UNIVERSITÉ DE MONTRÉAL

DETECTING PROBLEMATIC EXECUTION PATTERNS THROUGH AUTOMATIC KERNEL TRACE ANALYSIS

GABRIEL MATNI
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)
AVRIL 2009

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

DETECTING PROBLEMATIC EXECUTION PATTERNS THROUGH AUTOMATIC KERNEL TRACE ANALYSIS

présenté par: MATNI Gabriel
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. ROY Robert, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
Mme BOUCHENEB Hanifa, Doctorat, membre

To my mom, Nadia
whose infinite love,
turned me into who I am today...

ACKNOWLEDGEMENT

I would like to thank my beloved parents Nadia and Michel for their endless support, love and care. I thank my sisters Eva and Mirna who were always there to listen and advise.

I would also like to thank my research director, professor Michel Dagenais, for giving me the chance to acquire a very enriching experience in one of the most interesting fields in Computer Engineering. His advices were indispensable for the realization of this project.

Last but not least, I thank my laboratory collegues, Mathieu Desnoyers, Parisa Heidari, Pierre-Marc Founier and Benjamin Poirier for their technical support.

RÉSUMÉ

Les processeurs multicoeurs, les systèmes distribués et la virtualisation deviennent de plus en plus répandus, rendant ainsi le débogage des systèmes en production plus difficile, surtout quand les problèmes rencontrés ne sont pas facilement reproductibles. Cette complexité architecturale a introduit une nouvelle gamme de problèmes potentiels qu'on devrait pouvoir détecter à l'aide de nouvelles méthodologies efficaces et extensibles.

En effet, en traçant le noyau d'un système d'exploitation, on est capable d'identifier les goulots d'étranglement, les failles de sécurité, les bogues de programmation ainsi que d'autres genres de comportements indésirables. Le traçage consiste à collecter les événements pertinents se produisant sur un système en production, tout en ayant un impact minimal sur la performance ainsi que sur le flot normal d'exécution. La trace générée est typiquement inspectée en différé, n'introduisant aucun impact sur le système tracé.

Ce travail présente une nouvelle approche basée sur les automates pour la modélisation de patrons de comportements problématiques sous forme d'une ou plusieurs machines à états finis exécutables. Ces patrons sont ensuite introduits dans un analyseur qui vérifie leur existence simultanément et efficacement dans des traces de plusieurs giga-octets. L'analyseur fournit une interface de programmation offrant des services essentiels aux automates.

Les patrons implémentés touchent à différents domaines incluant la sécurité, le test de logiciels et l'analyse de performance. Les résultats de l'analyse fournissent suffisamment d'information pour identifier précisément la source du problème ce qui nous a permis d'identifier une séquence de code dans le noyau Linux pouvant générer un inter-blocage. La performance de l'analyseur est linéaire par rapport à la taille de la trace. Les autres facteurs affectant sa performance sont discutés. En outre, la comparaison entre la performance de l'analyseur par rapport à celle d'une approche dédiée, suggère que le surcoût de l'utilisation des machines à états pour l'exécution et non seulement pour la modélisa-

tion, est acceptable surtout lors d'une analyse différée.

La solution implémentée est facilement parallélisable et pourrait bien s'appliquer à des analyses en-ligne. Le mémoire se conclut par une liste de suggestions d'optimisations possibles pouvant encore améliorer la performance de l'analyseur.

ABSTRACT

As multi-core processors, distributed systems and virtualization are gaining a larger share in the market, debugging production systems has become a more challenging task, especially when the occurring problems are not easily reproducible. The new architectural complexity introduced a large number of potential problems that need to be detected on live systems with adequate, efficient and scalable methodologies. By tracing the kernel of an operating system, performance bottlnecks, malicious activities, programming bugs and other kinds of problematic behavior could be accurately detected. Tracing consists in monitoring and logging relevant events occurring on live systems with a minimal performance impact and interference with the flow of execution. The generated trace is typically inspected remotely with no overhead on the system whatsoever.

This work presents an automata-based approach for modeling patterns of undesired behavior using executable Finite State Machines. They are fed into an offline analyzer which efficiently and simultaneously checks for their occurrences even in traces of several gigabytes. The analyzer provides an Application Programming Interface offering essential services to the Finite State Machines. To our knowledge, this is the first attempt that relies on describing problematic patterns for kernel trace analysis.

The implemented patterns touch on several fields including security, software testing and performance debugging. The analysis results provide enough information to precisely identify the source of the problem. This was helpful to identify a suspicious code sequence in the Linux kernel that could generate a deadlock.

The analyzer achieves a linear performance with respect to the trace size. The remaining factors impacting its performance are also discussed. The performance of the automata-based approach is compared with that of a dedicated implementation suggesting that the overhead of using Finite State Machines for execution and not just for modeling is acceptable especially in post-mortem analysis.

The implemented solution is highly parallelizable and may be ported for online pattern

matching. The thesis concludes by suggesting a list of possible optimizations that would further improve the analyzer's performance.

CONDENSÉ EN FRANÇAIS

Les systèmes informatiques évoluent très rapidement, et se distinguent par un niveau de complexité supérieur à celui de la génération précédente. Parmi ces nouvelles technologies, la virtualisation, s'exécutant possiblement sur des processeurs multi-coeurs, offre des intérêts de plus en plus visibles, tel que l'utilisation optimale des ressources et la sécurité. Cependant, la complexité architecturale introduite présente une nouvelle gamme de problèmes difficiles à cibler surtout quand ils apparaissent occasionnellement sur des machines en production, et ne sont pas facilement reproductibles. En effet, le premier défi consiste à déterminer à quel niveau le problème s'est produit : au niveau d'un processus s'exécutant par-dessus la machine virtuelle, ou de la machine virtuelle elle-même, ou bien au niveau de la machine physique, pour en citer quelques-uns. Évidemment, les outils de débogage traditionnels ne sont pas adaptés à l'analyse du comportement global du système et visent surtout un seul processus en particulier. Le manque d'outil adéquat a donné naissance au traçage du noyau du système d'exploitation. Le traçage consiste à surveiller et enregistrer les événements pertinents se produisant sur un système en production. Le but principal est de déterminer précisément ce que le système fait à chaque instant.

Traçage du noyau Présentement, de nombreux efforts visent à extraire les événements du noyau tout en ayant un impact minimal sur la performance du système (Bligh et al., 2007), (Cantrill et al., 2004) et (Eckmann et al., 2002). Ces événements, typiquement générés par l'ordonnanceur, les systèmes de fichiers, les protocoles réseau, la communication inter-processus, les appels systèmes etc., donnent une vue globale et détaillée du fonctionnement du système. Le traçage peut être constamment activé pour une longue période de temps, générant ainsi plusieurs milliards d'événements ; il peut aussi être activé et désactivé par intermittence selon les conditions spécifiées par l'utilisateur. Il peut être constamment activé mais sans sauvegarder les données sur le disque ; les

événements sont écrits dans des tampons circulaires en mémoire où les données les plus anciennes sont écrasées par les plus récentes. L'écriture des événements sur le disque ne s'effectue que lorsqu'un problème se produit. Cette approche est particulièrement utile pour investiguer des problèmes se produisant accidentellement sur des machines en production.

Analyse de traces Une fois que la trace est générée, la prochaine étape consiste à l'analyser à l'aide d'outils de base, afin de déduire la source du problème. Les outils les plus populaires constituent les filtres et les visualisateurs de traces. Les filtres permettent d'identifier tous les événements satisfaisant un certain nombre de contraintes. Les visualisateurs tel que le diagramme de Gantt (LTTng, 2009), permettent au développeur de se déplacer à travers la trace afin de déterminer visuellement toutes sortes de comportements inattendus. Malgré l'utilisation de ces outils, l'analyse reste relativement manuelle, inefficace et dans certain cas, impossible. En effet, il serait très difficile et inefficace de détecter visuellement une mauvaise utilisation des verrous ou bien des signes d'attaques de déni de service. Notre travail consiste donc à automatiser cette étape pour aboutir à une analyse de trace plus précise et efficace. La trace est typiquement inspectée en différé, n'introduisant ainsi aucun impact sur le système tracé.

Objectifs et méthodologie Notre objectif consiste à détecter automatiquement la présence d'anomalies dans des traces d'exécution ce qui présente les trois défis suivants :

1. Fournir un moyen simple, flexible et non-ambigu pour représenter des comportements problématique.
2. Recueillir efficacement du noyau les événements pertinents pour l'analyse.
3. À partir d'un seul parcours à travers la trace, valider l'existence des patrons simultanément et efficacement. Ensuite, à la fin de l'analyse, fournir suffisamment d'information ciblant la source du problème.

Pour ce faire, nous avons commencé par collecter un ensemble de scénarios problématique touchant à différents domaines tels que la sécurité, l'analyse de performance et le test de logiciels. Ceci nous a permis de définir les critères de sélection qu'un langage de description de patrons devrait satisfaire. Ensuite, nous avons étudié les principaux langages utilisés dans divers domaines (tels que la détection d'intrusion et le débogage de performance dans les applications réparties), que nous avons regroupés en trois catégories : les langages spécifiques au domaine, les langages génériques et les langages à base d'automates. En se basant sur les critères de sélection établis, nous avons justifié notre choix du langage de description de patrons. L'analyseur ainsi que les différents patrons collectés ont été alors implémentés, tout en ajustant selon les besoins, l'interface de programmation offerte par l'analyseur.

Critères de sélection Les critères de sélection sur lesquels nous nous sommes basés pour choisir le langage de description de patrons le plus convenable se résument comme suit :

– Simplicité et Expressivité : Le langage devrait fournir un ensemble minimal d'opérateurs suffisant pour décrire les divers patrons déjà collectés. En même temps, il devrait être assez expressif pour exprimer des patrons non considérés.
– Aucune ambiguïté : La syntaxe du langage devrait être rigoureusement définie, ne permettant qu'une seule interprétation de la description.
– Indépendance du traceur : Le langage ne devrait pas dépendre du format interne de la trace. En effet, nous voudrions éviter d'utiliser un langage spécifique au domaine pour plusieurs raisons : premièrement, pour éviter le coût de maintenance d'un tel langage. Ensuite, les langages spécifiques au domaine sont généralement plus difficiles à déboguer. Finalement, si le format de la trace varie, c'est uniquement l'analyseur qui devrait le supporter, ainsi tous les patrons déjà définis resteront intacts.
– Support de patrons à plusieurs événements : certains langages fournissent uniquement des opérateurs touchant à un seul événement à la fois. Ceci est à éviter car

les comportements problématiques considérés dépendent très souvent d'une séquence d'événements bien définie.

– Distinction entre une analyse en ligne et hors-ligne : la description des patrons ne doit pas dépendre du mode en ligne ou hors-ligne de l'analyse.

Taxonomie des langages **Les langages spécifiques au domaine** constituent le premier groupe que nous subdivisons en deux catégories comprenant les langages descriptifs et impératifs. Dans la première catégorie, on étudie le langage utilisé par Snort (SNORT, 2009), un système de détection d'intrusion très populaire à base de règles. Dans la deuxième catégorie, on étudie les langages de DTrace (Cantrill et al., 2004) et de SytemTap (Eigler, 2006) qui sont deux traceurs de noyau, permettant avec leurs langages, d'effectuer une analyse en ligne. On étudie aussi RUSSEL (Habra et al.,), un langage de description de règles de sécurité.

Le deuxième groupe comprend **les langages génériques**. On identifie dans ce groupe EXPERT (Wolf et al., 2004), un système de détection de problèmes de performance dans des traces OpenMP et MPI. EXPERT emploie le language générique Python pour décrire les propriétés de performance.

Dans le troisième groupe, on décrit et compare les principaux **langages à base d'automates**. Les langages de ce groupe offrent un haut niveau d'abstraction permettant de définir des événements synthétiques à partir de plusieurs événements primitifs. On identifie dans cette catégorie Ragel (RAGEL, 2009), un langage souvant utilisé dans les analyses syntaxiques. De même, on étudie STATL (Eckmann et al., 2002), un langage qui permet de décrire des scénarios d'attaque sous forme de machines à états finis. Finalement, on étudie le compilateur SMC (Compiler, 2009), un projet accessible et open-source. Ce dernier fournit un langage offrant des fonctionnalités et caractéristiques le rendant très extensible. En effet, le compilateur convertit le langage en 14 langages génériques supportés, et donc le code généré est capable de profiter d'une interface de programmation fournie par un autre langage. Pour ces raisons, nous avons utilisé SMC

pour décrire les patrons capables de s'interfacer avec notre analyseur.
L'utilisation d'un langage à base d'automates pour la description de patrons consiste à modéliser le comportement problématique sous forme d'une machine à états. La présence d'un événement particulier dans la trace va déclencher une transition d'un état vers un autre, si la garde de transition est évaluée à Vrai. À ce moment-là, les actions définies pour cette transition sont exécutées. Les actions servent typiquement à mettre à jour les structures de données locales ou globales, à s'interfacer avec l'analyseur ou à générer des alertes.

Ensemble de patrons considérés Avant d'établir les critères de sélection d'un langage convenable, il était indispensable d'avoir une liste de patrons problématiques et représentatifs, touchant à différents domaines et qu'on regroupe en trois catégories :

Patrons de sécurité

1. **L'attaque SYN flood :** Le SYN flood est une attaque de déni de service qui consiste à inonder un serveur avec un grand nombre de connexions TCP semi-ouvertes. Les indices d'une telle attaque sont visibles dans une trace du noyau si les événements pertinents sont instrumentés. Il est très inefficace de détecter manuellement ce genre de patrons, d'où l'intérêt de notre approche.
2. **Prison chroot :** S'échapper d'une prison chroot est un autre type d'attaque qu'on peut détecter. L'appel système chroot() est parfois utilisé quand un processus privilégié veut restreindre l'accès à un sous-arbre du système de fichier. Si ce processus essaie d'ouvrir un fichier avant d'appeler chdir("/"), alors une faille de sécurité se présente. En effet, l'attaquant peut toujours sortir d'une prison chroot en truquant le programme pour ouvrir le fichier ../../../../etc/shadow par exemple. Restreindre l'accès à un sous-arbre n'est pas accompli par un appel unique à chroot. La bonne façon de procéder serait d'appeler chdir("/") immédiatement après l'appel système chroot().

3. **Les virus Linux :** Malgré qu'ils soient difficiles à trouver, les virus s'exécutant sous Linux existent et peuvent être détectés à l'aide d'une trace d'exécution. L'approche qu'on propose diffère de celles utilisées dans les logiciels anti-virus. Lorsqu'un nouveau virus est découvert, il suffit de l'exécuter sur une machine virtuelle roulant un noyau Linux instrumenté afin d'être capable d'enregistrer toutes ses interactions avec le système d'exploitation. Ces interactions définissent le comportement du virus que nous encodons sous forme de patron afin de détecter sa présence à partir d'une trace d'exécution.

Patrons de test de logiciels

1. **Utilisation des verrous :** Les ressources partagées nécessitent une prise de verrou avant d'y accéder pour éviter les situations de compétition. Dans le noyau Linux, le verrouillage est plus délicat qu'en mode utilisateur à cause des différents états possibles (préemption activée/désactivée, traitement d'IRQ, etc.). Par exemple, un verrou tournant pris quand les interruptions sont activées, ne doit jamais être pris dans un gestionnaire d'interruption. La raison est simple : l'interruption peut arriver à n'importe quel moment, particulièrement lorsque le verrou est déjà pris, inter-bloquant ainsi le processeur correspondant. Il est possible de s'assurer au moment de l'exécution que toute prise de verrou est valide et de signaler celles qui ne le sont pas. Cependant, cette option requiert la recompilation du noyau et ajoute un impact en performance continu au système. Par conséquent, nous encodons un sous-ensemble des règles d'utilisation des verrous sous forme d'une machine à états par CPU, et nous procédons à leur validation dans des traces d'exécutions. Ceci nous a permis d'identifier une séquence de code dans le noyau Linux qui pourrait générer un inter-blocage.
2. **Utilisation des descripteurs de fichiers :** Un autre bogue de programmation détectable à partir d'une trace, consiste à accéder à un descripteur de ficher après l'avoir fermé. L'ouverture, l'accès et la fermeture d'un descripteur de fichier doivent

suivre un ordre bien défini. Ceci représente une plus grande classe d'erreurs de programmation où deux ou plusieurs événements sont logiquement et temporellement reliés.

Patrons de débogage de performance

1. **Inefficacités I/O :** Certaines inefficacités dans les logiciels peuvent être détectées à partir des événements d'entrée et de sortie I/O. Par exemple, un processus qui effectue de fréquentes écritures de très peu de données au disque pourrait considérablement affecter la performance du système. De même, la lecture des données qui viennent d'être écrites, ou bien deux lectures consécutives des mêmes données, ou l'écrasement des données qui viennent d'être écrites, sont toutes des signes d'inefficacité visibles dans une trace du noyau.
2. **Contraintes temporelles :** Les applications multimédia, et plus généralement les applications temps-réel, sont caractérisées par leurs contraintes temporelles. En supposant que le traçage de l'ordonnanceur du noyau a un surcoût négligeable, on vérifie que les contraintes temporelles (telles que la période d'ordonnancement et la tranche de temps allouée) sont satisfaites et si elles ne le sont pas, on montre ce que le système faisait durant ce temps.

Implémentation LTTng (LTTng, 2009), un traceur noyau open-source, à faible impact, a été choisi pour instrumenter les événements requis pour notre liste de patrons. Le compilateur SMC a été utilisé pour générer du code C à partir des machines à états décrites par le langage SM (voir section 3.4.2). Pour chaque événement pertinent, l'analyseur enregistre des fonctions de rappel avec le programme de lecture et de visualisation de trace LTTV. Ce programme lit la trace séquentiellement en une seule passe. L'analyseur maintient une liste de toutes les instances des machines à états (modélisant le même comportement problématique) qu'il invoque séquentiellement. Lorsqu'un événement pertinent est rencontré, l'analyseur appelle les transitions correspondantes pour chaque

machine à états. Le flot de contrôle est alors transféré à la machine à états qui, à son tour, évalue les gardes de la transition et si elles sont évaluées à Vrai, les actions spécifiées sont exécutées. L'état courant de la machine devient alors l'état destination de la transition et le flot de contrôle revient à l'analyseur. Ce dernier invoque la machine à états suivante et ainsi de suite.

Dans certains cas, lorsqu'une transition est déclenchée, une duplication de la machine à états est nécessaire. Par exemple, si le patron consiste à détecter un accès erroné à un descripteur de fichier, une machine à états existerait pour chaque processus accédant à un descripteur particulier. L'analyseur offre une interface de programmation API fournissant des services tels que la duplication et la destruction des machines à états. Certains patrons, tel que celui qui modélise une utilisation erronée des verrous, requiert une seule machine à états par CPU. Dans ces cas, l'analyseur détermine sur quel CPU l'événement est apparu, et appelle la transition de la machine à états de ce CPU.

L'approche à base d'automates offre une grande flexibilité pour la modélisation, la mise à jour et l'optimisation des patrons. En effet, lorsqu'on a instrumenté les événements pertinents pour le patron qui consiste à détecter les mauvaises utilisations des verrous, il a été remarqué que les événements qui définissent l'entrée et la sortie d'une interruption ne sont pas nécessaires vu que cette information peut être recueillie à partir du site de prise du verrou. À ce point-là, un état et deux transitions de l'automate ont pu être éliminés.

Performance En exécutant l'analyseur pour valider des traces de différentes tailles, il a été constaté que le temps d'exécution de l'analyseur est linéaire par rapport à la taille de la trace. Cependant, trois autres facteurs affectent la performance de l'analyseur : le nombre de machines à états coexistantes durant l'analyse (section 3.6), leur complexité (consommation en mémoire et temps de calcul) et la fréquence d'événements pertinents dans la trace.

La performance de l'analyseur a aussi été comparée à celle d'une version dédiée. Cette dernière est implémentée en C et effectue les mêmes validations que celles définies

dans l'automate qui valide l'utilisation des verrous (voir section 3.4.3). La performance de l'analyseur est uniquement 4.5% plus lente que la version dédiée, suggérant que le surcoût d'utilisation des automates pour l'exécution, et non juste pour la modélisation, est acceptable surtout lors d'une analyse en différé.
La taille de la trace a été fixée alors que le nombre de clients *dbench* (un benchmark qui génère un grand nombre d'événements d'entrée et de sortie) exécutés sur la machine tracée varie. Pour le patron qui valide les utilisations des descripteurs de fichiers, le nombre de machines à états coexistantes augmente proportionnellement avec le nombre de clients *dbench*. On étudie alors le surcoût causé par le fait d'invoquer les machines à états. Le ralentissement est proportionnel au nombre de machines traitées simultanément par l'analyseur. Ceci est attendu car, pour chaque événement pertinent, l'analyseur invoque successivement toutes les machines à états coexistantes, même si l'événement est non pertinent dans l'état courant de l'automate. Si l'événement cause un déclenchement d'une transition dans une machine à états, on dit alors que l'événement a été consommé par cette dernière. Dans plusieurs cas, un événement ne peut être consommé que par une seule machine à états. Pour cela, on a introduit une fonction API qui force l'analyseur à sortir de sa boucle principale qui itère sur tous les automates, pour passer à l'événement suivant ; une amélioration non négligeable en termes de performance a été obtenue.

Travaux Futurs En première priorité, nous voudrions supporter les événements synthétiques. En exprimant un événement synthétique à partir de plusieurs événements primitifs, il serait possible de décrire des patrons plus complexes efficacement et d'une façon modulaire. Le temps de calcul par événement pertinent atteint 20 microsecondes pour un nombre total de machines à états coexistantes égal à 516. Par conséquent, si la fréquence d'événements pertinents, se produisant sur un système en production, est inférieure à 50kHz, les CPUs non-exploités complètement pourraient très bien être utilisés pour effectuer la validation des patrons en ligne. La solution proposée est facilement parallélisable et il serait intéressant d'étudier le gain en performance résultant, dans le

but d’effectuer l’analyse en ligne.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF NOTATIONS AND SYMBOLS

API	: Application Programming Interface
DSL	: Domain Specific Language
FSM	: Finite State Machine - Machine à états finis
IDS	: Intrusion Detection System
$LTTng$	: Linux Trace Toolkit next generation
$LTTV$	: Linux Trace Toolkit Viewer
MPI	: Message Passing Interface
$NIDS$	: Network-Based Intrusion Detection System
$OpenMP$	: Open Multi-Processing

INTRODUCTION

Context The development and operation of embedded systems, multi-core processors, multi-threaded and distributed applications with real-time constraints present new challenges that need to be overcome with adequate, efficient and scalable methodologies.

The biggest challenge is to understand what the system is doing at any given time, with a minimal interference, in order to avoid an impact on the performance or the system behavior. To achieve this goal, the kernel of an operating system needs to be efficiently instrumented either statically in the source code or dynamically at runtime, enabling experts to diagnose a large set of problems touching on several fields such as performance debugging and security.

Currently, many efforts are targetting data aggregation from the kernel and user-space at the lowest possible cost. This field is known as kernel and user-space tracing. Systems exhibiting an undesirable behavior are put under monitoring to generate a trace of relevant information, containing the sequence of events that happened on the system just before the time when the problem occurred.

Eventhough it is a relatively new field, open source kernel tracing has rapidly gained a wide popularity among kernel and driver developers. It became an essential tool to exactly understand what the system is doing at any given time and at a high level of granularity. However, analyzing the trace would require system experts to manually identify any suspicious sequence of events that is causing the undesired behavior, and this process is generally very time consuming. Instead, it would be beneficial to encode a list of problematic patterns and look for them efficiently in a trace maximizing therefore the chances of finding problems in a trace.

Tracing could be constantly active for a long period of time, generating billions of events;

it could be activated and disactivated intermittently on specific user-defined conditions. It could also be constantly activated but without writing the data to disk; there, the events are kept in cyclic memory buffers where the oldest data is overwritten by the newest. When the problematic condition occurs, the buffers are dumped to disk for analysis. This approach is analogous to the data flight-recorder found in airplanes and is very useful for accident investigation whenever the problem is hard to reproducible.

When tracing is completed, the next step requires experts to look at the data using basic tools and infer the main cause of the problem. Depending on the nature of the problem, this step may be very time-consuming and inefficient; and in some cases almost impossible.

Our work aims at automating the trace examination process whenever useful, enabling a precise, more efficient and faster data analysis. Problems we try to diagnose relate to different fields and we aim at looking for all problem types simultaneously in one or several large per-cpu traces.

Objectives Trace files of several gigabytes are generated at run-time. We would like to inspect the presence of anomalies in traces automatically. The lack of adequate tools for this purpose motivated us to develop a pattern detection trace analyzer which leaves us in front of three challenges:

1. Provide a simple, flexible and unambiguous way to represent patterns of problematic behavior.
2. Collect relevant events efficiently from production systems with a minimal performance impact.
3. Scan traces in one pass to validate them simultaneously against numerous patterns efficiently.

The project should enable us to encode undesirable properties into patterns which can later be used to automatically validate traces generated from production systems. Once a pattern is found, enough information should be supplied so that measures could be taken to eliminate the occurring problem.

Scientific Methodology The steps we took to reach this goal are summarized as follows:

1. **Typical patterns:** the work started by collecting a set of problematic patterns that would be typically looked for in a kernel trace, by performance and security engineers. This step is essential since we would like to provide a generic framework, supporting new pattern descriptions touching on different fields. This has greatly helped us understand the problem requirements and therefore determine the appropriate mechanism to express and validate multiple hybrid patterns. These patterns are described in Chapter 4.

2. **Analogy with other systems:** Eventhough pattern matching is new in the kernel field, it is not the case in the intrusion detection nor in the parallel computing fields. Many scenario-based intrusion detection systems provide a mechanism to describe security threats or rules, which are then used for validation when analyzing network traffic, whether live or offline. The logged network packets waiting to be processed could consume several gigabytes in storage, and therefore provide the same challenges as in large kernel trace. When a rule triggers a security violation, the security engineers are alerted, and measures are taken to deal with the problem. Similarly, in the parallel computing field, there is a wide interest in automatic performance debugging. Parallel programs may exhibit inefficient behavior especially when the message passing between different processes running on different CPUs don't occur in the right order. It was important to study the different

systems because of the tight similarity of their objective with ours. We didn't find a unique and common pattern description language used among the different systems we studied. However, we came up with a criteria list for a pattern description language based on the typical patterns we collected in the previous step. The main features as well as limitations of the different studied languages are summarized in Chapter 1. We divided the studied languages into 3 seperate categories: the Domain Specific Languages which are further divided into imperative and declarative DSLs, the General Purpose Languages and the Automata-Based Languages.

3. **Automata-Based Approach:** Based on the study we lead on the different pattern description languages which was summarized in Chapter 1, as well as the criteria list described in the previous step (details in Chapter 1), an automata-based approach for pattern description seemed the most convenient. We studied and compared some of the most important automata-based lanugages including STATL (Eckmann et al., 2002), RAGEL (RAGEL, 2009) and SMC (Compiler, 2009). We show how the automata-based approach is applicable by actually using it to implement the collected patterns.

4. **Implementation:** The implementation is divided into two parts. The patterns implementation using the automata-based approach and the analyzer's implementation. The different patterns described in Chapter 4 are implemented as well as the analyzer which checks for their existence in large traces. Since the analyzer needs to be very efficient, and capable of processing multiple traces while validating a set of patterns simultaneously, we studied a set of popular trace analyzers in Chapter 2. Our main interest was basically to study their overall system artchitecture as well as the optimization approaches, if any, they adopt.

5. **Testing:** Once the implementation phase is done, we proceed by testing the performance of the analyzer. The main goal is to study the scalability of the appoach and whether it may be possible to define a large pool of patterns for automatic

validation in reasonable amounts of time.

Thesis Organization The thesis is organized as follows. In Chapter 1, the different pattern description languages used in the Intrusion Detection and Parallel Computing fields, are studied. Furthermore, their main features and limitations are summarized after dividing them into 3 seperate categories. We also argument on the reasons behind adopting an automata-based language instead of an imperative or declarative DSL. Then, in Chapter 2, we study some of the popular trace analysis tools, where our main interest is to identify their optimization techniques as well as their overall system architecture. The core of our work is thouroughly described in Chapter 3. It is a journal article submitted to the Journal of Computer Systems, Networks, and Communications in which a quick litterature review is provided, followed by the explanation and implementation of our approach. Four patterns are implemented using a state machine language and fed to the analyzer for a successive testing phase. Results are presented, highlighting the performance of the analyzer as well as a potential problem it was able to detect. In Chapter 4 we provide some complemantary results that were not included in the Chapter 3 article. In Chapter 5 we provide a general discussion of our work touching on both the scalability and performance of the analyzer. The last Chapter concludes our work.

CHAPTER 1

PATTERN DESCRIPTION LANGUAGES

Analyzing kernel traces for pattern matching requires a non-ambiguous, easy to use pattern description language. It should be possible to express using this language a wide variety of patterns touching on several fields of interest such as security, performance debugging and software testing. We first define the language selection criteria based on the nature of patterns we would be typically looking for in a kernel trace. Then we study a wide variety of languages used in various fields for detecting problematic behavior in traces.

In intrusion detection, languages are used to describe security rules and attack scenarios for validation in audit trails (some of those languages are also used for lexical and syntactic analysis). Furthermore, in parallel computing, other languages are used to express performance-based properties to be be validated in MPI and OpenMP traces. Similarly, kernel tracers such as DTrace and SystemTap provide languages dedicated to perform run-time trace analysis.

1.1 Language Criteria

We first started by collecting a large set of typical patterns of problematic behavior to look for in a kernel trace. We summarize our findings in chapter 3 and based on them we come up with a criteria list that covers the most important aspects we are looking for in a pattern description language. They are:

1. Simplicity: The language should only provide a minimal set of operators sufficient

to describe the variety of patterns.

2. Expressiveness: It should be possible to describe using the language all sorts of known problematic patterns as well as new ones which may be defined in the future.

3. Tracer-independent: The language shouldn't depend on the internal format of the kernel trace. Whether the kernel events are collected using one tracer or another, the pattern description should not vary at all.

4. Unambiguous: The language should have a rigorously defined syntax and semantics allowing only one clear interpretation of a described pattern.

5. Multi-event-based patterns support: Some languages, such as the one used to write Snort rules, provide operators that are useful to process only one event, whereas in many cases, the patterns are composed of multiple events. It should be possible to describe such patterns including the ordering between the events.

6. On-line/Off-line distinction: the language should not make any apriori assumptions of whether the analysis is performed on-line or off-line.

1.2 Taxonomy of pattern description languages

We divide the languages to be studied into three seperate categories: Domain Specific Languages, General Purpose Languages and Automata-Based Languages. Domain Specific Languages are further divided into Declarative Domain Specific Languages and Imperative Domain Specific Languages. The languages studied in this chapter can be good candidates for event-based programming and we study how applicable they could be for us to describe problematic patterns that can be found in a kernel trace.

1.2.1 Domain Specific Languages

Domain specific languages DSLs, previously known as special purpose programming languages, are usually dedicated to solve a particular problem or implement a well-defined domain specific task as opposed to general-purpose programming languages such as C or Java. Examples of domain specific languages include VHDL, a hardware description language, Csound, a language for creating audio files, and DOT, an input language for GraphViz, a graph visualization software. We divide DSLs further into two more categories: Declarative DSLs and Imperative DSLs.

DSL Advantages The need for a new DSL is often justified whenever the introduced language helps formulating a particular problem or solution more clearly and more easily than by using preexisting languages. For instance, it should support the level of abstraction of the specific domain, hiding all the low-level complexity and implementation, so that the field experts can easily understand, maintain, update and develop new DSL-based programs.

1.2.1.1 Declarative DSLs

Declarative programming, in contrast with imperative programming, consists in describing what is to be done rather than how to do it. In other words, the logic of a computation is described rather than the control flow of the program. A popular intrusion detection system called Snort employs a declarative DSL to describe a large number of security related rules which are validated by inspecting the network packets. The language will be studied in this chapter.

Declarative DSLs for IDS DSLs are widely used in the intrusion detection field. Snort and Panoptis are both good examples of intrusion detection systems that use domain specific languages for threats detection. The introduced DSLs provide a number of advantages. First, they provide a domain-specific high-level of abstraction facilitating their use by domain experts. Second, the code is generally self-documenting, reducing the required time to generate the script documentation as well as the time needed to understand the code. Third, by being declarative, they remove the burden of specifying the full solution implementation including the control flow of the program. However they incur an additional cost for the design, implementation and maintenance of the language. Furthermore, the DSL-based programs can be hard to debug.

Snort Snort (SNORT, 2009) is a free, open source, extensible, intrusion prevention and detection system that can be used as a packet sniffer, a packet logger or a network-based intrusion detection system (NIDS). It can operate by validating a database of security rules against the network packets. Rules are used to capture malicious packets before they can cause any damage to the system.

Snort Rules Snort employs a declarative DSL for rules descriptions. Packet networks are inspected by Snort IDS based on the specified rules. As an example, a rule could state that any packet directed to port 80 and containing the string “cmd.exe” is considered malicious and therefore measures could be taken to deal with it. Snort rules are a set of declarative instructions designed to express a pattern to look for in the network packets. Many parts of the packet could be inspected such as the source and destination IP addresses, the source and destination port numbers, the protocol options and the packet payload. Rules are composed of two parts: the rule header and the rule options. The header contains the rule’s action, protocol, source and destination IP addresses and netmasks and the source and destination ports numbers. The rule options are used for

many operations such as content matching, TCP flags testing or payload size checking, etc. Below is an example illustrating a simple Snort rule:

alert tcp any any -> 192.168.1.0/24 111 (content:"|000186a5|"; msg:"mountd access";)

Rule Headers The text outside of the parentheses forms the rule header. The different fields that form a rule header are: Rule actions, protocols, IP addresses, and port numbers. In the first part of the header, we can specify one of the three possible actions:

- Log action: logs the packet that caused the rule to fire.
- Alert action: logs the packet that caused the rule to fire and generates an alert using the selected alert method.
- Pass action: drops the packet.

The next field in the rule header is the protocol. The three protocols that Snort supports are TCP, UDP and ICMP. Then, the source and destination IP addresses are specified in the header. The keyword "any" is used to define any IP address. The Direction operator "->" is used to differentiate between the source and the destination of the packet. Furthermore, the netmask could be specified to designate a block of addresses for the destination. In the last part of the header, port numbers are specified. One or a range of port numbers can be specified either directly or by negation.

Rule Options Snort provides 15 different rule options that are used for operations such as pattern matching or testing the IP and TCP fields. For instance, one can test the TCP flags or the TTL for certain values. All rule options deal with one packet at a time. So basically, writing a Snort rule would consist in defining a set of constraints on the fields of every occurring packet. Snort validates the packets one at a time, and doesn't allow

one rule to be fired based on constraints touching on two seperate packets. This is mainly due to the fact that Snort was designed to process packets online while incurring a low impact on the system's performance. By expressing more complex rules, the run-time packet inspection would significantly impact the system's performance.

Snort for pattern description The language used for Snort rules is a good example of a declarative DSL. The rule writer doesn't have to worry about the implementation details, such as accessing the fields pointers in the packet headers in order to perform the tests, or opening a file to log the packet. Providing a DSL for kernel trace analysis such as Snort language, would enable programmers to develop patterns at a high level of abstraction. However, the major limitation we found in Snort rules is that they don't allow us to express relationships between different events. In fact, many problematic patterns could only exist upon the occurrence of two or more events. For instance, detecting a potential deadlock by analyzing a kernel trace cannot be achived using a one-level, Snort-like rule, and therefore we consider other alternatives.

1.2.2 Imperative DSLs

According to the definition found in Wikipedia, imperative programming is a programming paradigm that describes computations in terms of statements that change a program state. In fact, imperative languages, in contrast with declarative languages, provide a mean to specify "how" something should be done rather than simply "what" should be done. Procedural programming for instance, is considered imperative by its nature, and is possible when the imperative language supports structuring statements into procedures. Furthermore, object-oriented programming such as in C++ and Java is also considered as being an imperative programming at their core but not Domain Specific. They are general purpose programming languages. Imperative Domain Specific Lan-

```
01    #rule1: when the encountered event is 'login' and the result is failure
02    #trigger the rule Count_failed_logins.
03    rule Failed_login (maxtimes, duration: integer)
04    begin
05    if evt='login' and res='failure' and is_unsecure(terminal)
06          -> Trigger off for next Count_failed_logins (maxtimes-1, timestp+duration )
07    fi;
08    Trigger off for next Failed_login(maxtimes, duration)
09    end
```

Figure 1.1 Russel Rule: Activates another rule upon detection of a failed login

guage are found in Intrusion Detection Systems such as ASAX. ASAX is an IDS that uses RUSSEL, a rule-based imperative DSL designed specifically for audit-trail analysis. In addition to RUSSEL, we study in this category SystemTap and Dtrace scripting languages which are also considered as imperartive DSLs.

1.2.2.1 RUSSEL

The RUle-baSed Sequence Evaluation Language RUSSEL (Habra et al.,), is an imperative, rule-based DSL that is used for audit-trail analysis as a part of the ASAX Intrusion Detection project. It is specifically designed to allow queries to be processed in one pass over the trace. Unlike the limitation found in Snort rules, RUSSEL provides a mechanism for one rule to trigger another. This can greatly enhance the flexibility of describing more complex rules based on simpler ones.

In the example shown in Figures 1.1 and 1.2, two rules are described. The first one tries to detect a failed login in the trace. Whenever it does, it activates another rule which counts the number of subsequent failed logins. When the maximum number of failed logins is reached (i.e. countdown reaches zero), the rule sends an alert message.

When the condition at line 5 is true, the rule Failed_login triggers off the Count_failed_logins rule. Then at line 8 the rule triggers itself unconditionally, keeping itself active during

```
01    #rule2: activated by rule1. Counts the number of subsequent failed logins.
02    rule Count_failed_logins (countdown, expiration: integer)
02    if evt='login' and res='failure'
03          and is_unsecure(terminal) and timestp<expiration
04          -> if countdown > 1
05                ->Trigger off for next Count_failed_logins(countdown-1, expiration);
06                  countdown=1
07                ->SendMessage ("too much failed login's")
08             fi;
09    timestp >= expiration
10    -> Skip;
11    true
12    -> Trigger off for next Count_failed_logins(countdown, expiration)
```

Figure 1.2 RUSSEL Rule: Counts the number of failed logins

the whole analysis time. This is not the case for Count_failed_logins rule shown in Figure 1.2 where the self-triggering is skipped whenever the condition timestp >= expiration is true.

RUSSEL for Pattern Description RUSSEL is a good example of a DSL that is flexible enough and doesn't have the limitation found in Snort rules. The idea is that one rule can trigger another and therefore any pattern composed of a succession of events may be expressed by dedicating one rule per event. While the rule triggering mechanism is interesting, complex patterns composed of multiple threads of events, which can also be interconnected, can quickly become very hard to express in the language. Consider for instance a pattern described using the following set of rules r1, r2, r3. Suppose that r1 triggers r2 upon the occurrence of a particular event in the trace satisfying a certain condition. Similarly, r2 triggers r3. Now, if the same pattern can occur simultaneously for different user-space processes, then it may not be possible to detect the two occurrences since even if every active rule has its local variables, it is not possible to have two active rules of the same type each having its own local variables. Furthermore, the rule triggering mechanism can be easily ported into a general purpose language by implementing it in a dynamic library for example. This would remove the maintenance cost

of the language and the resulting code may possibly run faster, and be easier to debug.

1.2.2.2 DTrace D Language

DTrace (Cantrill et al., 2004) is a dynamic and static instrumation framework for both kernel and user-space tracing. It runs under Solaris 10, Mac OS X 10.5 and FreeBSD. Recent work aims at porting DTrace for Linux. Dynamic instrumentation consists in adding instrumentation sites dynamically (i.e. at runtime) in contrast with static instrumentation where the source code is modified for the same purpose. DTrace employs a C-like language called D, that is used to specify probe points (instrumentation sites) in the kernel and to implement their associated handlers. These handlers could be used to perform run-time trace analysis such as pattern matching. The D language supports all of the ANSI C operators and data types as well as the struct, union and enum types.

D variables

Thread-local variables D supports the declaration of thread-local variables, where every thread gets its own copy of the variable. The per-thread variable can later be used in any probe handler.

A small example illustrating the declaration and use of a thread-local variable via the prefix self-> is shown in Figure 1.3. The example was taken from (DTrace., 2009).

At line 1, a probe handler is defined which is to be called at every entry of a read system call. A probe description has the following form: “provider:module:function:name”. Inside the handler, the event time stamp is saved in a thread-local variable accessed using the special operator self->. Every user-space thread issuing a read system-call will get its own copy of the t variable. At line 5, another handler is defined which is to

```
00    syscall::read:entry
01    {
02            self->t = timestamp;
03    }
04
05    syscall::read:return
06    /self->t != 0/
07    {
08           printf("%d/%d spent %d nsecs in read(2)\n",
09                  pid, tid, timestamp - self->t);
10          /*
11          * We're done with this thread-local variable; assign zero to it to
12          * allow the DTrace runtime to reclaim the underlying storage.
13          */
14          self->t = 0;
15    }
```

Figure 1.3 DTrace: Compute the time spent per thread in the read system call

```
# dtrace -q -s rtime.d
100480/1 spent 11898 nsecs in read(2)
100441/1 spent 6742 nsecs in read(2)
100480/1 spent 4619 nsecs in read(2)
100452/1 spent 19560 nsecs in read(2)
100452/1 spent 3648 nsecs in read(2)
...
```

Figure 1.4 DTrace: Output obtained upon running Example 1.3

be called just before returning from every read system call. Inside the handler, the time spent serving the system call is computed and the variable t is reinitialized to 0.

An example of the output is shown in Figure 1.4.

D Clause-Local Variables The D language also supports the declaration of clause-local variables which are only active during the execution of the probe handler. Their values remain persistent across handlers enabling the same probe.

```
00    syscall::write:entry
01    {
02            @counts[execname] = count();
03    }
```

Figure 1.5 DTrace: using the count aggregating function

External Variables Since the D probe handlers are executed at run-time, it is possible to access types and symbols from the kernel or the kernel modules. For instance, the Solaris kernel has a global variable called kmem_flags. It can be accessed in a D program by using the backquote character (‘) as a prefix before the variable name (i.e. ‘kmem_flags).

Aggregating functions The DTrace aggregating functions provide a mechanism for performing a set of predefined computations in a probe handler and are mainly used for data aggregation. As an example, the script shown in Figure 1.5 counts the number of times the write system call has been entered by storing the results per executable.

D For Pattern Description The D programming language is considered an imperative C-like language and provides at the same time new mechanisms dedicated for kernel trace analysis such as probe site definition, support for thread-local variables and a set of aggregating functions. This is why we placed the D language in the category of the imperative DSLs. There are many features of the language that are not covered here. We only cover those features that could be useful for our offline pattern recognition trace analysis.

The D language’s new features are very well adapted for online analysis. They include probes activation, accessing kernel variables and aggregating only the relevant data at run-time to minimize the amount of information to store. However we see two major limitations in terms of applicability to our problem. First, many features of the D lan-

guage are only applicable for online analysis such as activating probe points in the kernel or accessing the live kernel data. Second, the D language does not provide an easy way to describe complex patterns of problematic behavior. For example, the programmer would still have to manually encode the pattern in question across the different probe handlers, using an imperative C-like language.

1.2.2.3 SystemTap

SystemTap is a dynamic tracer for Linux that uses the kprobes infrastructure to dynamically instrument the kernel. It is an open source project with contributors from IBM, Red Hat, Intel, Hitachi, Oracle and others. Unlike the D script files which are interpreted, SystemTap scripts are translated into C code, compiled as kernel modules and inserted into the Linux kernel. As in DTrace, SystemTap scripts are used to activate probe points in the kernel and to implement their associated handlers. Furthermore, Systemtap scripts can have global variables, conditional statements, while and for loops. However the SystemTap language lacks types and declarations but adds associative arrays and simplified string processing.

Extraction functions As in Dtrace's aggregating functions, SystemTap has the extraction functions. The aggregation operator ($<<<$) is used to keep track of the values of interest. An example using the aggregation operator is shown in Figure 1.6.

At line 2, the probe point is defined. Probe definition has the following syntax:

$probe\ PROBEPOINT\ [, POBEPOINT]\ [STMT...]$

The global variable reads declared at line 1, can be seen as an array storing all the values and their number of occurrence, that are assigned to the array using the aggregation op-

```
01    global reads
02    probe netdev.receive {
03       reads <<< length
04    }
05    probe end {
06       print(@avg(reads))
07    }
```

Figure 1.6 SystemTap: using the aggregation operator

erator $(<<<)$ as shown at line 3. Then, at the end of the analysis, the extractor functions could be used to perform some operations over the aggregated data. For example, the avg() extractor function shown at line 6 is used to compute the average of all values accumulated into reads. Similarly, other functions can compute the minimum, maximum, sum or count the number of accumulated values. Other statistical functions could be called to show the distribution of values in text based histograms at the output.

SystemTap has many other features. For instance, it performs safety and security checking in order not to allow the running kernel module in which the probe handlers are running from performing unsafe operations which may lead to a kernel crash or data corruption. We are mainly interested in the features that could be applicable in our work.

Systemtap for Pattern Description SystemTap scripts can enable the programmer to perform many computations at runtime similarly to DTrace. Both offer specialized functions to visualize the distribution and statistics of relevant data. We considered the Systemtap language as being procedural and imperative due to the similarities it has with the C language. It is also a DSL because it allows the programmer to define probe points in the kernel and to perform specialized functions for data aggregation or for accessing the kernel data structures. The main features for both Systemtap and DTrace are well oriented towards online analysis putting less the emphasis on pattern description. They

don't provide a simple mechanism to describe a pattern composed of several events where multiple instances of the same pattern type could be alive simultaneously at certain periods during the analysis. The task of manually encoding such patterns across the different probe handlers could be a rather challenging task.

1.2.3 General Purpose Programming Languages

General purpose programming languages (such as C, C++ and Java) in contrast with Domain Specific Languages, are used to solve a wide variety of problems not necessarily related to a particular field. General purpose programming languages could be further divider into several other categories but they are out of the scope of our work. General purpose programming languages, particularly object-oriented languages could very well be used for patterns description. They actually provide a number of advantages over Domain Specific Languages. First, the code is easily debugged using sophisticated and dedicated debuggers. Second, since the programmer has the full control over the control flow of the program, he may be able to better optimize his code for his well-defined problem and therefore, achieve a faster program execution. However, using general purpose programming languages for patterns description may result in a larger number of lines of code, possibly making the program less trivial to undestand. Furthermore, field experts will have to learn the programming language in order to maintain or update the patterns in question.

Under this category, we identify the EXPERT analyzer tool discussed in 2.1. In our work, we will be using a general purpose programming language for patterns description in order to evaluate the performance of the dedicated approach and therefore be able to compare it with our approach.

1.2.4 Automata-Based Programming Languages

Automata-based programming, particularly finite state machine programming, consists in describing the problem to solve in the form of a state machine composed of a finite number of states, transitions and actions. The program evolves by transitioning from one state to another depending on both, the current state and the input. This may enable one or several actions to take place. Actions could take place when a transition is triggered or when a state is entered or left.

A problem described in the form of a state machine can be represented by a state diagram which is visually friendly. Furthermore, it is often easy to update such programs where the modification usually involves adding or removing states and/or transitions. Automata-based programming is used in many fields such as lexical and syntactic analyses, modeling real-time applications for formal verifications, and intrusion detection. In this chapter, we study Ragel, a state machine language and compiler, STAT an intrusion detection system based on FSM description of security patterns and the State Machine Compiler SMC which is used in many fields and can compile the state machine SM language into 14 different general purpose programming languages.

1.2.4.1 Ragel

Ragel (RAGEL, 2009) is a language for specifying state machines that can recognize regular string expressions. It has a compiler that converts the state machine definitions into six different target languages which are: C, C++, Objective-c, D, Java and Ruby. The generated code tries to match patterns to the input favouring longer patterns over shorter ones. Ragel can also generate the state machines in Graphviz's Dot file format for visualization. Ragel can optionally employ a state minimization algorithm which reduces the number of states in a FSM by merging equivalent states. The algorithm runs

```
00    #include <string.h>
01    #include <stdio.h>
02
03    %%{
04       machine foo;
05       main :=
06          ( 'foo' | 'bar' )
07          0 @{ res = 1; };
08    }%%
09    %% write data;
10
11    int main( int argc, char **argv )
12    {
13            int cs, res=0;
14            if ( argc > 1) {
15                    char *p = argv[1];
16                    char *pe = p + strlen(p) + 1;
17                    %% write init;
18                    %% write exec;
19    }
20
21    printf("result = %i\n", res);
22    return 0;
23    }
```

Figure 1.7 Example illustrating the Ragel syntax

at O(n log(n)) where n is the number of states and requires O(n) memory storage.

A simple example that highlights the Ragel syntax is shown in Figure 1.7

The Ragel code is placed between double percent signs and braces, i.e. %%{ Ragel code here... }%%. A single Ragel statement should be preceeded by double percent signes. The machine shown in the example recognizes the two strings 'foo' and 'bar' and whenever one of them is detected, the variable res is set to 1. The variable res can be later be accessed by the program written in one of the target languages supported by Ragel. The main program passes to the FSM a pointer to the beginning of the data to be processed and one pointing to the end of the buffer.

Language Operators Only the language operators which are relevant for our work will be covered in this section. The language includes manipulation operators that allow

the programmer to specify the actions of the state machine (i.e. transition actions and state actions) and to assign priorities to transitions. By assigning priorities to transitions, the nondeterminism problem is eliminated. For instance, if two transitions are possible for a given character, then the one with the higher priority will take precedence.

Finie State Machine's Actions Actions are block of code that could be fired by an FSM. The actions can be triggered by four different transition types identified by four different action embedding operators:

1. Entering transition operator: $>$
2. Finishing transition operator: @
3. All transition operator: $
4. Leaving transition operator: %

The "**entering transition**" operator is used to specify the actions to be triggered by any transition leaving the starting state. For instance, when the first character of a sequence is encountered, a transition originating from the starting state is fired. Therefore, the entering transitions, if any are specified, are executed.

The "**finishing transition**" operator is used to specify the actions to be triggered by any transition leading into a final state. For instance, when the full character sequence is matched, the state machine transitions to its final state. At this point, the finishing transition action is executed. Finishing transition actions can be executed more than once if the state machine has any internal transitions out of a final state.

The "**all transition**" operator is used to specify the actions to be triggered by any taken transition in the finite state machine. For instance, for every character in the sequence being matched, the "all transition" action is executed.

The "**leaving transition**" operator is used to specify the actions to be triggered when the state machine leaves its final state.

Ragel has other types of operators that help create new machines from previous ones. These operators include: the **Union** operator which produces a machine that matches any string in machine 1 or machine 2; the **Intersection** operator which produces a machine that matches only the strings that are both in machine 1 and machine 2; the **Difference** operator which produces a machine that matches the strings in machine 1 but are not in machine 2.

Ragel for Pattern Description While Ragel provides many features such as FSM optimization and visualization, it does not allow us to specify different actions on different transitions. In fact, the four types of transition actions it supports are mentioned previously in this section and none of them allows us to explicitly assign different actions for different transitions. This is a major limitation for us since we may need to assign different actions triggered by the occurrence of different kernel events in the trace. Not only that, we sometimes need to assign two or more different actions triggered by the occurrence of the same kernel event where the action to be executed will depend on the current state of the FSM.

1.2.4.2 STATL

The State Transition Analysis Technique (STAT) (Eckmann et al., 2002) is a scenario-based, intrusion detection system (IDS). STATL is the language it uses to describe the attack scenarios which will be used by the STAT core to detect possible ongoing intrusions in a stream of events. We study in this section the STAT language and show how applicable their approach could be for kernel trace analysis.

States and Transitions The language provides constructs to define the states and transitions of a given attack scenario. Since any security breach is usually composed of one or more steps, STAT maps every step of a given attack into a state in a finite state machine (FSM) representing the full attack scenario. Transitions between states represent the evolution of the attack. A critical state could therefore be reached if the attack was successful, or at least imminent. At this point, the notification action already specified by the pattern writer will be triggered so that system administrators take the necessary measures to solve the problem.

Scenario Definition A STATL scenario is an attack definition encoded into a finite state machine description. A scenario is identified by its name and can have parameters when being loaded. Scenario parameters are very useful when the user wants to set certain variables when loading the scenario, avoiding therefore the recompilation process. Since more than once instance of the same scenario could coexist, the parameters are accessible by the different instances as global constants.

Variable Declaration Inside the body of a scenario declaration, the language supports the declaration of local, global variables and constants. Global variables are shared by all the instances of the same scenario. As for local variable, each instance has its own copy of it.

State Declaration Only one state has to be set as the initial state. When a new scenario is loaded, it is created in its initial state. States can have optional assertions which are tested prior to state entry. Only when the assertion - if present - is evaluated to true, the state would be entered. Assertions can access the event fields, the local and global variables. They cannot however affect their values. The affectation is performed only inside the code blocks. Figure 1.8 illustrates the declaration of an initial state S1, and

```
Scenario example
{
        const int threshold = 64;
        int counter;
        ...
        state S1 { }
        ...
        state S3
        {
                counter > threshold
                {
                        log("counter over threshold limit");
                }
        ...
}
```

Figure 1.8 Example of State Declaration in STATL

another state S3 for which a state assertion is declared (counter $>$ threashold). For the same state S3, a code block is defined in which the action to take when the assertion is evaluated to true, is specified. The code block is delimited by braces in which statements are executed in order and can include regular assignments, for and while loops, if-then-else conditions as well as procedure calls.

Transition Declaration Transistions are identified by their name and the pair of states they connect together. The source and destination states could be the same and in this case, the transition is called a loop transtition. Transistions are triggered when a particular event occurs in the trace. This event on which the transition is sensitive, needs to be specified in the definition of the transition. In the example shown in Figure 1.9, the transition t2 joins state S1 with state S2, and is sensitive on the event READ. As in the case of states definition, transitions can have assertions as well as code blocks (actions). The assertion above states that the euid and ruind fields of the event must differ. In that case, the transition is fired and the state s2 is reached. The userid variable gets updated

```
scenario example
{
        int userid;
        ...
        transition t2
                (s1->s2) nonconsuming
        {
                [READ r] : r.euid != r.ruid
                { userid = r.euid; }
        }
        ...
}
```

Figure 1.9 Example of Transition Declaration in STATL

as a result.

Evaluation Order Since transitions and states have assertions as well as code blocks (actions to take) an execution order is needed and is as follows:

1. Evaluate the tranition assertion. If True, then
2. Evaluate the state assertion. If True, then
3. Execute transition code block, possibly modifying local and global environments;
4. Execute state code block, possibly modifying local and global environments;

Timers STATL supports the definition of timers which are initialized inside the code block of a transition or a state. As in the case of regular variables, timers could be either local or global. The timer is started at the current event time stamp. A timeout t is specified and will be fired t seconds after the event time stamp. A transition needs to be defined to be triggered upon the timer expiration. The timer will be reset if restarted in another state code block.

STATL for Pattern Description By the time the article (Eckmann et al., 2002) was written, around 35 attack scenarios were decribed using the STAT language and the authors claim that no limit in the expressiveness of the language was found. Furthermore, a recent work has been done to automatically translate the large collection of rules written for Snort - a popular intrusion detection system - into STATL scnearios. STATL provides many features not discussed here because they are out of the scope of our work. We believe the STATL approach could be very well applicable for kernel traces for many reasons. First the objective is quite similar; in both cases, patterns are composed of a sequence of traced events that could be used to describe security threats or performance problems. Second, the automata-based approach provides an easy way to describe complex patterns from multiple simple ones, through the creation of synthetic events.

1.2.4.3 State Machine Compiler

The State Machine Language compiled by the State Machine Compiler SMC (Compiler, 2009), provides many of the features found in STATL described previously. The SMC compiler is a free, open-source java program. It takes a state machine description written using the SM language (.sm file), and converts it into one of the 14 following languages: C, C++, C#, Groovy, Java, Lua, Objective-C, Python, Perl, PHP, Ruby, Scala, Tcl and VB.net. The description of the SM language here doesn't cover all of its aspects, however it highlights only those features that are relevant for us.

An example illustrating the syntax of the SM language is shown in Figure 1.10. The example illustrates the declaration of two states S1 and S2. From State S1, a transition called trans1 is possible. A main program should be typically receiving some input and invoking transitions accordingly. When the trans1 transition is triggered, the transition guard test() is evaluated, and if the result is True, the destination state S2 is reached. State S2 is defined at the end of S1's code block. Furthermore, the transition action

```
S1
{
        trans1(arg1: long) //Transition
                //Guard condition
                [test()]
                S2      //End state
                {
                        ... // Transition actions here
                        do_something();
                }
}
S2
{
        ...
}
```

Figure 1.10 Example of Scenario Declaration in SM

do_something() is called.

SM Transitions Transitions are FSM functions that are called from the main program upon the encounter of a certain event. These transitions can receive an argument list which is typically used in the guard conditions or possibly in transitions actions.

Transition Actions The transition actions are enclosed between braces as shown in the example in Figure1.10. Transition actions should be implemented in the target language, e.g. in C, and linked with the code generated from the SM language. The implemented function can have an argument list and needs to have a void return type or else, the FSM simply ignores the return value.

Transition Guards Transition guards or assertions should be composed of valid statements written in the target language source code. In fact, SMC copies the guard condition verbatim into the generated output. The transition is taken only when the guard

```
State1
        Entry
        {
                actionOnEntry();
        }
        Exit
        {
                actionOnExit();
        }
{
        //State Body
        ...
}
```

Figure 1.11 Example of State Actions Declarations in SM

condition is evaluated to true. Whenever a transition is not defined for a given state, any call to this transition from the main program will trigger a default transition. The destination state of the Default transition could be the same as its origin state and therefore, irrelevant transitions for this particular state won't have any effect on the FSM.

SM States States can have multiple transitions with the same name and argument list but with different transition guards. In this case, the order of precedence is the same as the order they are defined in the .sm file. The only exception is that the unguarded transition is given the least priority compared to guarded ones no matter where it is defined in the .sm file. States can also have on-entry and/or on-exit actions.

State Entry/Exit Actions Actions triggered at a state entry or exit are declared as shown in Figure 1.11. Entry/Exit actions are very useful to update some internal data structures or start/stop a timer.

SMC versus STATL

Similarities As our study shows, there are many similarities that the SM and the STAT languages share. They include:

- **Transitions with arguments list.**
- **Transition guards (or assertions):** In both languages, assertions can access the arguments list.
- **Transition actions with arguments list:** In both language, actions can access the arguments list.
- **State entry actions:** In both languages, it is possible to define an action to be executed upon any state entry.

It is important to realize that these common features are very important for patterns description. In fact, transitions are to be triggered upon encountering certain events in the kernel trace. The event fields are to be passed as arguments to the corresponding transition. Based on specific conditions (or guards), possibly relying on the event fields, a transition should take place or not. Then, whenever the transition occurs, actions need to be taken to update some local or global variables that could be later used in future assertions.

Differences There are some differences between the two languages though:

- In STATL it is possible to have state assertions unlike the SM language. However this problem can be overcome simply by replicating the state's assertion into every transition heading towards this state.

- The SM language allows the definition of state exit actions unlike the STAT language. This is also a minor problem and could be solved by replicating all the state exit actions into the transitions leaving the state.

- Consuming/Non-consuming/unwinding transitions: STATL defines three types of transitions: Consuming, Non-consuming and unwinding transitions.

  - **Non-consuming transitions:** According to STATL, a non-consuming transition is used to "represent a step of an occurring attack that does not prevent further occurrences of attacks from spawning from the transisiont's source state". In other words, when a non-consuming transitions fires, a new instance of the same FSM is forked so that the origin state and the destination state are both valid once the transition takes place.
  - **Consuming transitions:** In contrast with non-consuming transitions, the firing of a consuming transition invalidates the origin state and therefore no forking is required.
  - **Unwinding transitions:** According to STATL, "the firing of an unwinding transition from state Sy back to a previous state Sx for a certain scenario instance, causes the deletion of all the scenario instances that were created by the series of events that brought the unwinding instance from state Sx to state Sy."

  In SMC, all transitions are naturally consuming which means that no forking neither deletion of other scenarios will take place. In order to support scenario forking or deletion using SMC, one has to explicitely assign actions and implement them in order to achieve such goals.

Given the flexibility of both languages, it turned out that it may be possible to manually implement one language's features in another in order to achieve the same goals.

Choosing the SMC compiler We select the SMC compiler for our patterns description for many reasons. First, according to our study, the state-machine language provides similar features to those offered by the STAT language; STAT being a popular Intrusion Detection System having a large pool of attack scenarios described by the STAT language. Second, the SMC compiler is a free, accessible, open-source project which is important for us in order to perform our experimentations. Third, it turned out that the features we found in STATL but were not present in the SMC State Machine language could be easily implemented via the SMC transition actions.

CHAPTER 2

TRACE ANALYZERS

In the previous chapter, we studied the pattern description languages used by some popular analysis tools. In this chapter, we focus more on the main features of some of the trace analysis tools as well as on the optimization approaches they adopt. Some of these tools are used for network intrusion detection while others for detecting performance problems in parallel applications. While some of the studied tools analyze events offline, others do so at runtime. Our main concern in this study is to see how applicable their approaches could be, which would help us reach our goal for efficiently analyzing large kernel traces.

2.1 EXPERT

The EXPERT analyzer (Wolf et al., 2004), is a tool that automatically detects performance problems in OpenMP and MPI events saved in a trace file in the EPILOG format. It is part of a lager project called KOJAK which includes user-space instrumentation tools as well as a graphical user interface to show analysis results. EXPERT uses EARL, a high-level interface which provides random access capabilities to single events in EPILOG event traces.

2.1.1 EXPERT's Architecture

The Expert's overall architecture is shown in figure 2.1. Permission to reproduce the EXPERT figure found in (Wolf et al., 2004), in this thesis, was granted by the article's

first author, Felix Wolf.

The EXPERT's architecture is divided into two parts: the instrumentation component, and the analysis component. The user program instrumentation uses three tracing modules: one to trace openMP calls (OPARI), a second module is used to trace user functions (TAU), and a third module (PMPI) used to trace MPI calls. All three modules use the EPILOG run-time library which standardizes the traced events format. The instrumented executable will generate a trace file in the EPILOG format upon execution. The EXPERT's analysis component takes the trace file as input, and uses the EARL library to read the trace in one pass from the beginning till the end. At the end of the analysis, a results report is generated and can be used by the EXPERT presenter for results visualization.

2.1.2 Performance Properties

EXPERT's performance properties are Python classes which identify inefficient behavior in MPI and OpenMP applications including inefficient communication and synchronization. As an example, the Late Sender property consists in finding the cases where the sent message occurs much later than its corresponding blocking receive. Performance properties register call back functions for every event of interest. EXPERT reads the trace once from the beginning till the end, and invokes the call-back functions for every registered event. Multiple call-back functions could exist for the same event. The call-back functions can also access other events by following the links between the events (e.g. a receive event linked to its corresponding send event) or by accessing the updated state information such as the "message queue", which is an internal data structure containing all the events generated by the messages that are currently being transferred. An enhanced version of EXPERT allows patterns to generate compound events (also known as synthetic events) and to register for compound events generated by others. This helped

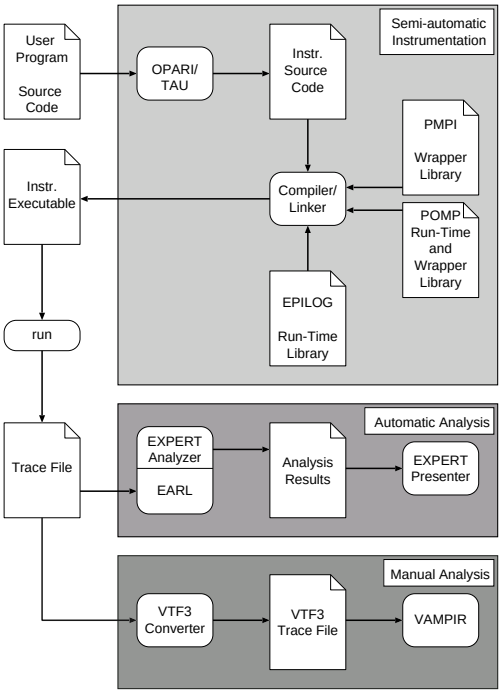

Figure 2.1 The EXPERT architecture

solve the case where a performance property is a subproperty of a more general property. Once a compound-event has been detected by a general pattern and published, a more specialized pattern can now consume it. This prevents redundant computation to occur from both pattern classes. Compound events also lead to more compact pattern specifications.

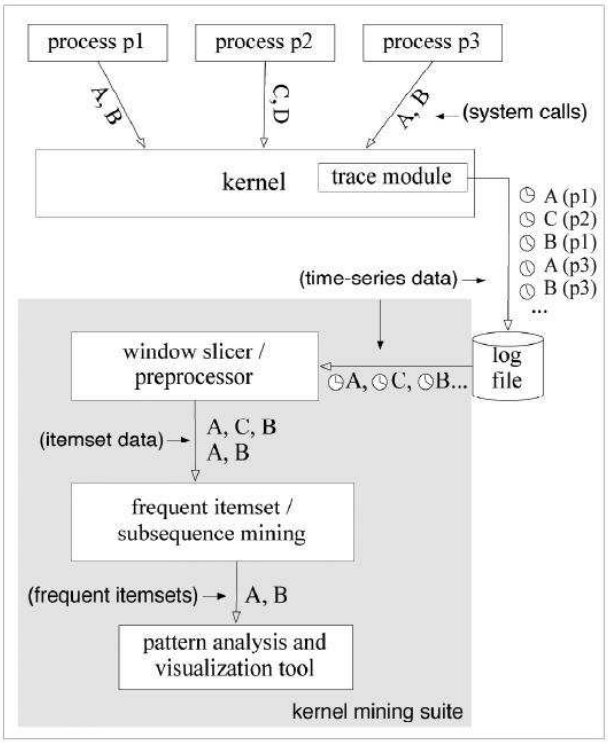

Figure 2.2 System Architecture

2.2 Frequent Pattern Mining

Frequent Pattern Mining for Kernel Trace Data (LaRosa et al., 2008) is a recent work aiming at analyzing kernel traces to detect the most recurring interprocess communication patterns impacting the system's performance. Recurring patterns may not be necessarily composed of an ordered sequence of events due to the reordering introduced by the scheduler. Temporally proximal events found in a kernel trace are treated as parallel and therefore unordered during the analysis.

2.2.1 System Architecture

The overall system architecture is shown in Figure 2.2. Permission to reproduce the figure was granted by one of the authors named Li Xiong. The Linux kernel was instrumented using the Linux Trace Toolkit LTT. The events of interest were mainly the system calls issued by the different processes running on the system as depicted in the figure. A trace file containing these events is generated. Since the work aims at finding the frequent unordered events forming a pattern, it would have to treat all the events found in a trace as parallel. This would lead to a linear data growth in memory which could result in a problem for large traces. Instead, the window slicing technique is used. It consists in dividing the trace into event sequences of a window period, treating them as parallel. The different events found in one window, form one record in the generated database. Then the maximal frequent itemset mining is performed in order to find the maximal frequent itemsets.

2.2.2 Discussion

The frequent pattern mining approach is useful to detect which process is the source of system overhead when traditional tools such as top cannot. For instance the gtik version 2.0 produces a number of high-impact X programming calls. It was possible to identify the itemset containing the list of events issued by gtik and are responsible for much of the Xfree86's work. This new approach is particularly useful for identifying which processes are impacting the performance of the system, whether these processes are the ones holding the CPU the most, or causing other processes to do so. However, it does not allow us to find specific problematic patterns that could occur very rarely in a trace. For instance, catching a potential deadlock is unlikely to occur whereas finding a denial of service attack such as the SYN flood attack could very well be detected using this approach.

This approach has two main limitations for us: first, by relaxing the ordering constraints, it won't be possible to catch patterns that rely mostly on a sequence of ordered events. For instance, attack scenarios usually consist of a sequence of two or more steps that need to be ordered in time of occurrence so that the attack is successful. This is why event ordering is very important for us. Second, frequent patterns occuring in a trace aren't the only potential source of performance bottlenecks. Some patterns occurring maybe once or twice could have some drastic effects on the overall system performance.

2.3 ASAX

The Advanced Security Audit-trail Analysis on uniX, ASAX, (Habra et al.,) is a scenario-based intrusion detection system that aims at analyzing audit trails to determine different kinds of security breaches caused by external penetrations, internal penetrations and viruses. External penetrations are caused by an unauthorized user who tries to access the system. This is caught when sucessive login commands fail. Internal penetrations take place when a constrained user tries to bypass his privileges. Viruses can also be caught if a sequence of events found in the audit-trail can uniquely identify the virus. ASAX uses the Russel language studied in 1.2.2.1, to describe any problematic occurrence of events. This is helpful to detect known penetrations. Furthermore, ASAX supports detecting unknown penetrations whenever a user activity deviates from his normal profiled activities.

2.3.1 ASAX Features

In order to support different trace formats, ASAX defines the normalized audit file format (NADF). Traces need to be converted first into the NADF format, using format adaptors, for analysis. Due to the large size of audit traces, ASAX goes over the trace in just one

pass analyzing it sequentially record after record. There is no way to access information from a past event. To overcome the problem, relevant data from past events need to be encapsulated into the set of active rules. One optimization idea it applies consists in converting the boolean conditions found in the rules description into more appropriate binary trees. For instance, in a given tree, if the current node's boolean expression is evaluated to false the left child is processed, otherwise the right one is. This is done in order not to evaluate elemantary conditions in many cases for every record.

2.3.2 Discussion

ASAX's interesting approach resides not only in the fact that it can detect predefined problematic patterns, but also in detecting new problems which occur when a user's behavior deviates from his preestablished profiled suggesting that an intruder has successfully accessed the system. Furthermore, by defining a normalized trace format, one could use the ASAX analyzer as a universal trace analyzer. However, converting large traces containing billions of events would be very costly. In terms of expressivity, the Russel language was studied in the previous chapter suggesting that two or more instances of the same pattern could not be easily expressed due to language limitation.

2.4 STAT

The State Transition Analysis Technique STAT (Vigna et al., 2000), is a scenario-based Intrusion Detection System. Computer penerations are encoded in a finite state machine that can take the system from an initial safe state to a final compromised state. It employs a flexible state-machine language for patterns description. The STAT Language is covered in the previous chapter where we studied the different pattern description languages.

2.4.1 STAT Features

The STAT-based instrusion detection system translates attack scenarios (or scenario plugins) written in the STATL language into C code, then compiles them into dynamically linked libraries (.so) which are linked into the runtime architecture. Traces (audit records) are generated from the operating systems's event logging facilities or from a network sniffer. They are preprocessed in order to filter out uninteresting events and to encapsulate the relevant ones into the normalized STAT events.

A normalized STAT event has a type, a timestamp and a reference to the application-specific event data. The runtime core loads a number of scenario plugins components. The first scenario instance is created at plugin loading time, at it's initial state. Depending on the scenario in question and the transition to be fired, new instances of the same scenario may need to be forked. When an event is read, the core determines the set of instances that may be interested in the event.

STATL identifies three types of transitions: consuming, non-consuming and unwinding. When a non-consuming transition is fired, a new scenario instance is forked. This occurs when the fired transition does not prevent further occurrences of the same attack from spawning from the scenario's initial state.

For instance, if the first step of a 2 steps attack occured on behalf of a certain user, it doesn't invalidate the fact that another user may start the attack from the first step and therefore a new finite state machine for the second user needs to be forked. In contrast, consuming transitions invalidate the source state and therefore no forking is required. A third transition type is called unwinding. In some cases, a particular event may require the return to an earlier state and at the same time, invalidate all the scenario instances that were forked from this earlier state.

2.4.2 Discussion

Several tools based on the STAT core framework have been implemented. For instance, USTAT is a Unix host-based IDS. It uses the Sun Microsystems Basic Security Module for data tracing of up to 125 event types of which only 35 are used by USTAT. It checks for scenarios causing a security violation such as non-root user making changes in restricted-write directories.

The STAT core was developped using the GNU build system and was ported to different operating systems including Solaris, Linux and Windows NT. The STAT approach is very interesting and applicable for pattern detection in kernel traces. The main features of the STATL language such as transitions guards and actions are also found in the State Machine Language, having an accessible, open-source compiler which we will be using in our work. Also, by doing so, we avoid converting large traces with billions of events into STAT events which is a very time consuming task for traces of several gigabytes. Alternatively, the conversion could be done at runtime increasing therefore the analysis execution time.

2.5 DTrace

DTrace is a dynamic instrumentation framework integrated into the Solaris 10 and Open Solaris operating systems. Dynamic instrumentation is architecture specific and induces a performance impact only when tracing is activated. The D scripting language which is derived from a large subset of C, is used to control tracing and to specify probe points in the kernel. It can also be used to define the actions to take whenever each probe is hit.

A D program can access types and symbols from the kernel and modules which could be very useful for run-time analysis and to generate warnings when an unexpected condition

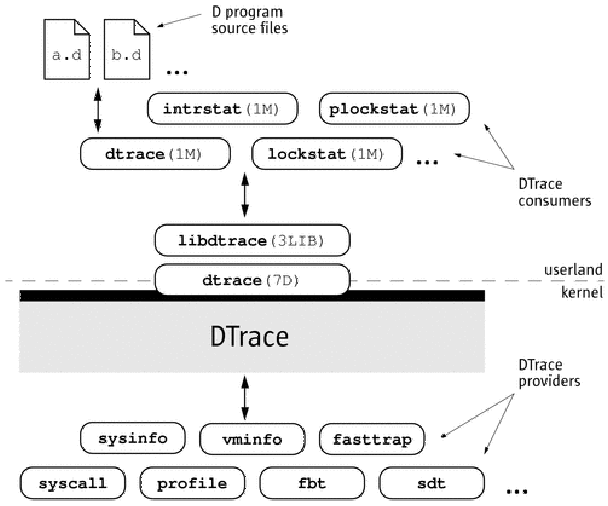

Figure 2.3 DTrace Architecture

takes place. The D script file is translated into C code and then compiled into a binary kernel module. More details about the D language can be found in the previous chapter. We show in Figure 2.3 an overview of the overall DTrace System Architecture. The lower part of the figure shows the modules residing in the kernel space that are called the DTrace providers. These providers are capable of instrumenting different parts of the kernel. All the user-space processes that use the dtrace library are called consumers. They are depicted in the upper part of the figure. There are many details concerning the dynamic probe insertion in the kernel as well as the events buffering but they are outside of the scope of this work.

2.5.1 Discussion

Online trace analysis is possible using DTrace. It provides a mechanism to register call back function for every event of interest, using the D language. These functions, called at runtime, may impact the overall system performance, and thus they should be very carefully implemented. Using DTrace, one could easily interfere with the normal execution of the kernel to achieve the desired behavior. While this approach is interesting, it

does not allow off-line analysis, and any attempt to perform complex analysis at run-time may considerably slow down the system.

2.6 Discussion

We have seen in this chapter the different architectures used by some popular trace analysis tools. A common adopted approach was the definition of a normalized event format in order to support different trace formats once the conversion is performed. Furthermore, we noticed that the majority of the tools came up with their own Domain Specific Languages to control and analyze traces, such as DTrace, SystemTap, ASAX and STATL. While introducing a new language presents a number of advantages for one particular tool, it incurs an additional maintenance cost that we are trying to avoid. In addition, we were not able to find a common pattern description language adopted by a majority of the tools. Table 2.1 summarizes the different features of the studied tools:

- Static instrumentation: consists in modifying the source code of a system and recompiling it so that tracing it is possible.
- Dynamic instrumentation: consists in adding tracepoints dynamically into a running system with no source code modifications.
- Normalized events: consists in the usage of software modules dedicated to normalize a trace format.
- Random access: consists in the ability to randomly access any event in the trace in contrast with the sequential read of the trace.
- Synthetic events: consists in supporting the definition of higher level events composed of multiple primitive ones.

Table 2.1 Summary Table.

	EXPERT	Pattern Mining	ASAX	STAT	DTrace
Static Instrumentation	✓	✓	✓	✓	
Dynamic Instrumentation					✓
Normalized event	✓		✓	✓	
Random Access	✓				
Synthetic Events	✓				✓
Domain Specific Language			✓	✓	✓
General Purpose Language	✓				
Automata-Based Language				✓	
Predefined Behavior Detection	✓		✓	✓	✓
New Behavior Detection		✓	✓		
Offline Analysis	✓	✓	✓	✓	
Online Analysis				✓	✓

- Domain Specific Language: consists in using a DSL language for trace analysis as defined in section 1.2.1.
- General Purpose Language: consists in using a general purpose language for trace analysis as defined in section 1.2.3.
- Automata-Based Language: consists in using an automata-based language for trace analysis as defined in section 1.2.4.
- Predefined behavior detection: consists in detecting a well definied problematic behavior by going over the trace.
- New behavior detection: consists in the ability to detect new behavior that deviates from the normal, profiled behavior.
- Offline analysis: consists in performing all the analysis offline, after the trace has been generated completely.
- Online analysis: consists in performing runtime validation of the trace.

In terms of architecture, the tools do share some parts in common. A tracing component collects the required events and saves them in a trace. The trace is processed by an analysis component which registers call back functions for every event of interest. This component is able to perform different kinds of analysis based on the description implemented by its tool's domain specific language.

CHAPTER 3

ARTICLE: SCENARIO-BASED APPROACH FOR KERNEL TRACE ANALYSIS

3.1 Abstract

Performance bottlenecks, malicious activities, programming bugs and other kinds of problematic behavior could be accurately detected on production systems if the relevant events were being monitored and logged. This could be achieved through kernel level tracing where every time a relevant event occurs, the information is saved in a trace file to be inspected during post-mortem analysis. While collecting the information from the kernel has a very low impact, the offline analysis is typically performed remotely with no overhead on the system whatsoever.

This article presents an automata-based approach for analyzing traces generated by the kernel of an operating system. Some typical patterns of problematic behavior are identified and described using the State Machine Language. These patterns are fed into an offline analyzer which efficiently and simultaneously checks for their occurrences even in traces of several gigabytes. The analyzer achieves a linear performance with respect to the trace size. The remaining factors impacting its performance are also discussed. As far as we know, this is the first work that targets patterns description for offline kernel trace analysis.

3.2 Introduction

By carefully examining execution traces of a computer system, experts can detect problematic behavior caused by software design defects, inefficiencies as well as malicious activities. Kernel tracing can often reveal the main source of such problems. Tracing consists in instrumenting the kernel code to precisely record its behavior at execution time.

It is now possible to achieve low overhead, low disturbance tracing of multi-core Linux systems with the Linux Trace Toolkit next generation (LTTng). It provides precise, low impact, highly reentrant tracing and is used for efficiently debugging large clusters (Bligh et al., 2007) as well as narrowing time constraints problems in real-time embedded applications (Desnoyers and Dagenais, 2006b). The information about the filesystem, inter-process communication, system calls, memory management and networking is efficiently collected, precisely time stamped and saved at runtime. This information is used to debug the monitored system and a large class of problems may be detected, such as excessive disk swapping, excessive threads migration, frequent writes of small data chunks to disk, locking problems, security problems and many others. Once the execution trace is available, the objective is thus to automatically validate it against a pool of predefined problematic patterns.

3.2.1 Motivations and Goals

The most popular kernel trace analysis tools that help simplify the debugging task provide offline event filtering and trace visualization. These tools include LTTV (LTTng, 2009), QNX Momentics (QNX, 2009) and Windriver Workbench (Windriver., 2009). Offline filters are used to highlight the events of interest satisfying a set of constraints. Visualizers, such as the Gantt chart of the control flow view (e.g. LTTV (LTTng, 2009)),

help the developer seek throughout the trace and determine visually any sort of unexpected behavior.

Even when these tools are used, validating the existence of a set of problematic patterns in one or several large traces remains a manual and time consuming task. This motivated the development of an automated approach to represent patterns of problematic behavior and to automatically and simultaneously check for their existence in one or several large traces.

3.2.2 Related Work

Frequent pattern mining for kernel trace data (LaRosa et al., 2008) is a recent work aiming at the detection of recurring runtime execution patterns, such as inter-process communication patterns. The work finds the set of all temporally proximal events that occured frequently in a trace. This helped identify the processes that are heavy consumers of system resources but still remain invisible to traditional tools such as top. This approach is interesting but doesn't allow validating the trace against a set of predefined patterns which may occur very rarely in the trace.

Systemtap (Eigler, 2006) and DTrace (Cantrill et al., 2004) provide scripting languages resembling C that are used to enable probe points in the kernel (instrumentation sites) and to implement their associated handlers. These handlers could be used to perform run-time checking and to generate warnings when something bad happens. The script file is translated into C code and then compiled into a binary kernel module. While this approach is interesting, it does not allow off-line analysis, and any attempt to perform complex analysis at run-time may considerably slow down the system. Furthermore, a C-like language does not provide a simple way to describe complex patterns at a high level of abstraction.

In parallel computing, many tools exist that are able to automatically detect performance problems in MPI, OpenMP or hybrid applications. These tools include Paradyn (Miller et al., 1995) and EXPERT (Wolf et al., 2004). EXPERT instruments the application's source code so that a trace file in the EPILOG format is generated upon running the program. The performance patterns are supplied to the tool and are written as Python classes implementing a common interface, making them exchangeable from the perspective of the tool. These pattern classes register callback functions for every event of interest and are capable of accessing additional events by retrieving the updated state information or by following some event dependencies. The LTTng Viewer also maintains an updated system state information as kernel events are processed. We consider implementing a similar approach for kernel traces using the State Machine Language to describe different kinds of problematic patterns at a higher level of abstraction.

Using Finite State Machines to describe patterns is found in the field of network based Intrusion Detection, particularly in the misuse detection systems or scenario-based systems. The State Transition Analysis Technique (STAT) (Eckmann et al., 2002), developed at the university of Santa Barbara, is used to model computer penetrations using Finite State Machines (FSM) called attack scenarios. Each scenario is composed of states and transitions. Transitions are triggered by the occurrence of particular events on the network and can take the system from an initial safe state to a final compromised state. The main features of the STAT language such as transition guards and actions are also found in the State Machine Language (Compiler, 2009) which we will be using in our work because of its open-source implementation.

By the time the article (Eckmann et al., 2002) was written, around 35 attack scenarios were decribed using the STAT language, and the authors claim that no limit in the expressiveness of the language was found. Furthermore, a recent work has been done to automatically translate the large collection of rules written for SNORT - a popular intrusion detection system - into STATL scnearios. We believe the STAT approach could

be very well applicable to kernel traces for many reasons. First the objective is quite similar; in both cases, patterns are composed of a sequence of events that could be used to describe either security threats or performance problems. Second, the automata-based approach provides an easy way to describe complex patterns from multiple simple ones, through the creation of synthetic events.

Ragel (RAGEL, 2009) is a popular state machine compiler used mainly to generate lexical analyzers and to validate user input. The generated code tries to match patterns to the input, favouring longer patterns over shorter ones. The Ragel language provides four types of transition actions. They allow the FSM developer to execute a particular action whenever the state machine transitions from one state to another. However none of the provided actions allows us to explicitly assign different actions for different transitions, from any random state in the FSM.

3.3 Faulty Behavior

While the system will be easily extensible at a later time, it was important to start by collecting a large representative set of problematic patterns touching on several fields such as security, software testing and performance debugging. For sake of brevity, a representative subset is described here.

3.3.1 Security

The SYN flood attack is a denial of service attack that consists in flooding a server with half-open TCP connections. Signs of a SYN flood attack may be found in a kernel trace if the relevant events are instrumented. It would be very inefficient to manually look for patterns caused by such an attack, thus the interest in automating the lookup process.

Escaping the chroot jail is another attack type that can be caught on a system: a privileged process (euid=0) may want to confine its access to a subtree of the filesystem by calling the chroot() system call. If this process ever tries to open a file after the call to chroot(), without a chdir(), then this is considered to be a security violation (Chen et al., 2004). Indeed, a malicious user can trick the program to open the system file ../../../../../etc/shadow for example. The right way to proceed would be to call chdir("/") right after the call to chroot() preventing the user from ever escaping the chroot jail.

Even though they are rare, Linux viruses do exist and they could be caught on a traced system. The approach we propose is different from the ones used in anti-virus software. Whenever a new virus is discovered, it could be executed on a virtual machine running an instrumented Linux kernel to record all its interactions with the operating system. This interaction, consisting in a sequence of system calls, identifies the behavior of that particular virus and we can look for it when analyzing traces generated from production systems. For example, in (Desnoyers and Dagenais, 2006a), the virus Linux.RST.B was observed generating the following actions: it executes a temporary file ".para.tmp" which creates three other processes; It opens and lists the current directory and modifies the binary files in /bin. By analyzing a kernel trace, it should be possible to detect a viral behavior automatically while diagnosing at the same time, some other security and performance problems.

3.3.2 Software Testing

Shared resources often require locks to be held before accessing them, to avoid race conditions. In the Linux kernel, locking is more complex than in user-space, due to the different states the kernel could be in (preemption enabled, disabled, servicing an irq, etc.). Validating each and every lock acquire has already been implemented in lockdep, the Linux kernel lock validator (Validator.,). For instance, it makes sure at run-time that

any spinlock being acquired when interrupts are enabled has never been acquired previously in an interrupt handler. The reason is that the interrupt could happen at anytime, in particular when the spinlock is already held, deadlocking therefore the corresponding CPU. Activating this option requires recompiling the kernel and adds a continuous slight overhead on the system. Instead, using a kernel trace and a posteriori analysis, the same kind of validations may be performed.

Another detectable programming bug consists in accessing a file descriptor after it was closed. This illustrates a more general class of programming errors where the usage specifications state that two particular events are logically and temporally connected.

3.3.3 Performance Debugging

Some inefficiencies in software could be detected from I/O events. For instance, frequent writes of small data chunks to disk would impact the overall system performance and are to be avoided. Similarly, reading the data that was just written to disk, or reading twice the same data, or even overwriting the data that was just written are all signs of inefficiencies that are visible in a kernel trace.

Multimedia applications, and more generally soft real-time applications, are characterized by implicit temporal constraints that must be met to provide the desired QoS (Abeni et al., 2002). Assuming that tracing the kernel scheduler has a negligible impact on the system, we can verify that temporal constraints are satisfied for one or multiple real-time applications, and whenever they're not, we can show what the system was doing at that time.

3.4 Automata-Based Approach

We first describe in 3.4.1 the state machine language and we show how it was used to model the three following scenarios: chroot jail escape, locking validation and real-time constraints checking.

3.4.1 SM Language

Describing the various patterns using the SM Language (Compiler, 2009) is straightforward. Even though many existing languages are capable of expressing the different scenarios described in section 3.3, a state-transition language was selected for the following reasons:

1. **Simplicity and expressiveness:** the language is easy to use and provides enough features to express new, yet to be defined, scenarios (Eckmann et al., 2002).

2. **Domain independent:** the language may be tailored to support a wide range of patterns that relate to different fields. In the Intrusion Detection field, state-transition language is widely used to model attack signatures (Vigna et al., 2000), (Eckmann et al., 2002). In model checking and Software Security, it is equally used for scenario-oriented modeling to examine security properties (Chen and Wagner, 2002), (Christodorescu and Jha, 2003) or to verify and validate software use cases (AsmL (Barnett et al., 2003)).

3. **Synthetic events:** the state-transition approach lets us easily generate synthetic events from lower level primary events (Eckmann et al., 2002). Consider for instance the SYN flood attack detection. We first model a half-open TCP connection using the state machine shown in Figure 3.1. When the server receives a connection request, the system moves to state S1. The server sends the acknowledgment

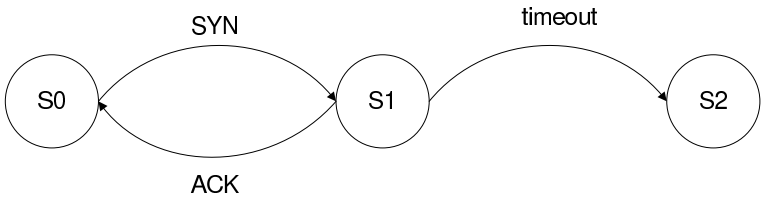

Figure 3.1 Detecting half-open TCP connections

and a timer is started. If the client sends back the acknowledgment, the system returns to state S0. Otherwise, when the timeout occurs, the system moves to S2 and a synthetic event is generated called “halfopentcp”. Frequent occurrences of this synthetic event would probably mean that an attack is taking place. Synthetic events are very useful when describing even more complex patterns.

The State Machine Language supports the declaration of a state and the transitions originating from it. Each transition has a name, an optional argument list, an optional transition guard, a destination state and a transition action. The guard is a boolean expression written in the target language source code and copied verbatim into the generated output. If the expression is evaluated to true then the transition is triggered and the transition action is executed. The destination state could be defined in another state machine declared in another file for simplicity. The transition actions are functions implemented in the target language and could have a regular argument list. Similarly, every state can have on-entry actions as well as on-exit actions than could be useful to start/stop a timer or update some internal data structures.

3.4.2 Escaping a chroot jail

An automaton showing the sequence of system calls that may result in a security violation is shown in Figure 3.2. The vulnerability is explained in 3.3.1. A call to chroot() brings the system to state S1 and saves the process id. Furthermore, a new FSM is forked

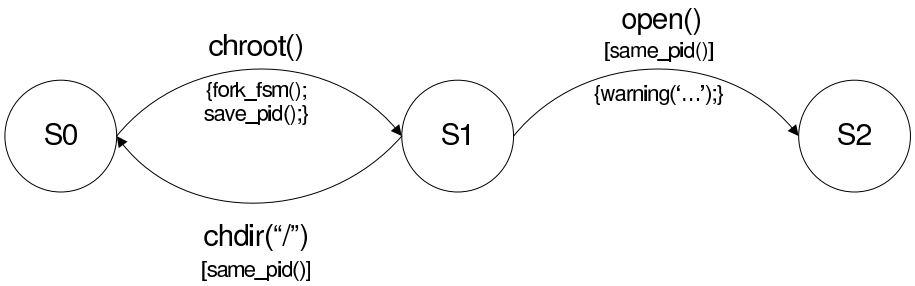

Figure 3.2 Escaping the chroot jail

```
S1{
chdir(pid: int, newdir: char *)
     [same_pid(pid) && check_new_dir(newdir)]
     S0
open(pid: int)
     [same_pid(pid)]
     S2
     {warning(pid); destroy_fsm();}
}
```

Table 3.1 SM Code Snippet

in case a new chroot() call is issued by another process. The FSM fork is initiated by the transition action fork_fsm(). Any process issuing a successive call to chdir("/"), brings back the corresponding FSM to state S0, whereas a call to open() brings it to S2 and generates a warning. The machine transitions to a fourth Exit state, not shown here, and it happens whenever the exit() call is issued by the process.

We show in table 3.1 a self explanatory code snippet of the language describing state S1 from Figure 3.2. From state S1, two transitions are possible, chdir() and open(). If the encountered event is a call to chdir, then the transition guard (between square brackets) is evaluated. In this case, if the functions same_pid() and check_new_dir() return true, then the transition is triggered and the system moves back to state S0. It is also possible to have a transition action (between braces). In our example, the call to the function warning() occurs only if the corresponding transition guard is evaluated to true.

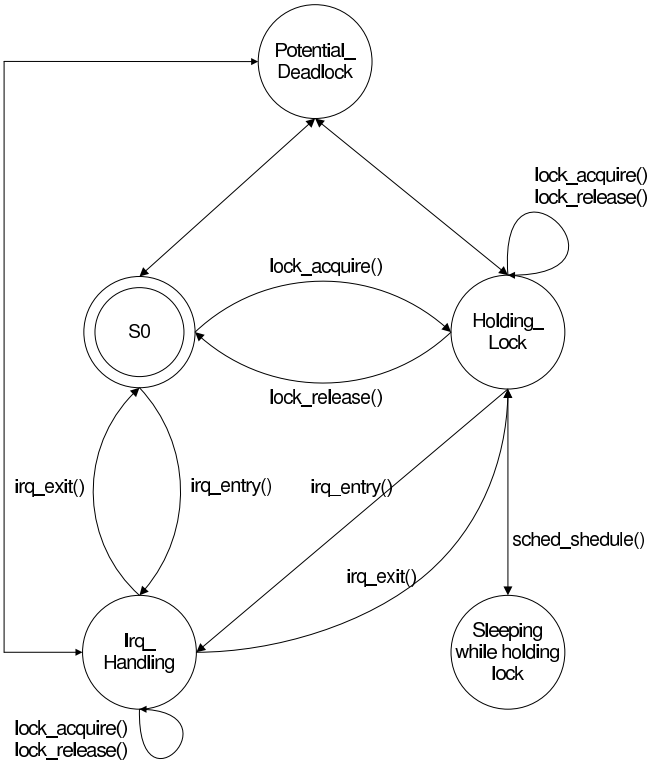

Figure 3.3 Locking Validation

3.4.3 Locking Validation

We generate in Figure 3.3 an automaton that will validate a subset of the kernel locking rules. The event irq_entry() brings the system to state Irq_Handling and event irq_exit() brings it back to its normal state. Any lock could either be acquired from the normal state (S0 or Holding_Lock) or the Irq_Handling state. If a lock being acquired when interrupts are enabled has been previously acquired from the Irq_Handling state, the system transitions to state Potential_Deadlock. The reason is that once this lock is taken and before it gets released, if the code is interrupted by the same handler which tries to acquire the same lock, then a deadlock occurs. Similarly, if a lock previously taken when irqs were on, is now being acquired from an irq handler, then the system should also transition to the state Potential_Deadlock.

Suppose the system is in state Holding_Lock on a particular processor, where a lock is being held on behalf of a certain process. If this process gets scheduled out, then there is another potential deadlock due to the fact that some other process may require the same lock.

Nested locks, taken on behalf of the same process could deadlock the system if they are not taken in the right order. When the system is in the state Holding_Lock, the arrival of a new event lock_acquire would trigger the corresponding transition. This results in a call to a function that generates trees of lock dependencies implemented in a hashing table. At the end of the analysis, if a cycle is found, then there is a potential deadlock and the involved locks are shown. The return address, which is a traced event argument, can help identify the code section responsible of holding the lock.

One interesting case was found in function copy_pte_range() in mm/memory.c which generated a cycle in our analysis. The suspicious code sequence that caused the problem is abstracted in Table 3.2. The function receives pointers to two mm_struct structures and always locks the destination page_table_lock spinlock, followed by the source lock. If another CPU is doing the copy but with the reversed parameters, then the locks would be taken in the opposite order and a deadlock can occur. After further investigation, we noticed that a call to this function is initiated by a call to copy_process() in fork.c which is called when forking a process. This function calls dup_mm() which allocates memory for a new mm_struct which becomes the dst_mm shown in Table 3.2. Since no other processor could be using the newly initialized structure as being the src_mm in function copy_pte_range(), there is no potential deadlock. However, this shows how our approach was useful to identify suspicious code sequences.

```
static int copy_pte_range(struct mm_struct *dst_mm, struct mm_struct
*src_mm, ...)
{
      ...
      spinlock_t *src_ptl, *dst_ptl;
      ...
      spin_lock(dst_ptl);
      ...
      spin_lock_nested(src_ptl, ...);
      ...
}
```

Table 3.2 Suspicious Code Sequence

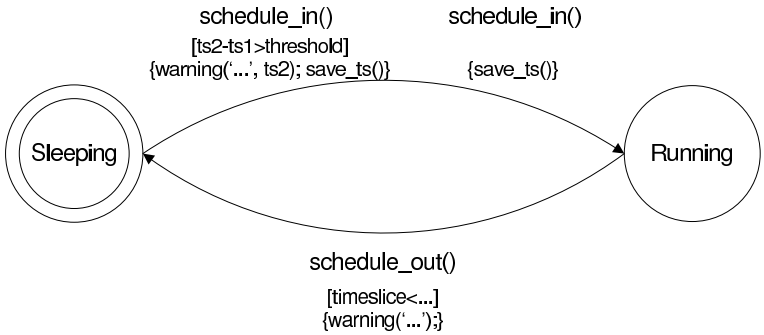

Figure 3.4 Real-Time Constraints Checking

3.4.4 Real-time Constraints Checking

To support soft real-time applications, the kernel should respect the application's temporal constraints and therefore a predictable schedule is desired (Abeni et al., 2002). Such applications may require periodic scheduling where the period is derived from the frame rate of an audio/video stream for example. We show in Figure 3.4 a detailed state machine that enables us to check if the application's execution period has been respected throughout the life of the trace. Whenever it's not, we show the list of events that hindered the application's scheduling.

From state Sleeping, the transition schedule_in() brings the FSM to the Running state and saves the event time stamp; it also computes the difference between every two con-

secutive schedule_in() events. If the result is greater than a user specified threshold, a warning is generated. The event time stamp displayed by the warning() call, can then be used to determine all the preceding events once the trace is opened using the Linux Trace Toolkit Viewer (LTTV). From the Running state, the event schedule_out() brings the FSM back to the Sleeping state. The time stamp of this event is also used to compute the assigned time slice for the application so that the transition could also trigger a warning when the time slice is less than expected.

3.5 Implementation

We used the Linux Trace Toolkit LTTng, a low-impact, open-source kernel tracer, to instrument the kernel events required by the patterns description. We used the SMC compiler to generate C code for the state machines written in the SM language. The compiler is an open-source java program that supports code generation in 14 different languages.

For every event required by a given pattern, the analyzer registers callback functions with the trace reader and visualizer program LTTV. The program reads the trace sequentially in one pass. When a registered event is encountered, the analyzer calls the corresponding transition for every related state machine. There, if the transition guard is evaluated to true, the transition action is executed before entering the destination state and returning control to the analyzer.

In some cases, when a transition is triggered, a new FSM of the same type needs to be forked. This is referred to as a non-consuming transition type in STATL terminology (see example in 3.4.2). Whenever required, a transition action can request a fork from the analyzer, generating therefore a new instance of the FSM.

In other cases, such as the locking validation pattern, one finite state machine per CPU is enough. There, the analyzer determines on which CPU the event occurred, and only calls the transition of the FSM for that particular CPU.

The FSM approach offers great flexibility to model, update and optimize one or several patterns. When we instrumented the events of interest for the locking validation pattern, we noticed that the irq entry and exit events are not needed because the information could be determined from the lock_acquire() event. At this point, we simply eliminated the Irq_Handling state from our FSM.

We study the performance of the analyzer and we compare the performance of the automata-based approach with that of a dedicated implementation in section 3.6. The dedicated implementation is implemented in C and statically linked to the analyzer. Its internal data structures are updated upon encountering relevant events without any FSM invocation.

3.6 Performance

We instrumented the Linux kernel version 2.6.26 using LTTng and the tests were performed on a Pentium 4 with 512 MB of RAM. In table 3.3 we show the execution time of our analyzer to look up 3 different patterns: real-time constraints, file descriptors and the chroot patterns. Our results show that the execution time is linear with respect to the trace size. In fact, the performance of the analyzer depends on three other factors: the number of coexisting finite state machines, their complexity (i.e. memory usage per state and transition) and the frequency of relevant events in the trace triggering a transition. The number of coexisting finite state machines depends on the pattern in question. For instance, checking the file descriptors usage required one FSM per process accessing one file descriptor, whereas for the real-time checking we used just one FSM for the Movie

	500 MB	1GB	1.5GB	2GB
rt checking	55s	117s	168s	252s
fd_checking	57s	119s	166s	266s
chroot_checking	55s	108s	166s	266s
all	67s	123s	184s	279s

Table 3.3 Performance Results

Player (mplayer) process. We obtained similar execution times due to the fact that event sched_schedule() was occurring more frequently than events read() and write().

Another factor impacting the performance of the analyzer is the following, consider the locking validation pattern in Figure 3.3. Even when the current FSM state is S0, every encountered sched_schedule() event would result in calling the corresponding transition which is irrelevant in state S0. This will call a default transition which maintains the current state and returns control to the analyzer. Instead, the analyzer could have skipped this step since, from the current state, there is no transition sensitive to the event sched_schedule().

The performance of the FSM checker was 4.5% slower than the dedicated version when validating the locking pattern. This was expected because it dealt with frequently occurring events leading to frequent transitions between states. However, our approach is more generic and the overhead is often acceptable in offline analysis.

We show in Table 3.4 the performance of the analyzer when validating the file descriptor pattern against traces of different sizes, and compare it with the analyzer's performance without invoking the FSMs, but only registering empty callback functions for the 6 events of interest. They are the following system calls: close(), open(), read(), write() and dup(), as well as the process_exit() kernel event. The slowdown is computed by comparing the execution time between the two configurations of the analyzer. The first 4 traces shown in the table were taken while running 1 dbench client, while the last trace was taken while running the full compilation of gcc v.4.2.0. The analysis time increases

		relevant events (millions)	coexisting FSMs	Ex. time invoking FSMs	Ex. time empty callbacks	**slowdown**
1 dbench client	500 MB	2.4	75	51s	50s	6.00%
	1 GB	**4.8**	**72**	**92s**	**86s**	**6.98%**
	1.5 GB	6.9	104	143s	114s	25.44%
	2.3 GB	**11.1**	**72**	**215s**	**189s**	**13.75%**
	3 GB	14.1	104	285s	250s	14.00%
	4.5GB	**18.5**	**83**	**369s**	**338s**	**8.87%**
gcc	2.5 GB	5.7	5241	853	227s	275.77%

Table 3.4 Slowdown of the analyzer due to FSM invocation with respect to its performance with empty callbacks

non-linearly with respect to the trace size for the two different tests. The slowdown was much higher for the gcc trace, having fewer relevant events than 2 of the other smaller traces. The computed slowdown suggests a direct correlation with the maximum number of coexisting finite state machines handled by the analyzer. For instance, the gcc compilation generated much more coexisting FSMs than running one dbench client due to the numerous processes (accessing different file descriptors) generated by the makefile. This resulted in a larger impact on the analyzer's performance.

We fixed the trace size to 500 MB and varied the number of dbench clients. The number of clients is directly proportional to the maximum number of coexisting FSMs during the analysis. The results in Figure 3.5 show the slowdown percentage with respect to the maximum number of coexising FSMs in the analyzer, for traces of the same size. The slowdown is directly proportional to the maximum number of FSMs handled by the analyzer. This is expected because the analyzer invokes sequentially all the FSMs in the list for every relevant event, whether the event is needed at the FSM's current state or not.

By carefully selecting which events to trace, it may be possible to reduce the execution time even further. For instance, the first version of the locking validation pattern required the events enable_irq() and disable_irq() to deduce in which context a given lock was

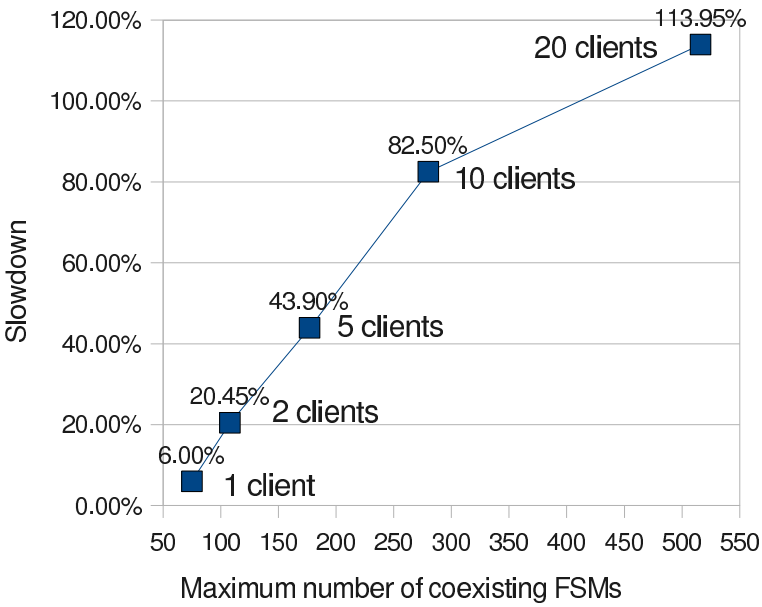

Figure 3.5 Fixing trace size to 500MB, varying the number of dbench clients

acquired. It turned out that this information is available at the site where the lock is being acquired. This reduced the number of events to trace resulting in a smaller trace and a faster analysis.

3.7 Conclusion

We presented an automata-based approach to represent some generic patterns of problematic behavior that might occur on production systems. The generated finite state machines can be easily maintained, expanded or even be used as synthetic events to model more complex scenarios. We implemented an analyzer that validates the existence of such patterns simultaneously in large traces and in one pass.

When dealing with a large number of FSM instances of the same pattern, the analyzer suffers a slowdown directly proportional to the number of coexisting FSMs. The dedicated approach for the locking pattern was only 4.5% faster than the automata-based

approach suggesting that the overhead of using finite state machines for execution, and not just for modeling, is acceptable especially in post-mortem analysis.

3.7.1 Future Work

One problem with our implementation is the fact that for a given scenario, a relevant event will trigger a function call whether the event is needed in the scenario's current state or not. This could be avoided if the analyzer knows beforehand the list of states for which at least one transition is sensitive given the current states. Ideally, the analyzer should only invoke the state machine when the event could trigger a transition from the current state. Also, when validating a trace against a set of predefined patterns, the analyzer should drop the ones that require at least one event that is not traced, assuming that the information about all the traced events is present in the trace header.

Further explorations could be done to support the definition and use of synthetic events. This will allow synthesizing more complex scenarios from multiple simple ones.

CHAPTER 4

METHODOLOGICAL ASPECTS AND COMPLEMENTARY RESULTS

In this Chapter we briefly discuss the methodology for implementing and checking for the existence of a new pattern, using the automata-based approach. Then we provide some complementary results that were not included in the journal article.

4.1 General Methodology

The following steps illustrate how a new problematic behavior can be modeled using a finite state machine, and fed to the analyzer for validation.

1. **Problem definition:** The problem to be modeled needs to be clearly defined. It usually involves a sequence of events, not necessarily temporally proximal. The sequence of events satisfying certain conditions constitute the core of the problem. As an example, a problem could be defined as accessing a file descriptor after it was closed. Another problem would be defined as accessing the locks in a wrong order from two or more sites, which may result in a deadlock. The problem could be a security violation, a performance bottleneck or any other sort of undesired behavior. Once the problem is clearly defined, it is possible to proceed to the next step.

2. **Instrumentation:** Different problems may require different events to be monitored. Therefore, one has to carefully determine the instrumentation sites in the kernel in order to insert the desired tracepoints. In one of our implemented patterns, we wanted to trace when the interrupts are enabled or disabled. However,

it turned out that this information was available at the site where a lock is being acquired. Since the latter event, also relevant for our analysis, was able to provide us with the needed information about the interrupts status, we dropped the instrumentation of enabling and disabling the interrupts. If the instrumentation is static, i.e. the source code is modified, then kernel recompilation is required.

3. **Tracing the system:** The system should be set to run under regular load, while tracing is activated. If possible, all the necessary conditions that would make the system exhibit the undesired behavior should be provided. Ideally, only the relevant tracepoints should be activated along with context switching events and, of course, a system state dump at the start of tracing.

4. **Implementation:** Modeling the problem in a finite state machine is straight forward. As an example, the chroot jail escape shown in Chapter 4, illustrates how the different states and transitions of the finite state machine are described. The occurrence of a certain event in the trace, satisfying a certain condition, would typically trigger a transition in the finite state machine. For instance, the event irq_entry() would result in a transition towards the IRQ_handling state while the event irq_exit() would result in a transition leaving the IRQ_handling state.

5. **Guards and Actions:** Besides the high level of abstraction the FSMs offer, a major feature they provide is the ability to define transition guards and actions. The transition guard is a boolean expression in which all the necessary conditions for taking the transition are provided. When using the SMC compiler to generate C code from the modeled FSMs, transition guards are copied verbatim into the generated output. In other words, one can provide a boolean expression composed of one or more function calls as a transition guard. These functions need to be implemented in C and not in the SM language. This provides a great flexibility by being able to use all the power of C and not limit ourselves with a subset of C operators as was the case in other Domain Specific Languages such as the D or the

SystemTap languages. Similarly, transition actions need also to be implemented in the target source code. Their main purpose is threefold: first, they can be used to update the internal data structures and variables local to every state machine. For example, if one FSM per process is required, a variable pid would be required to identify the represented process. Second, transition actions can be used to perform all kinds of analysis based on the received event arguments. For instance, a transition action could compute and store the time spent in a particular state. Last but not least, transition actions are ideal to use the analyzer's API. This is described in the following step.

6. **Analyzer's API usage:** The analyzer offers an API which provides an added flexibility for patterns description. The API is typically used in the transition action. We have seen in Chapter 4 that multiple instances of the same pattern can coexist. For instance, when checking if no file descriptor was accessed after being closed, one pattern per process accessing a file descriptor was required. In order to do so, new FSMs need to be forked upon the occurrence of the close() event. This is achieved using the fork_FSM() API called from the transition action. Similarly, when the FSM reaches a state from which it cannot evolve anymore, it should be able to request its self destruction from the analyzer and therefore freeing the memory it consumed. This is achieved via the call destroy_FSM(). Other variants of this call could exist as well. For instance, whenever a particular event is encountered, an FSM could ask the analyzer to destroy all the instances of the same pattern type. Furthermore, the FSMs can ask the analyzer to log a warning message.

4.2 Performance Analysis and Scalability

In Chapter 4, we showed how the performance of the analyzer drops as the number of coexisting FSMs rises. This was mainly due to the fact that the analyzer needs to invoke all the forked FSMs for every encoutered event of interest. When the number of coexisting FSMs is relatively high, a frequent occurrence of relevant events resulted in a 275% slowdown. The pattern in question consisted in checking that no file descriptor has been accessed after being closed.

For this particular pattern, the number of coexisting FSMs during the analysis depends on the number of processes accessing a file descriptor. For instance, if 10 processes have closed 3 file descriptors, and never exited, a total of 30 FSMs would be handled simultaneously by the analyzer. Furthermore, if one event is consumed by one FSM, there is no need to keep on invoking the rest of the FSMs of the same type. The reason is simple: there is no two different FSMs for the same tuple (pid, fd) where pid is a process id and fd is the file descriptor the process is using.

4.2.1 Optimization

In many cases, an event would be consumed by at most one FSM. A simple but effective optimization mechanism is introduced to skip the invocation of the rest of the FSMs once the event is consumed by one of them. It consists of the analyzer's API skip_FSM() called from within a transition action code block. When this function returns, and control is transferred back to the analyzer, the analyzer breaks from the main loop which was invoking all the FSMs in turn, and forces it to jump to the next event.

To study the performance improvement, the following test is conducted. Five traces of 500MB were taken while running respectively for each, one, two, five, ten and twenty

Table 4.1 Table showing the slowdown obtained with respect to the analyzer's performance when it only registers empty callback functions for the relevant events.

	1 client	2 clients	5 clients	10 clients	20 clients
Exact trace size	554MB	510MB	501MB	502MB	504MB
Total # of coexisting FSMs	75	108	177	280	516
Total # of forked FSMs	1,006,761	1,026,601	1,009,000	983,949	954,760
Number of pertinent events	2,396,931	2,446,646	2,400,984	2,349,137	2,300,114
Ex. time (empty callbacks)	50s	44s	41s	40s	43s
Ex. time (invoking FSMs)	53s	53s	59s	73s	92s
slowdown	6.00%	20.45%	43.90%	82.50%	113.95%
Ex. time after optimization	51s	48s	49s	51s	60s
slowdown	2.00%	9.09%	19.51%	27.50%	39.53%

dbench clients. A higher number of dbench clients implies a larger number of coexisting FSMs handled simultaneously by the analyzer. In Table 4.1 we show a summary of the obtained results. First, we notice that the number of coexisting FSMs increases proportionally with the number of dbench clients. This is expected because each client is an additional process accessing multiple file descriptors. Second, the number of forked FSMs as well as the number of relevant events remain relatively constant for a fixed trace size.

Three different configurations of the analyzer are executed. The first consists in reading the full trace with empty callback functions for the six events of interest which are the following system calls: open(), close(), read(), write(), and dup(), as well as the process_exit() kernel event. For the remaining two configurations, the slowdown is computed with respect to the first one. The second configuration consists in invoking all the registered FSMs upon encountering a relevant event. The results are shown in Figure 4.1. The third configuration involved skipping unnecessary FSM invokations whenever possible. This is possible since, in this particular pattern, any event cannot be consumed by more than one FSM. Therefore, whenever the invoked FSM consumes the event, it notifies the analyzer via the skip_FSM() call. The analyzer will thus stop invoking any additional FSM and will skip to the next event. This greatly improved the analyzer's performance. The slowdown percentages are shown in Figure 4.2.

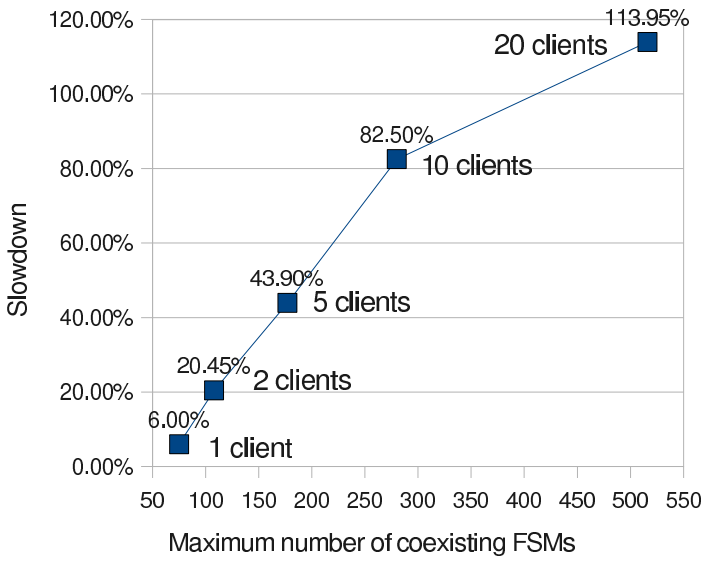

Figure 4.1 Fixing trace size to 500 MB, varying the number of dbench clients.

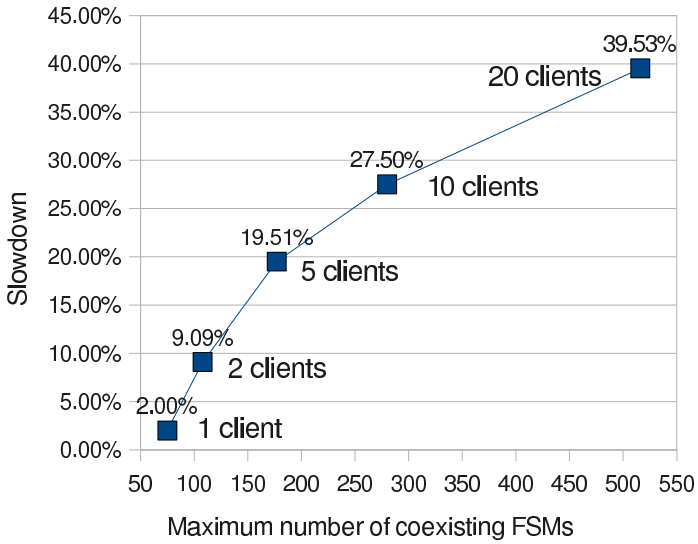

Figure 4.2 Fixing trace size to 500 MB, varying the number of dbench clients and skipping unnecessary FSM invocations.

CHAPTER 5

GENERAL DISCUSSION

In this chapter we discuss some of the advantages our solution provides. Then, we describe how flexible and scalable the approach is.

5.1 Current Limitations

As seen in Chapter 2, the available trace analysis tools do not provide a way to describe patterns and validate their existence in a kernel trace. They do provide, however, high level views of the trace enabling experts to visually identify erroneous behavior such as the disactivation of interrupts for a relatively long period. They also provide domain specific C-like languages that are used to implement the callback functions for every registered event. These functions are typically used to print some warnings, log some information or compute some statistics. They do not provide an easy way to describe complex patterns that could be composed of multiple simple ones, nor do they support managing multiple coexisting instances of the same pattern. For instance, the programmer would still have to manually encode the pattern in question across the different probe handlers, using an imperative C-like language.

5.2 Proposed Solution

Our proposed solution has many advantages:

5.2.1 High-level Pattern Description

The automata-based approach provides a high level of abstraction for patterns description, resulting in a more compact and simple pattern definition. This relieves the programmer from the burden of dealing with implementation details of the patterns, especially when multiple instances of the same pattern could coexist. FSMs are therefore used not only for modeling the problem but also for execution. While the automata-based approach offers great flexibility for describing patterns, the analyzer's API provides very useful features such as FSM forking and destruction, or skipping unnecessary FSM invokations whenever an event can be consumed by only one pattern instance.

5.2.2 Exhaustive Trace Analysis

The analysis framework we propose enables field experts to share their knowledge by writing as many patterns as they want based on their experience, to form a pool of patterns. By having a large pool of problematic patterns to be validated automatically and simultaneously, we maximize our chances of finding one or more problems occurring on traced production systems.

5.2.3 Offline/Online Analysis

Eventhough our main concern was more oriented towards offline trace analysis, our solution could be easily ported to run online, on production systems. The impact greatly depends on the pattern being validated and therefore one should carefully decide on the nature of the patterns used for online validation.

5.2.4 Maintenance

The proposed solution encourages the maintenance and optimization of pattern descriptions to maximize the number of problems we would want to detect from a trace. Furthermore, there are no maintenance costs for the State Machine Language which is an open-source project, unlike the Domain Specific Languages we studied in Chapter 1.

5.2.5 Parallelization

The proposed solution can be easily parallelizable on multiple levels:

5.2.5.1 Multiple Instances

An important factor to consider is that two or more instances of the same pattern are sensitive on the same events. Very often, no execution order is required when invoking multiple FSMs of the same type. They can therefore be executed in parallel among multiple processors.

5.2.5.2 Multiple Distinct Patterns

Distinct patterns targetting different problems, may also run in parallel. For instance, if pattern A and pattern B require same events but for different kinds of analysis, parallelization would result in a speedup. When A and B's common events are encountered, both patterns could be processed in parallel. However, if A and B don't share any common events of interest, parallelization is less convenient; for a given event, either A or B will be processed but not both.

5.2.5.3 Sequential Ordering

The only case where parallelization is non convenient, occurs whenever a complex pattern is composed of multiple simple ones and, therefore, an evaluation order is required. There, the simple patterns should be evaluated first, generating synthetic events, which are then be consumed by the higher level complex pattern.

5.3 Promising Results

We showed in chapters 4 and 5 that the analyzer exhibits a slowdown proportional to the number of coexisting FSMs. However, this slowdown could be minimized in certain cases, depending on the nature of the pattern in question. This was shown in section 4.2.1. Whenever one or very few coexisting instances of a pattern are required throughout the analysis, the analyzer's performance would scale up linearly with respect to the trace size. Furthermore, the results shown in the previous chapter suggest a 20 microsecond processing time per relevant event. Therefore, whenever the frequence of relevant events occurring on a live system is less than 50kHz, non-fully-exploited CPUs under production could very well be used for online pattern matching.

CONCLUSION

In this work, a solution was proposed for pattern matching in large kernel traces. We started by collecting a set of typical problems that could occur on production systems. This enabled us to create a list of requirements for a pattern description language. Then, we studied the different trace analysis tools. A large majority of these tools provide their own domain specific languages which are not always adapted for pattern description. We were able to group the languages used by these tools into three different categories: Domain Specific Languages, General Purpose Languages and Automata-based Languages. We presented the pros and cons of each and every category. The automata-based languages fulfilled the pattern description requirements we came up with.

A number of patterns were actually implemented using the selected state-machine language. They include:

- **Locking Validation:** This pattern validates a subset of the kernel locking rules. The details are found in Chapter 4. This pattern enabled us to find a suspicious code sequence in the Linux kernel which could generate a deadlock.
- **Real-time constraints checking:** This pattern verifies that the real-time constraints of a certain application were being respected throughout the trace. Metrics such as the scheduling frequency as well as the allocated time slice are computed by the FSM so that a warning is generated whenever an unexpected condition occurs.
- **Escaping the chroot jail:** This pattern makes sure that no privileged process erroneously calls the chroot() system call without an immediate subsequent chdir() call.
- **Acessing a closed fd:** This pattern checks that no process has ever accessed a

closed file descriptor throughout the trace.

The analyzer's performance depends greatly on the nature of the patterns being validated. For patterns that do not require multiple self-instances to be generated during validation, the analysis time is almost equivalent to reading the trace which is directly proportional to the trace size. However, for patterns that do require self-instances to be generated, the analyzer exhibits a higher slowdown as the number of coexisting FSMs increases. This is mainly due to the fact that the analyzer will have to iterate over all the coexisting FSMs, invoking them one after the other for every relevant event. For certain cases, where the event could be consumed by at most one FSM, we were able to reduce the execution time of the analyzer by a considerable factor. We also compared the performance of the automata-based approach with that of a dedicated approach for the locking validation pattern. The dedicated approach is implemented in C and its internal data structures are updated upon encountering relevant events without any FSM invocation. The dedicated approach was only 4.5% faster than the automata-based approach, suggesting that the additional overhead of using finite state machines for execution, and not just for modeling is acceptable especially in post-mortem analysis.

We argue that the solution we propose is highly parallelizable and may be very well ported for online pattern matching. We also believe that the generated FSMs can be easily maintained, expanded, or even be used to generate synthetic events to model more complex scenarios.

Future Work Further explorations could be done to support handling synthetic events. The main objective would be to allow the representation of high-level compound events, composed of multiple primitive ones, so that they could be used to describe more complex patterns efficiently and modularly.

One limitation in our implementation is the fact that the analyzer maps a given pattern

to the list of the events it requires regardless of the FSM's current state. For instance, a relevant event for the pattern will trigger a function call, whether the event is required in the FSM's current state or not. This could be avoided if the analyzer knows beforehand the list of states from which at least one transition is sensitive on that event of interest. Ideally, the analyzer should only invoke the state machine when the event could trigger a transition from its current state.

The solution we proposed is highly parallelizable and we would like to study the resulting speedup in an attempt to perform the analysis online.

REFERENCES

Abeni, L., Goel, A., Krasic, C., Snow, J., and Walpole, J. (2002). A measurement-based analysis of the real-time performance of linux. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium.*

Barnett, M., Girieskamp, W., and Gurevich, Y. (2003). Scenario-oriented modeling in asml and its intstrumentation for testing. In *Foundations of Software Engineering.*

Bligh, M., Desnoyers, M., and Schultz, R. (2007). Linux kernel debugging on google-sized clusters. In *Proceedings of the Linux Symposium.*

Cantrill, B., Shapiro, M., and Leventhal, A. (2004). Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference.*

Chen, H., Dean, D., and Wagner, D. (2004). Model checking one million lines of c code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium.*

Chen, H. and Wagner, D. (2002). Mops: an infrastucture for examining security properties of software. In *Proceedings of the ACM Computer and Communications Security Conference.*

Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium.*

Compiler, T. S. M. (2009). http://smc.sourceforge.net. Retrieved on 2009-01-22.

Desnoyers, M. and Dagenais, M. (2006a). Tracing for hardware, driver, and binary reverse engineering in linux. *CodeBreakers Journal.*

Desnoyers, M. and Dagenais, M. R. (2006b). Low disturbance embedded system tracing with linux trace toolkit next generation. In *Embedded Linux Conference 2006.*

DTrace. (2009). http://wikis.sun.com/display/DTrace/Variables. Retrieved on 2009-03-05.

Eckmann, S. T., Vigna, G., and Kemmerer, R. A. (2002). Statl: An attack language for state-based intrusion detection. *Journal of Computer Security*, pages 71–103.

Eigler, F. C. (2006). Problem solving with systemtap. In *Ottawa Linux Symposium*.

Habra, N., Charlier, B. L., Mounji, A., and Mathieu, I. Asax: Software architecture and rule-based language for universal audit trail analysis. *Computer Security*, pages 435–450.

LaRosa, C., Xiong, L., and Mandelberg, K. (2008). Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing*.

LTTng (2009). http://lttng.org. Retrieved on 2009-03-10.

Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irving, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. (1995). The paradyn parallel performance measurement tool. In *IEEE Computer magazine*.

QNX (2009). http://www.qnx.com. Retrieved on 2009-03-12.

RAGEL (2009). http://www.complang.org/ragel. Retrieved on 2009-03-07.

SNORT (2009). http://www.snort.org/docs. Retrieved on 2009-03-07.

Validator., T. K. L. http://lwn.net/Articles/185666. Retrieved on 2009-03-10.

Vigna, G., Eckmann, S. T., and Kemmerer, R. A. (2000). The stat tool suite. In *DARPA Information Survivability Conference & Exposition*.

Windriver. (2009). http://www.windriver.com/products/workbench. Retrieved on 2009-03-12.

Wolf, F., Mohr, B., Dongarra, J., and Moore, S. (2004). Efficient pattern search in large traces through successive refinement. In *Lecture Notes in Computer Science*.