trigger the state change. Attribute *next* specifies the next state to reach if the *input* is validated. The attribute *action* makes

a reference to the action that will be executed then. The transition has two other attributes *saveSpecialFields* and *clearSpecialFields* that are described later in the special fields section.

Synthetic events

The synthetic events are events that are manually generated and described using the proposed declarative language. They have the same behavior as the regular events except that they has two timestamps: a start timestamp and an end timestamp. The fields of the event can be directly defined in the description file, and can use a static value or a state value directly. They are then continuously inserted into the trace events queue based on their end timestamp, and processed like the regular events. This enables matching them as they are created. They can then be directly used in the matching process by the FSMs. Figure 4.6 shows an example of synthetic event described with our proposed language.

Special fields

The special fields are not part of the event handler. They are defined before it, using the tag *definedField*. The definition of the special fields has two attributes, name and id, that respectively represent the name of the field in the incoming events and the name that we want to assign to it when it is the time to save the information value. It works as follows: we use the name specified to find the field within the ongoing event. If the field is found, we save its value using the id specified. Then, this value can be used in the filtering process or can be added as a field for a synthetic event at its generation. The tags *saveSpecialFields* and *clearSpecialFields* are used in the transitions to validate if the special fields have to be saved or erased from the database.

```
<synEvent>
<synType>
        <eventName>
            <stateValue type="string" value="softirq"/>
        </eventName>
    </synType>
    <synContent>
        <synField name="cpu" type="long">
            <stateValue type="eventField" value="cpu" />
        </synField>
        <synField name="vec" type="long">
            <stateValue type="query">
                <stateAttribute type="location" value="CurrentCPU" />
                <stateAttribute type="constant" value="vec" />
            </stateValue>
        </synField>
    </synContent>
</synEvent>
```

Figure 4.6: Example of a synthetic event description

A complete example of pattern described using our modeling language is displayed in Figure 4.7. It is a simple pattern example to update the current thread in the system. The rest of the definition uses the basic XML elements developed by Wininger et al. [38] (location, defined value, head, …).

```
<filterHandler filterName="sched_switch">
    <initialFsm id="sched_switch" />
    <transitionInput id="sched_switch">
        <event eventName="sched_switch"/>
    </transitionInput>
    <action id="update current thread">
        <stateChange>
            <stateAttribute type="location" value="CurrentCPU" />
            <stateAttribute type="constant" value="Current_thread" />
            <stateValue type="eventField" value="next_tid" />
        </stateChange>
    </action>
    <fsm id="sched_switch" multiple="false">
        <precondition input="sched_switch"/>
        <stateTable>
            <stateDefinition name="sched_switch">
                <transition input="sched_switch" next="sched_switch" action="update current thread"/>
                <transition input="#other" next="sched_switch" />
            </stateDefinition>
        </stateTable>
        <initialState id="sched_switch"/>
    </fsm>
</filterHandler>
```

Figure 4.7: A complete example of pattern described with the proposed language

4.6 Experiences And Evaluations

In this section, we provide two complete case studies. The proposed the tool is intended to be integrated into Trace Compass. We performed tests to validate the performance of the solution under Ubuntu Linux SPM 14.10 (running kernel version 3.13.0-43), on an Intel core i7 with 8 GB of RAM. We have instrumented the system using LTTng 2.4.0.

4.6.1 Performance analysis

We explained earlier that the analyzer creates models from the user patterns. The models are then used in the filtering phase to process the trace data. Since the creation of the models is a one time process, its cost should be negligible as compared to the magnitude of the whole trace analysis time. The following performance analyses focus on the execution time of the analyzer.

4.6.1.1 Comparison with the JAVA hardcoded approach

In this section, we evaluate our tool on its capacity to process large trace files. We tried to compare our approach with the regular JAVA hardcoded approach.

1) Reading A Trace

Tables 4.1, 4.2 and 4.3 summarize an experimental comparison of both approaches in terms of reading the trace without creating any state change. The comparison is made using huge trace files with up to 90 million events. We can clearly see that there is no performance loss.

Table 4.1: Pattern processing time for a 1 GiB kernel trace

Trace 1 GiB	JAVA	XML
Average time (s)	70,6151	67,6755
Standard Deviation (s)	1,6232	1,7934
Min (s)	67,5900	64,7120
Max (s)	72,8930	69,9050
T-Test	0,0012	

The two data sets are statistically different with a t-test = 0.0012 < 0.01.

Table 4.2: Pattern processing time for a 1.5 GiB kernel trace

Trace 1,5 GiB	JAVA	XML
Average time (s)	108,1156	100,1926
Standard Deviation (s)	2,9696	4,0117
Min (s)	102,3300	93,7140
Max (s)	111,5030	109,0190
T-Test	0,0001	

The two data sets are statistically different with a t-test = 0.0001 < 0.01.

Table 4.3: Pattern processing time for a 2 GiB kernel trace

Trace 2 GiB	JAVA	XML
Average time (s)	144,6816	130,0086
Standard Deviation (s)	4,3270	5,1628
Min (s)	139,4340	123,3870
Max (s)	150,5270	136,5280
T-Test	0,0012	

The two data sets are statistically different with a t-test = 0.0012 < 0.01.

2) Applying a pattern

In a second test, we applied a concrete pattern to the trace. We tried to find all the file accesses within the trace, from their opening to their closing. We described the pattern both with the proposed declarative language using a text editor, and with JAVA directly into Trace Compass. We applied this pattern to three traces of 500MB and 1GB and measured the slowdown percentage. Tables 4.4, 4.5 and 4.6 present the result for this analysis. As expected, we have a gain of performance which decreases linearly proportional to the trace size.

Table 4.4: File access pattern processing time for a 500 MiB kernel trace

Trace 500 MiB	JAVA	XML
Average time (s)	27,0952	24,6457
Standard Deviation (s)	0,9535	1,0766
Min (s)	25,0418	23,5096
Max (s)	28,3119	26,3327
Gain (%)	9,0402	
T-Test	0.0000	

The two data sets are statistically different with a t-test = 0 < 0.01.

Table 4.5: File access pattern processing time for a 1 GiB kernel trace

Trace 1 GiB	JAVA	XML
Average time (s)	57,0497	52,5761
Standard Deviation (s)	2,5640	1,2578
Min (s)	54,6482	50,5621
Max (s)	61,9640	54,3863
Gain (%)	7,8416	
T-Test	0,0002	

The two data sets are statistically different with a t-test = 0. 0002 < 0.01.

Table 4.6: File access pattern processing time for a 1.5 GiB kernel trace

Trace 1.5 GiB	JAVA	XML
Average time (s)	88,8581	82,6866
Standard Deviation (s)	4,1580	1,2753
Min (s)	84,6790	80,3653
Max (s)	97,3249	84,1174
Gain (%)	6,9454	
T-Test	0,0006	

The two data sets are statistically different with a t-test = 0. 0006 < 0.01.

4.6.1.2 Comparison with previous work

There is, in the literature, earlier systems with some of the same capabilities. Studies from Matni et al. [25] and Waly et al. [37] used a stateful approach to detect problematic behaviors within traces. This will be used here as a basis for comparison. We described, with our declarative language, the same patterns they used in their performance analysis. Since our approach uses the

state system and the data are written directly on the disk, the size of the RAM is not really significant.

In [25] the authors described the pattern *fd_checking* that checks if someone tries to read a file that has been already closed. We applied this pattern on traces from 500MB to 2GB and compared the results of our declarative pattern approach with those presented in [25]. Table 4.7 displays the details of the results.

Table 4.7: Comparison with Matni et al. work

Trace size (MiB)	XML (s)	MATNI (s)
500	37	57
1000	79	119
1500	141	166
2000	188	266

Waly et al. [37] used the abstraction scenarios of the file management to test the performance of their work. Table 4.8 compares our results with theirs. Despite a higher number of events, the proposed declarative pattern approach provides better result than those in [37].

Table 4.8: Comparison with Waly et al. work

System calls	XML (s)	WALY (s)
OPEN	13	32
READ	13	36
WRITE	14	37
CLOSE	12	40
Number of events	4602278	4109316

4.6.2 Experiences with the tool

In this section, we provide two complete case studies. The first case is a system performance test. We try to find the origin of a web request latency. We will describe a set of steps, using a pattern matching based analysis, to find the source of the problem. The second example is a security attack test case. We try to detect the presence of a SYN flood attack within the trace. We will present the state machines describing the attack pattern and we will apply them to a problematic trace.

4.6.2.1 Unusual I/0 latency issue in a web request

One of the main goals of tracing is to find the root cause of problems. Today's systems can have performance issues that are difficult to understand using conventional tools. Tracing in many cases is the only viable solution. In this use case, a PHP web request issue is simulated and we try to use tracing to discover what really happened. A similar analysis of this case was presented by Julien Desfossez in the LTTng Blog[2].

Experiment description and setup

This test case is a PHP image board (tinyib-mappy). This experience aims to understand why some client requests take much more time than others. A problem is injected for the test purpose by manually issuing a *sync* call, something that different system applications may actually generate in many real situations. The *sync* call flushes the dirty buffers on disk.

On the server, a trace has been recorded using the LTTng tracer. On the client, a series of requests were made and their latency measured. Figure 4.8, shows the latency of the requests. We can see that one request took much more time (around 430ms more) than the others. Why?

[2] http://lttng.org/blog/2015/02/04/web-request-latency-root-cause/

```
$ while true; do time wget -q -p http://10.2.0.1/; sleep 0.5; done
...
real    0m0.017s
real    0m0.016s
real    0m0.454s
real    0m0.016s
real    0m0.015s
...
```

Figure 4.8: Latency of the client requests

Analysis and result

To understand the source of that request latency, we start our analysis focusing on the I/O latency. First, we designed a pattern that finds all the system calls within the trace and generated a high-level synthetic event to represent them. Figure 4.9 displays the state machine described by the pattern to abstract the system calls.

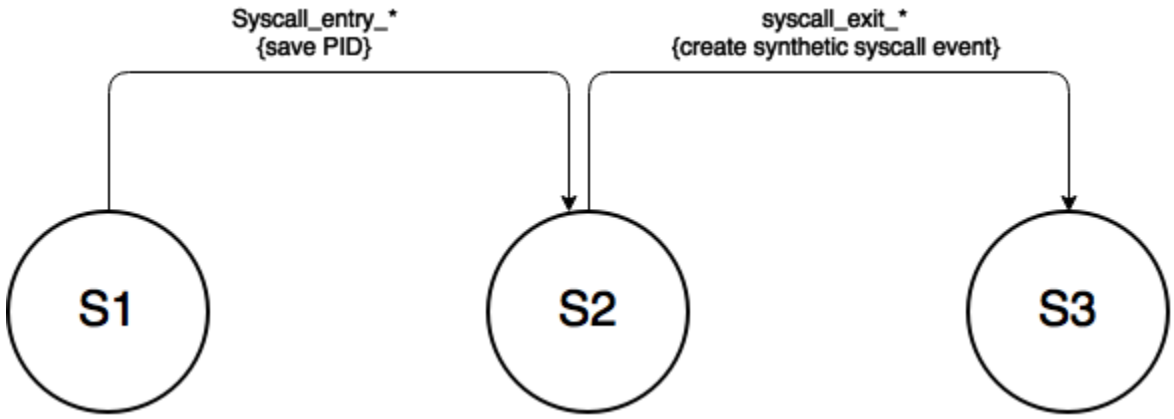
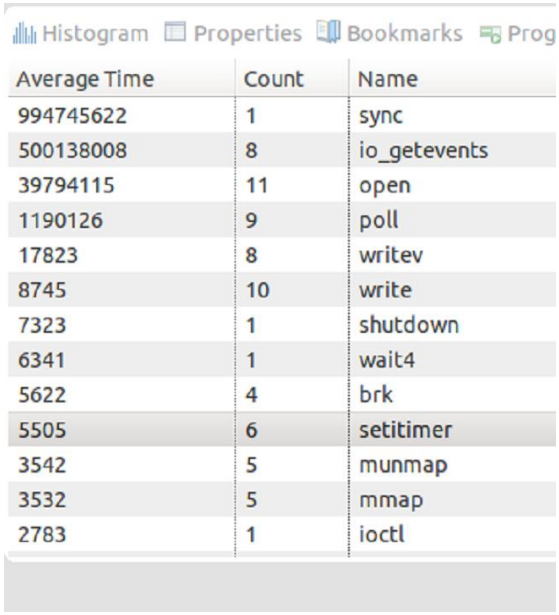


Figure 4.9: system call generic pattern

Since each synthetic event has a duration, we have developed a new tabular view, called ‘Latency View’ that displays the average latency for each generated synthetic events. Figure 4.10 presents the system calls average latency sorted by maximum average of time in nanoseconds.



Average Time	Count	Name
994745622	1	sync
500138008	8	io_getevents
39794115	11	open
1190126	9	poll
17823	8	writev
8745	10	write
7323	1	shutdown
6341	1	wait4
5622	4	brk
5505	6	setitimer
3542	5	munmap
3532	5	mmap
2783	1	ioctl

Figure 4.10: System calls average time in nanoseconds

The first four entries (sync, io_getevents, open, poll) have average times well above the others, prompting further scrutiny. We modified the previous pattern to generate, for each system call, an attribute *'status'* that specifies whether the system call is being executed or not. We added the description of an XML time graph view to display the generated status of the system calls. Figure 4.11 displays the resulting time graph. If we filter the view to only display the four entries identified earlier (Figure 4.12), we can focus on their analysis.

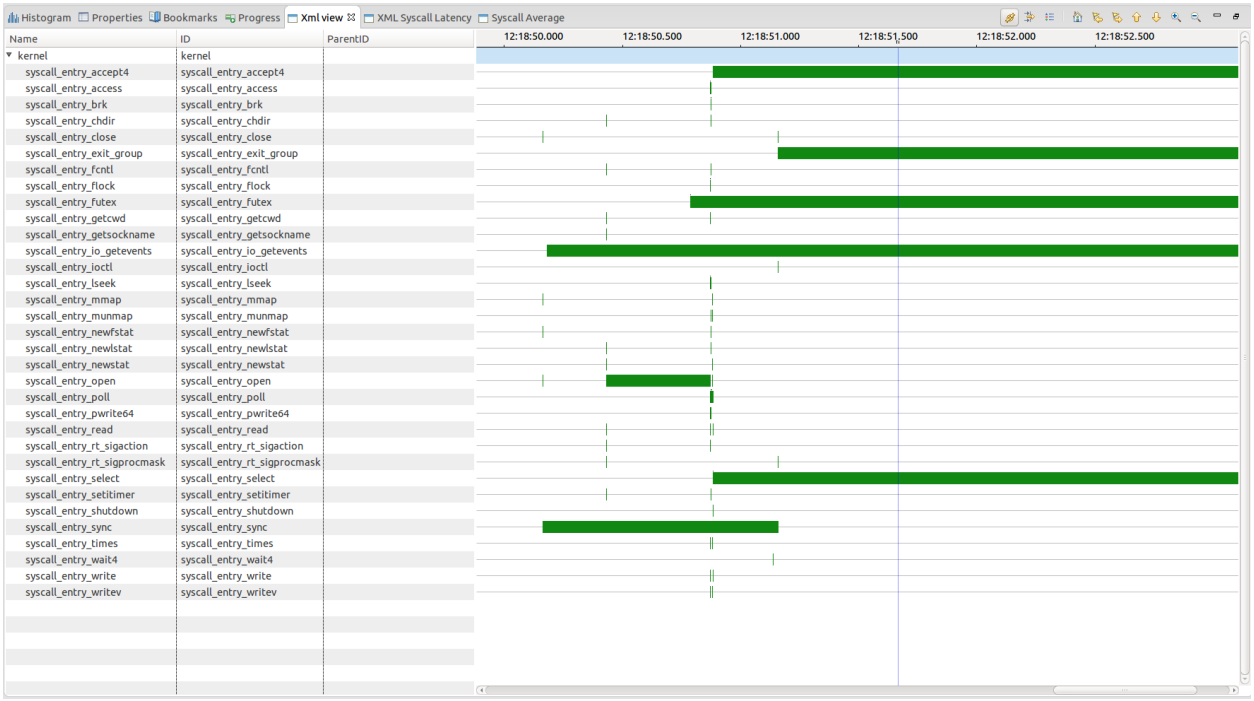


Figure 4.11: Time graph of the status of each system calls

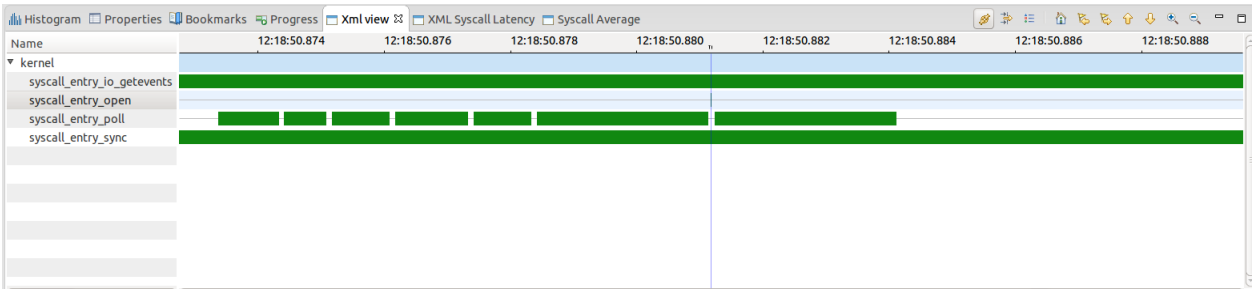


Figure 4.12: Time graph of the status of system calls with big average latency

The entry times of the *io_getevents* calls and the *poll* calls are evenly distributed. There is nothing alarming here. Also, Figure 4.10 shows that there is only one *sync* call within the trace. On the other hand, if we zoom on the *open* call entries (Figure 4.13), we can see that there is one entry that is way longer than the others. Moreover, if we look at the events table view of Trace Compass, we can clearly see that the file opened is a PHP session file, probably from our request.

Thus, the server took around 437ms to open a PHP session file, whereas it took around 20ms to open other files. Now, we need to understand why this latency occurred.

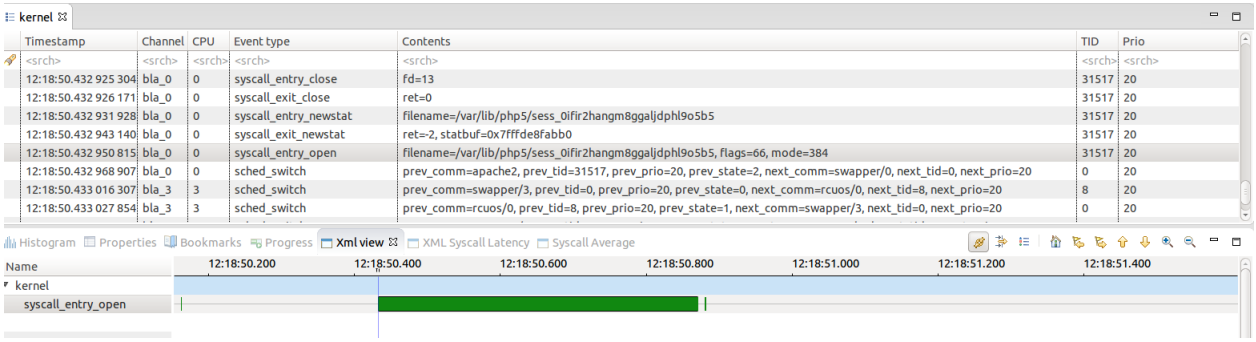


Figure 4.13: Time graph of the status of the open system call

We can then focus our analysis around this call and filter the view displayed in Figure 4.4 to only show I/O calls. Figure 4.14 shows the resulting view. We can clearly see that the *sync* call, found in Figure 4.10, is running at the same time as our problematic *open* call.

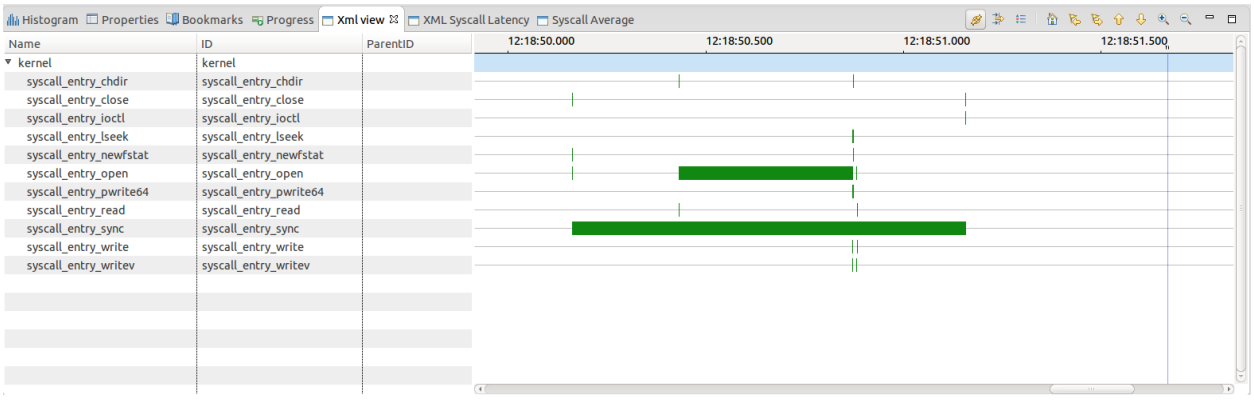


Figure 4.14: Time graph of the I/O calls

Summary

The *sync* call causes a high I/O load. At the same time, the client sends a PHP request. PHP tries to open a temporary session file on disk. The request took more time than usual because of the current disk activity.

This test case is an example of how a system performance issue is identified using the pattern matching based analysis. The files used to analyze this inefficient I/O case are accessible from our public web page[3]. The same kind of analysis has been applied to IRQs in order to understand their latency issues. This example clearly shows how our tool can be use to discover performance issues using customs analyses.

4.6.2.2 Security attack : SYN flood detection

The SYN flood attack is a well-known attack in computer network security. It's a typical form of denial-of-service attack that our tool should be able to detect. In this case study, we will describe how our tool could be used to detect such kind of attacks.

Experiment description and setup

A system is under attack when it is being flooded with connection attempts from the same user. The detection of this attack works in 2 steps:

- Half-open TCP connections: The detection of the failed connection attempts.
- Counting the failed attempts until it reaches a certain threshold is the second step.

Each step represents a pattern that is be described with our declarative language. The sub-steps for an half-open TCP connection are:

- The client sends a request for connection. The SYN flag of the TCP header is then set to 1.

[3] http://secretaire.dorsal.polymtl.ca/ jckouame/xml files/

- The server responds with a SYN and ACK in the TCP header flags. The acknowledgement sequence number of the server is equal to the sequence number of the client plus 1.
- The pattern is matched when a timeout is generated because the client never answers to the server with an ACK, or if the client tries to reset the connection with the server by setting the RST flag to 1 and using the acknowledgement sequence number of the server in the previous step as its sequence number.

Figure 4.15 displays the states of the pattern for the detection of a half-open TCP connections. We used the flags and the value of the sequence numbers as conditions for transitions in the pattern description. A synthetic event is produced for each half-open TCP connection found. A timeout of three seconds is used.

The second pattern counts the number of generated synthetic events. When the number of half-open TCP connections reaches the predefined threshold, the system is under attack. We generate a ”SYN flood attack” synthetic event at this moment. For our testing, we used a threshold of 100 attempts.

We specify in the pattern description file that both patterns should be coexisting. It is possible to have several half-open TCP connection patterns at the same moment but only one counter is allowed.

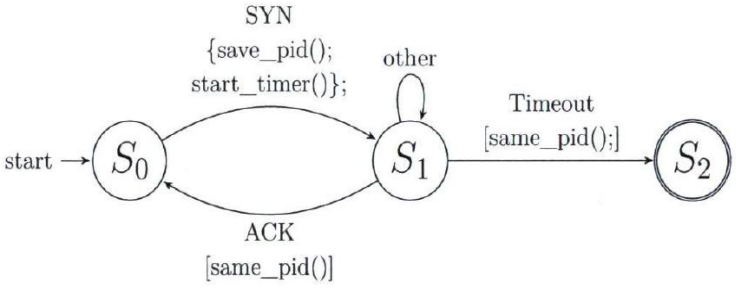


Figure 4.15: Half-open TCP pattern

source[37]

The hping3[4] tool is used to simulate a SYN flood attack on the system. We have recorded the system data with LTTng 2.4.0 and we have enabled the recording of the *inet* events in addition to all the kernel events. We sent requests to an Apache server running on Ubuntu SPM 14.10 (running kernel version 3.13.0-43). We instrumented an Intel core i7 with 8 GB of RAM. We traced the system for a small duration, but long enough to have a lot of half-open TCP connections.

Analysis and result

The first pattern has generated 569 half-open TCP synthetic events and the second raised one SYN flood synthetic event when the counter reached 100. Figure 4.16 shows the results of the analysis. We can see, in the synthetic events table, all the abstract events generated. The SYN flood attack event generated reflects, in the other views of Trace Compass, when the attack starts and when the system was declared to be under attack.

The whole pattern description file of this scenario is accessible from our public web page[5]. A full video of this work can be found on the internet[6].

[4] http://www.hping.org/hping3.html

[5] http://secretaire.dorsal.polymtl.ca/ jckouame/xml files/

[6] A full video demo of this work can be found here: https://www.youtube.com/watch?v=ghBHqhq8LXI.

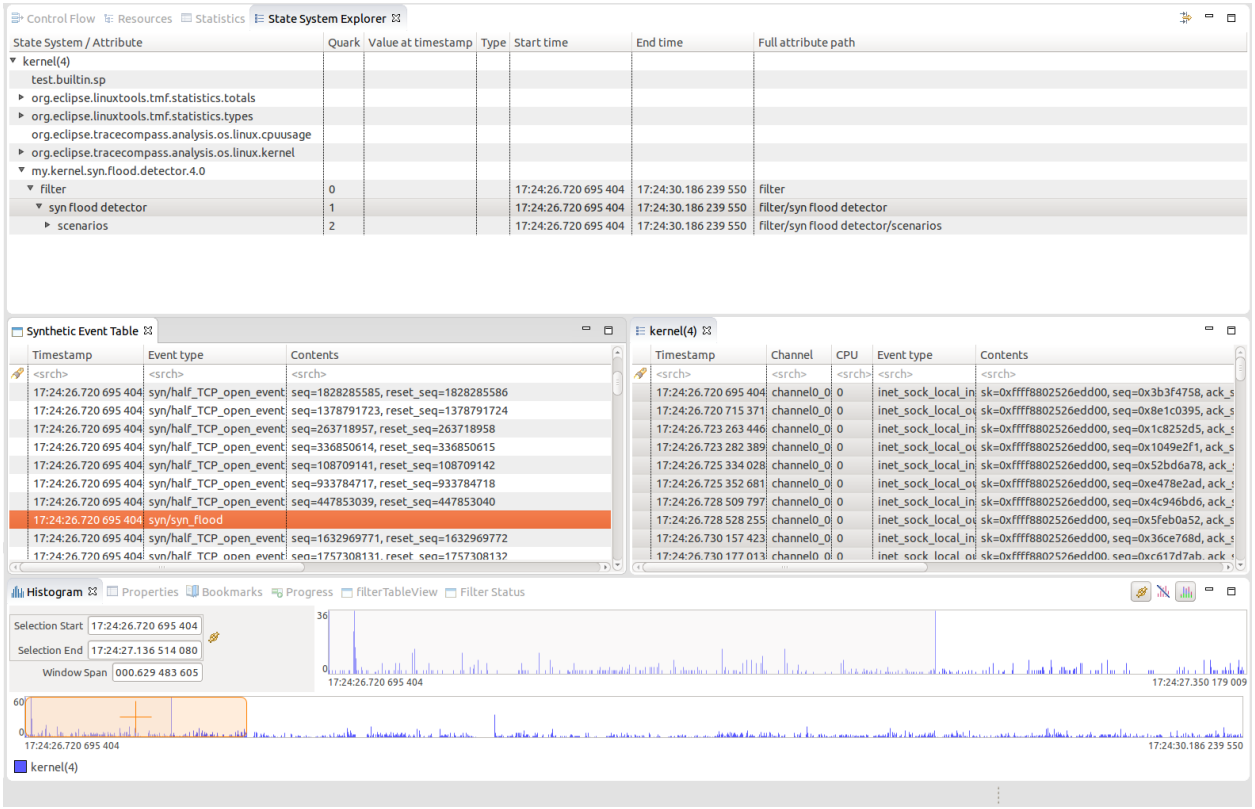


Figure 4.16: Synthetic events generated from the SYN flood attack detection patterns

Summary

This test case is an example of security attack detection using the proposed approach. We successfully detected the attack. The SYN flood attack event generated can be used to produce a state in the system or to raise an alert into Trace Compass. Our tool was successfully able to describe, detect and represent a security attack in the system. The generated 'SYN flood attack' event is synchronised with the others analyses of Trace Compass.

4.7 Conclusion and future work

In this paper, we presented a data-driven approach to match generic filtering patterns within the trace. Our proposal is a pattern matching technique that helps reducing the difficulties related to the large amount of data within the trace. The described tools process large trace files efficiently.

We used the state system to model the system, and the state history as container to store the data. This offers interesting capabilities such as debugging, playing forward/backward the patterns, and sharing the data.

The proposed solution provides a declarative XML-based pattern description language and user-friendly analysis that reflects the matched patterns. The simplicity of the modeling language eases both the description and the understanding of the patterns. It is a flexible approach where the users have the possibility to describe their own custom patterns outside of the visualizer environment.

Our proposed language is based on some simple key concepts. Patterns for several domain applications can be described, and the tool can process data from any trace format. Synthetic events can be generated and are used to locate the patterns in the analysis. They can also provide details about the matched patterns. Moreover, timeouts and maximum duration conditions can be described and processed.

This study completes previous research [13, 15, 25, 37]. Moreover, the performance of our engine shows better results than some of them while offering enhanced functionality.

The performance could be further enhanced by carefully choosing to have some pattern data being temporary, instead of being stored in the state system. This solution could reduce the time spent in the requests for storing and reading the data. This work would also benefit from a user interface to interact with the XML files. Work is underway to graphically specify the patterns and FSM represented in the XML files. This new level of abstraction could further facilitate the description of the patterns and increase the flexibility of the tool.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Analyses complémentaires

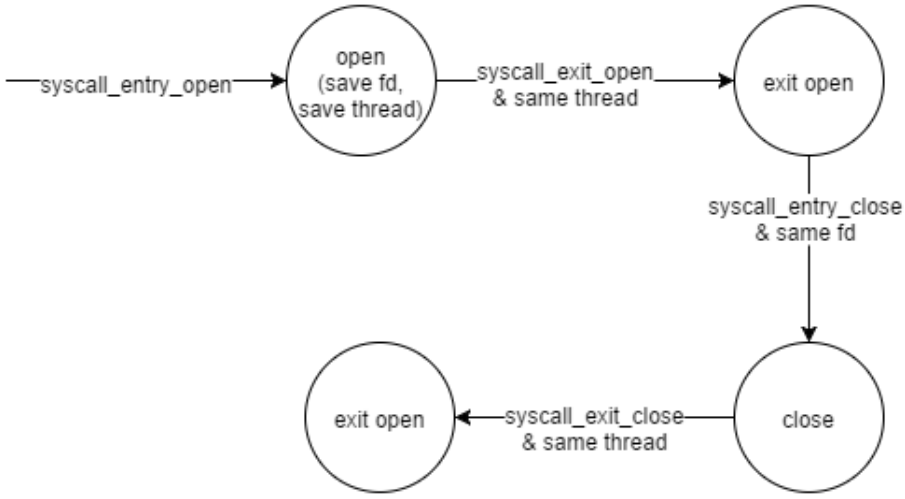
5.1.1 Comparaison des approches JAVA et XML

Dans le chapitre 4, nous avons comparé notre solution avec l'approche des analyses décrite avec le langage JAVA directement dans l'environnement de Trace Compass. Pour ce faire, nous avons appliqué un patron standard où les deux approches enregistrent leurs données dans l'arbre à historique; et donc directement sur le disque. Dans cette section, nous effectuons une comparaison de notre solution avec une analyse de Trace Compass qui ne sauvegarde pas ses informations sur le disque, mais plutôt en mémoire. Nous conservons la configuration des tests effectués dans le chapitre 4. Cette configuration correspond aux caractéristiques standards des machines et des outils que nous utilisons au moment de nos tests.

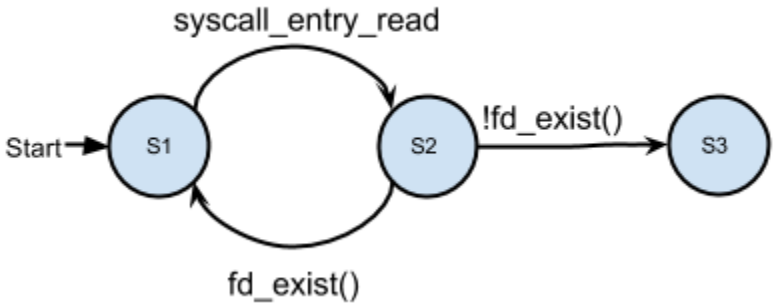
Nous appliquons le patron "*fd_checking*" qui est un patron beaucoup plus complexe que celui utilisé au chapitre 4. Ce patron utilise deux machines à états. La figure 5.1 présente les machines à états nécessaires pour ce patron. La première machine à états correspond au patron d'accès de fichier utilisé dans les tests au chapitre 4. Cependant pour chaque fichier accédé, nous enregistrons le descripteur de fichier *fd*. La seconde machine à état vérifie si une tentative de lecture de fichier est effectuée sur un fichier qui n'est pas ouvert.

Les tableaux 5.1, 5.2 et 5.3 présentent les résultats pour cette analyse. Comme attendu, nous avons un ralentissement de la performance qui décroit linéairement proportionnelle à la taille de la trace.

Le patron décrit avec JAVA sauvegarde très peu de données sur le disque. Notre approche enregistre beaucoup plus de données sur le disque ce qui explique la perte de performance.



(a) Premier patron : accès à un fichier



(b) Second patron : lecture de fichier non ouvert

Figure 5.1: Patron fd_checking

Tableau 5.1: Temps de traitement pour le patron fd_checking pour une trace noyau de 500 MiB

Trace 500 MiB	JAVA	XML
Temps Moyen (s)	30,8270	32,9270
Déviation Standard (s)	0,8908	0,8908
Min (s)	28,6103	30,7103
Max (s)	31,6477	33,7477
Ralentissement (%)	6,8122	
T-Test	0.0001	

Les deux ensembles de données sont statistiquement différents avec un t-test = 0.0001 < 0.01.

Tableau 5.2: Temps de traitement pour le patron fd_checking pour une trace noyau de 1 GiB

Trace 1 GiB	JAVA	XML
Temps Moyen (s)	64,7367	71,1899
Déviation Standard (s)	1,8708	0,8778
Min (s)	60,0818	69,7911
Max (s)	66,4603	72,665
Ralentissement (%)	9,9683	
T-Test	0,0000	

Les deux ensembles de données sont statistiquement différents avec un t-test = 0 < 0.01.

Tableau 5.3: Temps de traitement pour le patron fd_checking pour une trace noyau de 1.5 GiB

Trace 1.5 GiB	JAVA	XML
Temps Moyen (s)	103,5099	117,9828
Déviation Standard (s)	1,5944	4,7876
Min (s)	101,8667	114,0857
Max (s)	105,7866	125,499
Ralentissement (%)	13,9821	
T-Test	0,0006	

Les deux ensembles de données sont statistiquement différents avec un t-test = 0 < 0.01.

5.1.2 Comparaison avec les travaux précédents

Les tests que nous effectuons au chapitre 4 (tableaux 4.7 et 4.8) révèlent 1 gain de performance de la solution que nous proposons par rapport aux travaux précédents possédant des capacités semblables. Ce gain de performance est essentiellement lié à l'utilisation des requêtes optimisées, développées par Montplaisir et al [26], dans l'arbre à historique. Cependant, les conditions dans lesquelles ces tests ont été effectués diffèrent (capacité des machines, nombre d'événements).

5.1.3 Description des patrons

Pour décrire un patron, l'utilisateur devrait commencer par décrire les machines à états qui constituent le patron. Il doit connaitre le fonctionnement de chacune des machines à états. Pour chaque état de chacune des machines à états (FSM), l'utilisateur fait référence à des transitions et

des actions qui permettent de définir le comportement de la machine à état et d'enregistrer les données de la FSM. Les transitions et les actions doivent être définies et décrites plus haut dans le fichier XML, au fur et à mesure que l'utilisateur conçoit sa machine à état. L'exemple suivant est la description complète de la FSM générique pour abstraire tous les appels systèmes :

```
<fsm id="syscall">
<precondition input="tid_18985"/>

<stateTable>
<stateDefinition name="start">
<transition input="syscall_entry_x" next="syscall_entry_x" action="save_data">
<transition input="#other" next="start"/>
</stateDefinition>

<stateDefinition name="syscall_entry_x">
<transition input="syscall_exit_x:same_thread" next="end" action="generate_synthetic_ev
ent/>
<transition input="#other" next="syscall_entry_x"/>
</stateDefinition>

<stateDefinition name="end">
<transition input="#other" next="end"/>
</stateDefinition>
</stateTable>

<initialState id="wait_syscall_entry_x"/>
<endState id="syscall_exit_x"/>
</fsm>
```

La figure 5.2 décrit la machine à états illustrée. Pour l'état *start*, par exemple, l'utilisateur doit décrire, dans le fichier, la transition *syscall_entry_x* et l'action *save_data* pour que l'analyseur de trace puisse les utiliser au moment de l'analyse. Leur description est comme suit :

```
<transitionInput id="syscall_entry_x">
<event eventName="syscall_entry_*"/>
</transitionInput>

<action id="save_data">
<stateChange>
<stateAttribute type="location" value="CurrentThread"/>
<stateAttribute type="constant" value="syscall"/>
<stateAttribute type="constant" value="name"/>
<stateValue type="eventName"/>
</stateChange>
<stateChange>
<stateAttribute type="location" value="CurrentScenario"/>
<stateAttribute type="constant" value="cpu"/>
<stateValue type="eventField" value="cpu"/>
</stateChange>
<stateChange>
<stateAttribute type="location" value="CurrentScenario"/>
<stateAttribute type="constant" value="thread"/>
<stateValue type="query">
<stateAttribute type="location" value="CurrentCPU"/>
<stateAttribute type="constant" value="Current_thread"/>
```

```
</stateValue>
</stateChange>
</action>
```

L'utilisateur doit répéter ces étapes pour chacune des machines à états du patron jusqu'à ce que la description de ce dernier soit complète. Au cours de notre étude, nous avons trouvé dans la littérature un ensemble de problème intéressant que nous avons décrit à l'aide du langage déclaratif proposé. Ces patrons pourraient servir d'exemple de base aux utilisateurs afin qu'ils puissent décrire leurs propres patrons. Ils sont disponible sur internet[7].

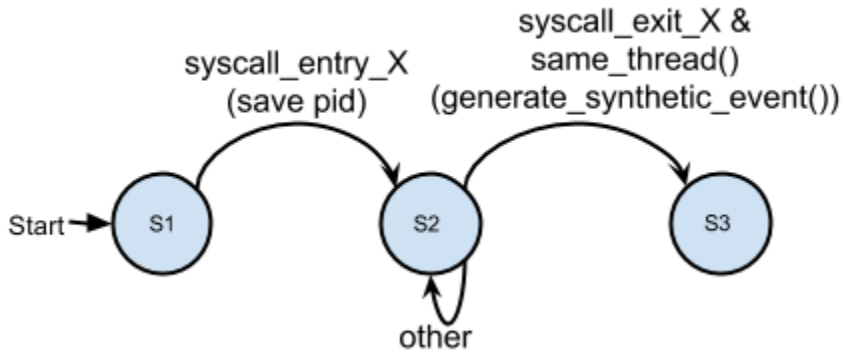


Figure 5.2: Patron générique pour les appels système

5.2 Analyse des facteurs qui affectent la performance

Dans le chapitre 4, nous avons comparé notre approche avec les analyses régulières de Trace Compass ainsi qu'avec quelques travaux antérieurs au nôtre. Dans cette section, nous apportons plus de détails sur les critères qui ont un impact sur la performance de notre approche et comment nous avons tenté de résoudre ces problèmes.

5.2.1 Quantité de données

La quantité de données a un impact très important sur la qualité de notre solution. En effet, les patrons utilisent les informations contenues dans les événements pour décrire des scénarios. Plus

[7] https://github.com/jckouame/xml_files

il y a d'événements pertinents dans la trace, plus il y a de chance d'avoir des patrons concurrents. Plus il y a de patrons concurrents plus le traitement de la trace prend du temps. Dans notre solution, nous avons rajouté la possibilité à chaque état (*e)* d'un FSM de préciser à ce dernier quels sont les événements qui sont pertinents pour lui à ce stade afin d'éviter de réduire la portée du problème.

5.2.2 Données de débogage

Les données de débogage sont les données internes que le système conserve pour suivre l'évolution des patrons. Ces données peuvent elles-mêmes être utilisées dans le processus de filtrage de la trace. Le nombre de requêtes au système d'état peut donc rapidement grimper dépendamment de la complexité du patron et du nombre de patrons en concurrence. Pour pallier à ce problème, nous avons créé un mode de fonctionnement de débogage qui doit être activé dans la déclaration du patron lorsque l'on souhaite enregistrer ces informations. Dans le cas contraire, elles ne sont pas enregistrées. Cette méthode a permis d'accélérer l'analyse de la trace de 50%.

5.2.3 Description des patrons

La description des patrons elle-même a un impact sur la solution. Premièrement, l'utilisateur qui décrit un patron très complexe pourrait être confus dans sa propre conception. Ensuite, le nombre d'états des FSMs ainsi que la complexité des conditions sont une source de ralentissement quand la trace est très large. La vue *Filter Status* décrite dans le chapitre 4 permet à l'utilisateur de suivre l'exécution des patrons et par la même occasion de vérifier s'il se comporte comme ils le devraient.

5.3 Avantages et limites de la solution

Dans cette section, nous listons les avantages et les désavantages de notre solution.

5.3.1 Avantages

5.3.1.1 Au niveau de la description des patrons

L'utilisation de machines à états permet de décrire efficacement des patrons complexes mettant en jeu plusieurs ressources.et informations telles que les estampilles de temps, les fils d'exécution. Ensuite, l'utilisation du langage XML apporte une certaine flexibilité à la solution. Les utilisateurs peuvent décrire des patrons personnalisés sans se soucier de l'implémentation de la visionneuse. De plus, le langage XML étant facilement extensible, la spécification du langage déclaratif proposé pourrait évoluer afin de décrire de nouveaux types de patrons; pas encore rencontrés dans notre étude.

5.3.1.2 Au niveau des fonctionnalités des patrons

La spécification des actions que peuvent exécuter les patrons offre la possibilité de démarrer une instance d'un patron. Ainsi, il est possible que chaque patron puisse en activer un autre. Le système supporte également la présence de plusieurs instances d'un même patron simultanément ainsi que la concurrence entre les patrons indépendants. De plus, exploiter la structure de l'arbre d'attributs du système d'états offre la possibilité aux patrons de partager des informations. Cela permet, par exemple, de maintenir des compteurs et d'obtenir des données statistiques.

5.3.1.3 Au niveau de la visualisation

Notre solution propose une vue tabulaire pour la représentation des patrons ainsi qu'une vue en forme de diagramme de Gantt pour suivre leur évolution. Ces deux vues sont synchronisées avec les autres analyses dans l'environnement de Trace Compass de façon à pouvoir repérer clairement où les patrons se situent dans la trace. Notre approche permet également de générer des événements synthétiques qui sont des événements de haut niveau. Ces événements peuvent être traités par l'analyseur au fur et à mesure qu'ils sont créés. Une vue tabulaire permet également de les représenter et de les synchroniser avec les autres analyses de Trace Compass.

5.3.2 Limites

5.3.2.1 Conditions basées sur le temps

En mode normal, sans débogage, l'enregistrement des informations internes sur l'exécution des patrons dans le temps est désactivé. La possibilité de spécifier des conditions basées sur le temps dans les patrons est par la même occasion altérée. Il est toujours possible de comparer l'estampille de temps *t* d'un événement à une valeur constante *c*. Cependant, il n'est plus possible de se repérer dans le patron grâce à l'estampille de temps t_1 d'un état e_1.

5.3.2.2 Analyse en temps réel

La version actuelle de la solution ne permet pas d'effectuer la recherche de patrons au sein des traces en temps réel. Elle a été développée pour le mode d'analyse différé.

5.3.2.3 Étude comparative

Notre solution semble moins performante que l'approche avec JAVA lorsque la quantité de données enregistrée dans l'arbre à historique est énorme. (Tableaux 5.1, 5.2, 5.3). De même, nous n'avons pas réussi à reproduire entièrement les mêmes conditions de test que les travaux antérieurs auxquels nous comparons notre étude. Nous ne disposons pas des mêmes spécifications de machines et des mêmes versions des traceurs. Ainsi, il est difficile de valider sans aucun doute possible les résultats de notre étude comparative.

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS

Ce chapitre conclut le mémoire en effectuant une synthèse de nos travaux de recherche. Nous présentons également les limitations de notre solution et effectuons quelques recommandations pour des améliorations futures.

6.1 Synthèse des travaux

Les travaux de recherches présentés dans ce document ont porté sur la détection de patrons à l'intérieur de traces volumineuses. L'objectif est d'évaluer la possibilité de concevoir des patrons personnalisés pour des analyses automatisés au sein de traces d'exécution et de les représenter efficacement. La solution proposée réalise avec succès cet objectif. Nous avons présenté dans ce mémoire deux exemples concrets de l'utilité de la solution : une tentative de découverte de la source d'un problème de latence dans une requête web et un exemple de détection d'une attaque dans le système.

Nous avons atteint quatre de nos cinq objectifs spécifiques. (1) Nous avons élaboré un langage à automate, de manière déclarative en XML, pour la description des patrons. (2) Nous avons développé un analyseur de traces capable de modéliser les patrons décrits et de filtrer la trace à partir des modèles créés. (3) Nous avons élaboré une vue sous forme de diagramme de Gantt qui permet de suivre l'évolution des patrons dans la trace. (4) Nous avons conçu des vues qui résument l'exécution des patrons et montrent leur position dans la trace; dans le but de représenter de façon efficace les patrons repérés dans la trace. Le cinquième objectif n'a pas pu être validé entièrement. En effet, notre solution pourrait être moins performante que les analyses conventionnelles de Trace Compass dans certains cas. De même, il est difficile de valider qu'elle reste néanmoins aussi performante que les travaux antérieurs qui lui sont semblables.

6.2 Limitation de la solution proposée

Une fonctionnalité importante du langage de modélisation proposé est la possibilité de spécifier des conditions basées sur le temps. En effet, il est possible par exemple de vérifier qu'un événement *e* soit arrivé avant/après un temps *t* ou à l'intérieur d'un intervalle *delta* ou dans un délai de temps *dt* par rapport à un état du patron. Cependant, pour que cette fonctionnalité soit active, le système a besoin d'enregistrer constamment les données sur l'avancée de l'exécution des

patrons. De plus, les données enregistrées sont elles-mêmes utilisées dans le processus d'analyse de la trace. Vu que les données sont enregistrées directement dans le système d'état, le nombre de requêtes de sauvegarde et de lecture des données croît rapidement. Le nombre élevé de requêtes est une source de ralentissement et constitue une limitation à notre solution.

Une autre limitation de nos travaux est que l'analyse des traces en temps réel n'est pas encore supportée. Les travaux réalisés dans cette recherche se sont uniquement concentrés sur des traces en mode hors-ligne.

6.3 Recommandations

Dans cette section, nous proposons un ensemble d'améliorations futures à notre solution.

Dans le chapitre précédent, nous avons évoqué le problème de performance lié au nombre élevé de requêtes dans le système d'état. Il pourrait être intéressant de développer un mécanisme de gestion de variables temporaires. Les variables temporaires seront des valeurs transitoires, générées par le patron et dont la sauvegarde est réalisée quelque part autre que dans le système d'état de façon à faciliter leurs accès. Par exemple, si un patron possède un compteur, la valeur du compteur peut être sauvegardée en mémoire et seulement écrite dans l'arbre à historique à la fin du scénario si nécessaire. Cela éviterait de lire et écrire continuellement, sur le disque, la valeur du compteur.

La description des patrons devrait être encore plus simple en introduisant un nouveau niveau d'abstraction qui pourrait permettre de décrire les patrons graphiquement. Une interface graphique semblable au diagramme UML par exemple, pourrait être élaborée. La description des patrons avec cette interface graphique devrait générer automatiquement une description du patron avec le langage déclaratif proposé dans notre solution. C'est cette description qui sera utilisée par l'analyseur.

BIBLIOGRAPHIE

[1] Habra, N., Le Charlier, B., Mounji, A., & Mathieu, I. (1992). ASAX: Software architecture and rule-based language for universal audit trail analysis. Dans *Computer Security—ESORICS 92* (pp. 435-450). Springer Berlin Heidelberg.

[2] Alawneh, L., & Hamou-Lhadj, A. (2011, March). Pattern recognition techniques applied to the abstraction of traces of inter-process communication. Dans *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*(pp. 211-220). IEEE.

[3] Befrouei, M. T., Wang, C., & Weissenbacher, G. (2014, January). Abstraction and mining of traces to explain concurrency bugs. Dans *Runtime Verification* (pp. 162-177). Springer International Publishing.

[4] Cantrill, B., Shapiro, M. W., & Leventhal, A. H. (2004, June). Dynamic Instrumentation of Production Systems. Dans *USENIX Annual Technical Conference, General Track* (pp. 15-28).

[5] Chan, A., Gropp, W., & Lusk, E. (2008). An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, *16*(2-3), 155-165.

[6] Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., & Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, *35*(5), 684-702.

[7] Desfossez, J. (2015). Finding the Root Cause of a Web Request Latency — LTTng. Consulté le 21 juin 2015, Disponible à http://www.lttng.org/blog/2015/02/04/web-request-latency-root-cause/

[8] Dindar, N., Güç, B., Lau, P., Ozal, A., Soner, M., & Tatbul, N. (2009, June). Dejavu: declarative pattern matching over live and archived streams of events. Dans *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (pp. 1023-1026). ACM.

[9] Dtrace, 2011. disponible à http://wikis.stm.com/display/DTrace/Documentation.

[10] 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), 71-104.

[11] Edge, J. (2009). Perfcounters added to the mainline. disponible à http://lwn.net/Articles/339361/

[12] EfficiOS. Common trace format (ctf) specifications, 2015, disponible à http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md;hb=master.

[13] Ezzati Jivan, N. (2014). *Multi-Level Trace Abstraction, Linking and Display* (Doctoral dissertation, École Polytechnique de Montréal).

[14] Ezzati-Jivan, N., & Dagenais, M. R. (2012). A stateful approach to generate synthetic events from kernel traces. *Advances in Software Engineering*, *2012*, 6.

[15] Fadel, W., & Hamou-Lhadj, A. (2010). Techniques for the abstraction of system call traces. *Master's thesis, Concordia University*.

[16] Fahem, R. (2012). *Points de trace statiques et dynamiques en mode noyau* (Doctoral dissertation, École Polytechnique de Montréal).

[17] Eigler, F. C., & Hat, R. (2006, July). Problem solving with systemtap. Dans *Proc. of the Ottawa Linux Symposium* (pp. 261-268).

[18] Han, J., Cheng, H., Xin, D., & Yan, X. (2007). Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, *15*(1), 55-86.

[19] Hamou-Lhadj, A. (2007, June). Effective exploration and visualization of large execution traces. Dans *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on* (pp. 152-153). IEEE.

[20] Hamou-Lhadj, A., & Lethbridge, T. C. (2002). Compression techniques to simplify the analysis of large execution traces. Dans *Program Comprehension, 2002. Proceedings. 10th International Workshop on* (pp. 159-168). IEEE.

[21] Jerding, D., & Rugaber, S. (1997, October). Using visualization for architectural localization and extraction. Dans *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on* (pp. 56-65). IEEE.

[22] LaRosa, C., Xiong, L., & Mandelberg, K. (2008, March). Frequent pattern mining for kernel trace data. Dans *Proceedings of the 2008 ACM symposium on Applied computing* (pp. 880-885). ACM.

[23] LTTng, disponible à http://lttng.org/

[24] Desnoyers, M., & Dagenais, M. R. (2006, July). The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. Dans *OLS (Ottawa Linux Symposium)* (Vol. 2006, pp. 209-224).

[25] Matni, G. (2009). *Detecting Problematic Execution Patterns Through Automatic Kernel Trace Analysis* (Doctoral dissertation, École Polytechnique de Montréal).

[26] Montplaisir, A. (2011). *Stockage sur disque pour acces rapide d'attributs avec intervalles de temps*. Memoire de maitrise, Ecole polytechnique de Montreal.

[27] Montplaisir, A., Ezzati-Jivan, N., Wininger, F., & Dagenais, M. (2013, January). Efficient model to query and visualize the system states extracted from trace data. Dans *Runtime Verification* (pp. 219-234). Springer Berlin Heidelberg.

[28] Montplaisir-Gonçalves, A., Ezzati-Jivan, N., Wininger, F., & Dagenais, M. R. (2013, September). State history tree: an incremental disk-based data structure for very large interval data. Dans *Social Computing (SocialCom), 2013 International Conference on* (pp. 716-724). IEEE.

[29] Noda, K., Kobayashi, T., & Agusa, K. (2012, October). Execution trace abstraction based on meta patterns usage. Dans *Reverse Engineering (WCRE), 2012 19th Working Conference on* (pp. 167-176). IEEE.

[30] Pirzadeh, H., Hamou-Lhadj, A., & Shah, M. (2011, September). Exploiting text mining techniques in the analysis of execution traces. Dans *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* (pp. 223-232). IEEE.

[31] "Rfc 793: Transmission control protocol," May 2011.

[32] Rostedt, S. (2009). Finding origins of latencies using ftrace. *Proc. RT Linux WS*.

[33] RT Patch, disponible à http://www.kernel.org/pub/linux/kernel/projects/rt/, http://rt.wiki.kernel.org/index.php/Main_Page

[34] Schnorr, L. M., Huard, G., & Navaux, P. O. A. (2009, May). Towards visualization scalability through time intervals and hierarchical organization of monitoring data. Dans *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on* (pp. 428-435). IEEE.

[35] Snort, 2015. Disponible à http://www.snort.org/documents.

[36] Vergé, A. (2014). *Traçage logiciel assisté par matériel* (Doctoral dissertation, École Polytechnique de Montréal).

[37] Waly, H. (2011). *Automated Fault Identification: Kernel Trace Analysis* (Doctoral dissertation, Université Laval).

[38] Wininger, F. (2014). *Conception flexible d'analyses issues d'une trace système* (Doctoral dissertation, École Polytechnique de Montréal).

[39] Wolf, F., Mohr, B., Dongarra, J., & Moore, S. (2004, January). Efficient pattern search in large traces through successive refinement. In *Euro-Par 2004 Parallel Processing* (pp. 47-54). Springer Berlin Heidelberg.