Modélisation du comportement hybride d'une application distribuée pour la détection d'intrusion

David Lanoë*†, Eric Totel†, Michel Hurfin* et Carlos Maziero‡
*Univ Rennes, INRIA, CNRS, IRISA, 35000 Rennes, France david.lanoe@inria.fr michel.hurfin@inria.fr
†CentraleSupélec, INRIA, CNRS, IRISA, 35000 Rennes, France eric.totel@centralesupelec.fr
†Univ Federal Paraná, 81531-980 Curitiba, Brazil maziero@inf.ufpr.br

***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 nécessaires pour la construction du modèle. Enfin, une évaluation du modèle 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 relatives à la construction du modèle de comportement hybride. Dans un second temps, nous évaluons l'approche dans le contexte de la détection d'intrusion. Cet article est structuré comme suit : après un bref état de l'art dans la section II, nous introduisons quelques concepts clés 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 l'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 « happened 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, l'utilisation d'un algorithme de généralisation [8] vise 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. Notre objectif est de limiter l'introduction de comportements erronés en proposant une modélisation composée de deux sous-modèles. D'autre part, nous souhaitons évaluer à la fois l'intérêt de la généralisation et de l'utilisation de plusieurs types de modèles sur le processus de détection.

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 totaux compatibles avec l'ordre partiel [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 [9]. 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 et généralisation 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, on fusionnne les états initiaux de l'ensemble des automates. L'automate obtenu 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 possibles lors de l'apprentissage. Pour introduire de nouveaux comportements, l'algorithme de généralisation Ktail [8] est appliqué sur l'automate A_{G}. Cet algorithme fusionne les états ayant les mêmes k-futurs (mêmes séquences futures de longueur k).

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'invariants 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.

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 `InvCom` visant à vérifier qu'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 [10]. Une plateforme comportant cinq composants d'XtreemFS a été utilisée pour collecter des traces correspondant à des exécutions légitimes (40 traces) et à des attaques contre l'intégrité du système distribué (20 traces). Notre analyse porte sur l'évaluation de la précision du processus de détection. L'évaluation s'effectue selon une méthode de cross validation V-fold [11]. Cette méthode consiste à diviser l'ensemble des traces légitimes en V ensembles de taille similaires, de construire plusieurs modèles avec V-1 ensembles de traces et d'évaluer ces modèles. Dans nos expérimentations, la valeur de V est fixée à 5. D'autres part, nous identifions les différents modèles en utilisant une convention de nommage composée de 2 champs séparés par le caractère « - ». Le premier champ indique le modèle auquel nous nous référons (`Mod` représente l'ensemble du modèle, `Aut` pour le sous-modèle Automate, et `Inv` pour le sous-modèle Invariant). Le deuxième champ correspond a la valeur numérique de k utilisé dans *Ktail*.

A. Évaluation de modèles dans un environnement sain

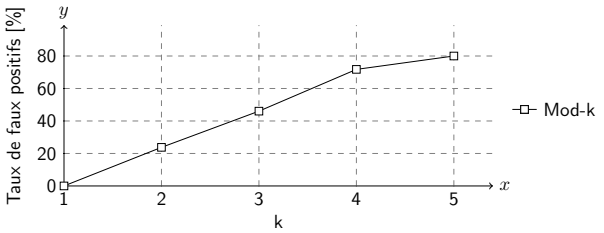

FIGURE 1: Taux de faux positifs pour différentes valeurs de k

Évaluer la précision du modèle dans un environnement sans attaquant revient à évaluer la reconnaissance d'exécutions

TABLE I: Détection d'attaques dans différents contextes à l'aide de différents sous-modèles

Attack	*NewFile*				*DeleteFile*				*OsdChange*				*Chmod*				*Chown*			
Context	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4
`Aut-5`	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5
`Aut-1`	-	2/5	-	5/5	-	3/5	1/5	5/5	2/5	5/5	4/5	5/5	-	1/5	-	5/5	-	1/5	-	-
`Inv`	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	-	-	-	-
`InvCom`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	-

apprises et d'exécutions non apprises mais légitimes. Pour cela, on mesure un taux de faux positifs en fonction du paramètre de généralisation k. Dans la figure 1, on peut observer un résultat prévisible, plus la valeur de k est petite plus le taux de faux positifs est faible. On remarque que si la valeur du paramètre k est trop élevé (k=4 ou k=5), le modèle ne reconnait pas ou peu de nouveaux comportements.

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

L'évaluation d'un modèle de comportement pour un environnement comportant des attaquants 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. Nous avons observé précédemment qu'une valeur trop élevée pour le paramètre k limitait fortement l'acceptation de nouvelles traces.

Pour évaluer la capacité de détection des attaques de chaque modèle, nous avons déployé cinq attaques connues contre l'intégrité du système de fichier (*NewFile*, *DeleteFile*, *OsdChange*, *Chmod* et *Chown*) suivant quatre contextes différents : (c1) aucun client n'est actif, (c2) avant les actions du client, (c3) après les actions du client, et (c4) effectué à partir d'une source qui n'est pas l'adresse du client.

Dans le tableau I, les sous-modèles `Aut` et `Inv` sont étudiés séparément, afin de montrer leur complémentarité et de valider la nécessité de conserver deux sous-modèles. Même s'il ne compose pas directement notre modèle, nous indiquons avec une marque « ✓ » si l'invariant sur les communications (voir Section 4) est capable de détecter l'attaque. Pour chaque type de modèle, on évalue les attaques. Un résultat d / 5 signifie que d modèles sur cinq ont détecté l'attaque (« - » signifie 0/5). On constate dans ce tableau que le paramètre k a le même impact sur l'acceptation de nouveaux comportements pour les automates `Aut-1` et `Aut-5`. On observe aussi que les invariants semblent particulièrement appropriés pour détecter les attaques dans les contextes c1 et c2 alors que l'automate `Aut-1` a l'air plus adapté pour le contexte c4. L'utilisation d'un modèle composé de deux sous-modèles complémentaires (un automate généralisé et une liste d'invariants) permet d'introduire de nouveaux comportements non appris (et corrects pour certains d'entre eux) sans trop impacter sa capacité de détection.

VI. CONCLUSION ET TRAVAUX ANNEXES

Dans cet article, nous avons détaillé un processus de modélisation de comportement hybride d'une application distribuée. D'autre part, nous avons présenté un mécanisme de détection correspondant ce type de modèle. Enfin, nous avons proposé une méthode d'évaluation introduisant la notion de contexte d'attaque. Lors de la phase d'évaluation, nous avons souligné l'impact de la généralisation sur les taux de faux positifs et de faux négatifs. Par ailleurs, cette évaluation a justifié l'utilisation de plusieurs types de modèles pour détecter les attaques dans différents contexte. Des travaux annexes portent sur l'utilisation d'autres types d'invariants [13], l'introduction de « scopes » pour les invariants [14], l'utilisation d'autres types de modèles [15] et la réduction de la complexité liée aux processus de construction [12] et de généralisation.

REMERCIEMENT

Nous remercions le ministères des Armées (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] 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

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

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

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

[13] 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

[14] M. B. Dwyer, G. S. Avrunin et J. C. Corbett, Patterns in property specifications for finite-state verification, *Int. Conf. on Software engineering*, 1999

[15] R. Sekar, M. Bendre, D. Dhurjati et P. Bollineni, A fast automaton-based method for detecting anomalous program behaviors, *Symposium on Security and Privacy*, 2001