Apprentissage d'un modèle de comportement d'une application distribuée pour la détection d'intrusion

David Lanoë*†, Eric Totel† and Michel Hurfin*
*Univ Rennes, Inria, CNRS, IRISA david.lanoe@inria.fr michel.hurfin@inria.fr
†CentraleSupélec, Inria, CNRS, IRISA eric.totel@centralesupelec.fr

***Résumé*—Les systèmes d'information hébergeant des applications distribuées sont de plus en plus courants. Les systèmes de détection d'intrusion ont besoin de suivre cette évolution pour détecter efficacement les attaques. Actuellement, les approches de détection d'intrusion classiques reposent sur l'hypothèse d'un ordonnancement total des évènements observés. Cependant, cette hypothèse est souvent trop forte dans le cas d'environnements distribués où l'ordre d'observation des évènements est partiellement inconnu. Cet article traite de l'apprentissage d'un modèle de comportement hybride d'une application distribuée pour la détection. Il décrit des étapes et choix possibles lors de la construction du modèle. Enfin, une évaluation de l'impact de certains de ces choix est effectuée.**

I. Introduction

Les systèmes distribués de grande taille, comme le cloud, hébergent des applications distribuées qui peuvent être très sensibles (ex : application de e-commerce, système de fichier distribué, ...). La détection d'intrusion dans un système distribué consiste essentiellement en la surveillance locale des processus ou du réseau. Cette détection locale engendre des alertes qui sont remontées au corrélateur qui a pour but de réduire le nombre d'alertes et de les rendre plus pertinentes. Cependant, cette corrélation nécessite une connaissance préalable des scénarios d'attaque et d'un ordonnancement total des évènements pour la reconnaissance d'un scénario. Dans cet article, nous étudions l'apprentissage d'un modèle de comportement d'une application distribuée pour la détection d'intrusion. L'objectif d'une telle approche est de lever une alerte lorsque le comportement de l'application surveillée ne correspond pas au modèle de référence. Dans un premier temps, nous détaillons les étapes et quelques choix possibles pour construire notre modèle de comportement. Dans un second temps, nous évaluons les impacts de ces choix sur la précision des modèles et sur le processus de détection. Cet article est structuré comme suit : après un bref état de l'art dans la section II, nous introduisons quelques concepts clés et les choix possibles dans l'approche de modélisation retenue dans la section III. Ensuite, nous détaillons le processus de détection d'intrusion dans la section IV. Puis, nous évaluons certains choix possibles dans notre approche à l'aide d'une mesure des faux positifs et faux négatifs. Enfin, nous concluons en proposant des pistes sur les travaux futurs dans la section VI.

II. État de l'art

Les approches traditionnelles de détection d'intrusion dans les systèmes distribués, qu'elles soient comportementales ou par reconnaissance de scénario se basent sur l'hypothèse d'une horloge globale. L'horloge commune permet de dater les occurrences d'évènements et de déterminer l'ordre total dans lequel ils se sont produits et ce même lorsqu'ils se sont produits sur des machines différentes. Cependant, dans un système distribué, l'hypothèse de l'existence d'une horloge globale est souvent trop forte. Lamport [1] introduit une notion d'horloge logique en proposant un ordonnancement partiel des évènements observés grâce à une relation « happening before ». De cet ordre partiel on peut déduire un ensemble d'ordres totaux possibles incluant celui correspondant à l'exécution.

Dans le domaine de l'apprentissage et la modélisation d'un comportement, certains travaux utilisent des logs générés par des applications pour inférer un modèle de comportement sous forme d'automates [2][3][4] ou d'invariants temporels [5]. D'autres approches se basent sur l'inférence d'un modèle à l'aide de logs provenant d'une application distribuée [6][7]. Ces approches rencontrent des difficultés liées à l'apprentissage et à la modélisation de tous les comportements possibles et légitimes d'une application.

Dans l'optique d'introduire de nouveaux comportements légitimes, des techniques de généralisation [8][9][10] visent à la fois à réduire la taille d'un automate et à introduire de nouveaux comportements non appris. De fait, l'automate généralisé accepte plus de comportements que l'automate d'origine. Parmi les nouveaux comportements, certains peuvent être légitimes et d'autres erronés. Selon les algorithmes de généralisation utilisés, le modèle obtenu peut fortement varier. Notre but est de proposer différentes approches de modélisation de comportements pour la détection d'intrusions et d'évaluer la précision de ces approches.

III. Construction du modèle de comportement

Notre approche vise à construire un modèle de comportement d'une application distribuée contenant n processus (p_1, p_2, ..., p_n). Lors de l'apprentissage, l'application est exécutée un nombre fini de fois dans un environnement sain (sans violation de l'intégrité de son flot de contrôle).

A. De la trace d'exécution à l'automate

À la fin d'une exécution α de l'application distribuée, on obtient une trace d'exécution E^α contenant n fichiers de logs. Chaque processus p_i de l'application a produit un fichier $E^\alpha{}_i$ contenant les occurrences d'évènements locaux.

Les évènements au sein d'un fichier de log ${E^\alpha}_{\mathrm{i}}$ sont totalement ordonnés. Il existe trois sortes d'évènements : les actions locales, les envois de messages et les réceptions de messages. Bien que les ensembles ${E^\alpha}_{\mathrm{i}}$ soient localement ordonnés, l'ensemble E^α ne peut être que partiellement ordonné. Pour cela, on utilise la relation *happened-before* [1]. Sur la base d'un ensemble partiellement ordonné, on construit un automate à état fini $A\alpha$ qui reconnaît l'ensemble des ordres possibles de consommation des logs [7].

B. De la trace d'exécution aux invariants

Une fois que l'ordre partiel de l'exécution E^α est obtenu, il est également possible d'extraire un ensemble d'invariants temporels [11]. Les invariants retenus pour notre modèle sont les suivants : a toujours suivi de b, noté $a \rightarrow b$: une occurrence de l'évènement de type a doit être suivi d'une occurrence de l'évènement de type b dans la trace d'exécution ; b toujours précédé de a, noté $a \leftarrow b$: une occurrence de l'évènement de type b doit être précédée d'une occurrence de l'évènement de type a dans la trace d'exécution ; b jamais suivi de a, noté $b \nrightarrow a$: une occurrence de l'évènement de type b ne doit jamais être suivi d'une occurrence de l'évènement de type a dans la trace d'exécution. Pour une exécution donnée, ces trois propriétés sont testées en considérant tous les couples de type d'évènements observés durant l'exécution. Pour un couple particulier, une propriété est un invariant si elle est satisfaite pour tout ordre total compatible avec l'ordre partiel caractérisant l'exécution. Une liste d'invariants est ainsi calculée pour chaque exécution.

C. Fusion d'automates correspondant à plusieurs exécutions

Une fois que les automates ($A\alpha, A\beta, A\gamma$, ...) correspondant à plusieurs exécutions (E^α, E^β, E^γ, ...) ont été construits, on cherche à obtenir un automate global A_{G} qui reconnaît l'ensemble des traces utilisées lors de la phase d'apprentissage. Pour cela, deux approches peuvent être retenues : la simple fusion des états initiaux Ei de l'ensemble des automates ; et la fusion des préfixes communs de l'ensemble des automates.

D. Fusion d'invariants correspondant à plusieurs exécutions

Cette étape a pour but de fusionner les x listes d'invariants obtenues lors des x exécutions afin d'obtenir une liste d'invariants globaux I_{G} qui sont vrais pour toutes les exécutions. Partant de la liste d'invariant correspondant à la première exécution, nous construisons en $x-1$ étapes la liste I_{G} . À chaque étape la liste en cours de construction est mise à jour en prenant en compte une nouvelle exécution. À noter qu'une propriété impliquant deux types d'évènements a et b peut ne pas faire partie d'une liste d'invariants associée à une exécution pour deux raisons distinctes. Soit des évènements de type a et b se sont effectivement produits durant l'exécution mais la propriété n'est pas vérifiée par au moins une séquence d'évènements. Soit l'exécution ne comporte aucun évènement correspondant au type a ou au type b. Dans ce dernier cas, la propriété n'a pas été évaluée. Le mécanisme de fusion des ensembles d'invariants est conçu pour accepter que les x exécutions ne soient pas caractérisées par un même ensemble de type d'évènements.

E. Généralisation d'automates

L'automate global A_{G} reconnaît l'ensemble des séquences d'évènements compatibles avec l'ordre partiel d'une trace apprise pour le construire. Cependant, il est compliqué d'exhiber l'ensemble des comportements lors de l'apprentissage. Les algorithmes de généralisation ont pour but de réduire la taille des automates tout en introduisant de nouveaux comportements. Ktail Kt [8] est un de ces algorithmes : il fusionne les états d'automates ayant les mêmes k-futurs (les mêmes séquences futures de longueur k). Cet algorithme a plusieurs variantes [9] [10] en fonction de spécificités recherchées. Le processus de fusion peut s'effectuer lors de différentes étapes. Il est possible d'adopter une fusion globale Fg suivie d'une généralisation ou une fusion itérative Fi en $x-1$ étapes avec une généralisation intermédiaire à chaque étape de la fusion. Le choix du type de fusion et du type de généralisation peut avoir un impact sur la précision du modèle après généralisation. Par ailleurs, nous envisageons une variante nommée ktail inclusif Kti [3] où nous effectuons une fusion des états de l'automate $A\gamma$ d'une exécution E^γ qui ont leurs k-futurs inclus dans les k-futurs d'un automate global temporaire A_{GT} en cours de construction pour obtenir l'automate global A_{G} = Fi-$Kti(A\gamma, A_{\mathrm{GT}})$.

IV. Vérification de la conformité d'une exécution

Dans l'optique de détecter si une exécution $E\delta$ est conforme à notre modèle de comportement légitime, on utilise les trois mécanismes suivants : le mécanisme *log checking* visant à vérifier que les fichiers de logs sont correctement formés (un message a bien été envoyé avant d'être reçu), le mécanisme *global automaton checking* vérifiant que la trace d'exécution $E\delta$ est bien reconnue par l'automate global A_{G} et le mécanisme *invariant checking* vérifiant la validité des invariants lors de la consommation de la trace.

V. Évaluation

Pour évaluer les approches de modélisation, on a choisi de construire un modèle de comportement du système de fichiers distribué nommé XtreemFS [12]. Une plateforme comportant cinq composants d'XtreemFS a été utilisée pour effectuer des exécutions légitimes et jouer des attaques contre l'intégrité du système distribué. Notre analyse porte sur l'évaluation des taux de faux positifs et de faux négatifs. Elle s'effectue selon une méthode de cross validation V-fold [13]. Cette méthode de validation permet de mesurer la précision de l'approche en divisant l'ensemble des traces en V ensembles de taille similaires, de construire plusieurs modèles avec V-1 ensembles de traces et d'évaluer ces modèles à l'aide de l'ensemble des V traces.

A. Évaluation de modèles dans un environnement sain

Évaluer la précision du modèle dans un environnement sans attaquant revient à évaluer la reconnaissance d'exécutions apprises et d'exécutions non apprises mais légitimes. Pour

cela, on évalue le taux de faux positifs de chacun de nos modèles. Pour construire ces modèles, on utilise une approche 5-fold sur 40 traces d'exécutions de notre système distribué. 32 traces parmi ces 40 sont donc utilisées. Ici nous avons retenu trois cas de figure : Ei-Fg-Kt (fusion des états initiaux, fusion globale puis généralisation et utilisation de ktail), Ei-Fi-Kt (fusion des états initiaux, fusion et généralisation itérative et utilisation de ktail) et Ei-Fi-Kti (fusion des états initiaux, fusion et généralisation itérative et utilisation de notre variante inclusive de ktail).

	Ei-Fg-Kt	Ei-Fi-Kt	Ei-Fi-Kti
Construction	$\sim$34 min	$\sim$34 min	$\sim$34 min
Fusion & Généralisation	3-10 min	$\sim$1 min	$\leqq$30 sec
Détection	1 sec - $\geqq$ 2 h	1 sec - $\geqq$ 2 h	1 sec - 3 min
Faux positifs	7% - 20%	4% - 20%	0.5 %

TABLE I: Comparaison d'approches de modèlisation

Ces expériences ont eu lieu sur une machine ayant un processeur de 4x1.8GHz, avec 4GB de RAM. Les résultats obtenus dans le tableau I présentent un comparatif des temps d'exécutions aux différentes étapes de notre approche ainsi qu'un comparatif du taux de faux positifs des différentes approches. Pour toutes les approches, le temps de détection pour une exécution varie entre 1 seconde et une dizaine de minutes. Cependant, pour les approches Ei-Fg-Kt et Ei-Fi-Kt, certaines détections ne sont toujours pas finies au bout de 2 heures. Nous considérons deux taux de faux positifs pour ces approches. Dans le premier cas, au bout de 2 heures la détection s'arrête sans générer d'alertes. Au contraire, dans le second cas, l'arrêt s'accompagne de la levée d'une d'alerte et donc d'une augmentation des faux positifs.

B. Évaluation de modèles à l'aide d'attaques

L'évaluation d'un modèle de comportement pour un environnement avec attaquant porte à la fois sur la capacité de notre modèle à accepter les traces d'exécution valides (apprises ou non) et à détecter celles qui contiennent des attaques. Pour effectuer cette évaluation, on utilise une approche 5-fold sur 200 traces d'exécutions. De plus, nous utilisons 15 traces d'exécutions qui correspondent à 5 types d'attaques différentes. L'évaluation porte sur l'approche Ei-Fi-Kti.

	Ei-Fi-Kti
Construction	$\sim$5h30
Fusion & Généralisation	$\sim$1 min
Détection	1 sec - 3 sec
Faux positifs	0%
Faux négatifs	20%

TABLE II: Évaluations de l'approche Ei-Fi-Kti

Dans cette phase on joue 5 attaques : *newfile*, *delete file*, *osd change*, *chmod*, *chown*. Le tableau II présente des résultats similaires à l'évaluation précédente. Cependant, le temps de construction des modèles pour chaque trace est plus important. Le taux de faux positifs s'est légèrement amélioré. Ces deux résultats s'expliquent par le passage de 40 à 200 traces. Enfin, on constate un taux de faux négatifs de 20% (une attaque *chown* n'a pas été détectée).

VI. CONCLUSION ET TRAVAUX FUTURS

Dans cet article, nous avons détaillé un mécanisme de modélisation de comportement hybride d'une application distribuée. Ce mécanisme présente de nombreuses options dans les différentes étapes du processus de modélisation. Nous avons présenté le mécanisme de détection utilisant ce type de modèle. Nous avons souligné l'impact des choix de modélisation sur les temps et la précision du mécanisme de détection. Les premiers résultats montrent que les temps de détection et que les taux de faux négatifs peuvent être améliorés. Les travaux futurs porteront sur la définition de nouvelles pistes (granularité des logs, utilisation d'ordres totaux particuliers pour construire l'automate, ...), l'amélioration des processus de détection [14] et de généralisation [15] ainsi que l'introduction de nouveaux types d'invariants [16].

REMERCIEMENT

Nous remercions la DGA pour son soutien financier.

RÉFÉRENCES

[1] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM 21.7*, 1978

[2] D. Lo, L. Mariani et M. Santoro, Learning extended FSA from software : An empirical assessment, *Journal of Systems and Software 85.9*, 2012

[3] D. Lorenzoli, L. Mariani et M. Pezzè, Automatic generation of software behavioral models, *Int. Conf. on Software engineering. ACM*, 2008

[4] M. Mukund, K. N. Kumar et M. Sohoni, Synthesizing distributed finite-state systems from MSCs, *Int. Conf. on Concurrency Theory. Springer Berlin Heidelberg*, 2000

[5] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy et T. E. Anderson, Mining temporal invariants from partially ordered logs, *SIGOPS Operating Systems Review*, 2011

[6] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, Inferring models of concurrent systems from logs of their behavior with CSight, *Int. Conf. on Software Engineering. ACM*, 2014

[7] E. Totel, M. Hkimi, M. Hurfin, M. Leslous, Y. Labiche, Inferring a Distributed Application Behavior Model for Anomaly Based Intrusion Detection, *European Dependable Computing Conf. (EDCC)*, 2016

[8] A. W. Biermann et J. A. Feldman, On the synthesis of finite-state machines from samples of their behavior, *IEEE transactions on Computers 100.6 : 592-597*, 1972

[9] D. Lorenzoli, L. Mariani et M. Pezzè, Inferring state-based behavior models, *Int. Workshop on Dynamic systems analysis. ACM*, 2006

[10] F. Pastore, D. Micucci et L. Mariani, Timed k-Tail : Automatic Inference of Timed Automata, *Software Testing, Verification and Validation (ICST)*, 2017

[11] M. D. Ernst, J. Cockrell, W. G. Griswold et D. Notkin, Dynamically Discovering Likely Program Invariants to Support Program Evolution, *IEEE Transactions on Software Engineering 27.2*, 2001

[12] XtreemFS Team, *"XtreemFS : http ://www.xtreemfs.org/."*, 2016.

[13] P. Burman, A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods, *Biometrika 76*, 1989

[14] C. Flanagan et P. Godefroid, Dynamic partial-order reduction for model checking software, *ACM Sigplan Notices*, 2005

[15] L. Mariani et M. Pezzè, Dynamic detection of cots component incompatibility, *IEEE software 24.5*, 2007

[16] J. G. Lou, Q. Fu, S. Yang, J. Li et B. Wu, Mining program workflow from interleaved traces, *SIGKDD Int. Conf. on Knowledge discovery and data mining*, 2010