RENPAR'16 / CFSE'4 / SympAAA'2005 / JC
Le Croisic, France, 5 au 8 avril 2005

Detection d'Attaques Massives et Certification Probabiliste d'un Calcul Global*

Sébastien Varrette[1,2], Jean-Louis Roch[1], Axel W. Krings[3], Samir Jafar[1]

[1] Projet APACHE (CNRS/INPG/INRIA/UJF), Laboratoire ID-IMAG, Grenoble, France
[2] Université du Luxembourg, LIASIT/FNRL, Luxembourg
[3] University of Idaho, USA

Résumé
Cet article présente une nouvelle approche permettant de certifier l'exactitude de l'exécution d'un programme de calcul global en environnement hostile. Ainsi, les tâches ou les résultats produit peuvent avoir été corrompus par un attaquant. Etendant des approches précédentes limitées aux cas de tâches indépendantes, nous introduisons la *certification probabiliste* Monte-Carlo et Atlantic-City, permettant d'établir si les résultats du calcul global sont corrects ou pas. Cette approche probabiliste ne fait aucune hypothèse sur l'attaque et l'erreur de certification est uniquement liée à des choix aléatoires malchanceux. Des bornes associées à la certification sont également fournies dans le cas général de graphes de tâches, mais aussi plus spécifiquement pour des dépendances arborescentes que l'on trouve dans les applications d'analyses d'images médicales qui motivent nos recherches.

Mots-clés : Certification de résultats, Algorithme probabiliste, Calcul Global

1. Introduction

Cet article adresse le problème de la certification de programmes massivement parallèles (i.e possédant un grand nombre n de tâches) sur une plateforme distribuée de grande taille, type grille ou pair-à-pair. Dans un tel environnement, les ressources partagées peuvent être victimes d'attaques massives qui affectent plus de $q.n$ tâches, où $q \in]0, 1]$ est le taux de l'attaque. Un exemple typique d'une telle attaque est un virus, propagé par un ver et affectant un type de système d'exploitation donné en permettant à un attaquant de contrôler les ressources affectées. Les statistiques de CERT/CC [1] témoignent de l'impact des attaques massives: tandis que le nombre de vulnérabilités reportées se stabilise (4129 en 2002, 3784 en 2003), le nombre d'incidents reportés croît exponentiellement (82094 incidents reportés en 2002 contre 137529 en 2003).
Dans le cadre de cet article, nous considérons une application parallèle modélisée par un graphe de tâches. Nous supposons que l'application tolère un grand nombre d'erreurs, typiquement $\delta.n$ fautes où n est le nombre de tâches de l'application et δ son taux de tolérance. Cette hypothèse est vérifiée par de nombreuses applications cible des plateformes de calcul global [5, 10] qui sont souvent de type simulation Monte-Carlo. Pour assurer un tel taux de tolérance lorsque les tâches sont indépendantes, L. Sarmenta propose dans [10] une technique basée sur le calcul de la crédibilité qui peut être accordée à un résultat donné. Cette approche reste limitée car elle suppose d'une part que l'utilisateur détermine la crédibilité qu'il accorde à chaque composant impliqué dans le calcul et d'autre part que les tâches affectées par les attaques suivent une distribution de Bernouilli. Dans le cadre de tâches avec dépendances, L. Gao et G. Malewicz [6] utilisent des techniques de duplication et d'exécution de certaines tâches sur des ressources fiables (toujours supposées peu nombreuses) qui permettent d'atteindre un taux de tolérance δ.
Ce taux de tolérance est en particulier vérifié par l'application médicale étudiée dans le projet Ragtime [8] qui motive ce travail. Dans cette application, une image médicale est soumise pour comparaison (analyse de similarité) aux images stockées dans différentes bases comportant un grand volume de

* Ce travail est supporté par le CNRS et la région Rhône-Alpes (projet Ragtime)

données radiographiques. Dans un premier temps, les images liées à la requête soumise sont extraites; puis les calculs de similarité sont effectuées sur chaque image extraite; enfin, les résultats sont classés en fonction de leurs similarités. Il y a deux niveaux de parallélisme: entre les calculs de similarité d'une part et au sein d'un même calcul qui peut lui-même porter sur une séquence d'images. Cette application correspond à un arbre de tâches. Plus généralement, nous considérons que l'application est modélisée par un graphe de dépendances (flot de données macroscopique), implicite à de nombreux langages de programmation parallèle comme Athapascan [13]. Etant donnés les résultats d'une exécution, le problème est alors de déterminer si les résultats sont corrects (i.e. le taux d'erreur est alors garanti inférieur à δ) ou si l'exécution a été victime d'une attaque massive qui a conduit à la falsification des résultats de $n_F \geq \lceil q.n \rceil$) tâches.
Pour résoudre ce problème lorsque les tâches sont indépendantes, C. Germain et N. Playez [5] considèrent le cas où la plupart des ressources utilisées sont honnêtes alors que les ressources corrompues falsifient toujours leurs résultats. En supposant une distribution de Bernouilli des erreurs, les auteurs proposent l'utilisation d'un test statistique séquentiel pour se ramener à la vérification d'un nombre limité de tâches indépendantes choisies aléatoirement et ré-exécutées sur des oracles fiables. Ce test prend en compte deux paramètres fixés par l'utilisateur: le risque de fausse alarme (faux positif) et de non detection (faux négatif). Suivant les mêmes hypothèses, S. Jafar, S. Varrette et J.L. Roch [7] proposent d'étendre cette approche aux cas de calculs parallèles composés de tâches avec dépendances, en supposant aussi qu'une tâche falsifiée renvoit toujous un résultat erroné. Pour lever cette hypothèse restrictive, nous étudions dans cet article le cas général d'une attaque malicieuse dans laquelle une tâche falsifiée peut renvoyer un résultat correct lorsque ces entrées sont incorrectes. Enfin, de façon similaire à [5], nous considérons une borne inférieure (et non supérieure comme pour [6, 10]) sur le nombre de tâches falsifiées ce qui permet d'adresser les attaques classiques qui peuvent affecter tout un sous-réseau (ver, attaques basées sur une plage d'adresses IP etc...).
Après une description des hypothèses et des notations (§2), nous proposons d'adopter une vue directement inspirée des algorithmes probabilistes (§3). Pour détecter si une exécution contient des résultats erronés, un algorithme probabiliste est utilisé qui effetue des choix aléatoires de tâches à vérifier pour savoir si l'exécution est correcte ou non. Comme cette détection est probabiliste, le résultat de l'algorithme de détection peut être faux mais sa probabilité d'erreur est bornée indépendamment de l'attaque effective qui a pu avoir lieu. L'intérêt de cette approche est de limiter le coût de la certification tout en garantissant que la probabilité d'erreur de certification reste bornée par un seuil fixé par l'utilisateur. Nous explicitons des algorithmes de détection probabiliste d'abord lorsque les tâches sont indépendantes (§4) puis dépendantes (§5). Nous montrons que lorsque le graphe des tâches est de type arbre, le coût moyen de la certification est lié à la profondeur du graphe, qui est petite pour l'application médicale considérée.

2. Définitions et hypothèses

2.1. Plateforme d'exécution

L'exécution de l'application considérée repose sur une plateforme distribuée (de type grille ou pair-à-pair). Cette plateforme se divise en deux types de ressources : les *workers* d'une part, considérés comme non sûrs car exécutant les tâches dans un environnement peu fiable et les *oracles* d'autres part qui sont des ressources sûres utilisées pour certifier les résultats de l'exécution. Les communications entre ces ressources sont effectuées par un serveur de checkpoint considéré comme sûr et sécurisé qui stocke le graphe de l'exécution (voir paragraphe suivant).

2.2. Représentation des exécutions

Dans toute la suite, l'application parallèle est modélisée par un DAG biparti $G = (\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ est un ensemble de noeuds divisés en deux classes: les tâches d'une part et les paramètres (d'entrée et de sortie) de ces tâches d'autre part. Une tâche T_j est vue au sens de l'ordonnancement de tâches, autrement dit T_j dénote la plus petite unité de programme dans une exécution donnée. $\mathcal{E}$ est un ensemble d'arêtes e_{jk}, $j \neq k$, représentant les précédences entre les noeuds de G. Dans toute la suite, ce graphe est supposé déterministe et connu de l'utilisateur *a priori*. Enfin, le nombre total de tâches T_j dans G est n.

2.3. Impact des fautes

On note E l'exécution d'un programme G sur un ensemble de ressources non fiables. à partir de l'ensemble des paramètres d'entrée initiaux $\hat{I}$. Chaque tâche T de E s'exécute sur les paramètres d'entrée $i(T, E)$ et génère les résultats $o(T, E)$. $i(T, E)$ est constitué soit de données de $\hat{I}$, soit de sorties d'autres tâches T_k (i.e $o(T_k, E)$). Soit $\hat{E}$ l'exécution du même programme G mais sur un ensemble de ressources fiables.

Definition 1 (Etat d'une exécution). *Une exécution* E *est dites "correcte" et on note* $E = \hat{E}$ *si toute tâche utilise les mêmes paramètres d'entrées et génère les mêmes sorties dans* E *et* $\hat{E}$*. Dans le cas contraire, on dit que l'exécution est "falsifiée".*

On aura noté que l'on utilise la notation "chapeau" (par exemple $\hat{I}$ ou $\hat{E}$). pour qualifier un ensemble sûr de valeurs de paramètres ou une exécution effectuée sur des ressources fiables, Ainsi, soit T une tâche et $i(T, E)$ ses paramètres d'entrées lors de l'exécution E sur des ressources non fiables. Les paramètres d'entrées de la même tâche T exécutée avec les mêmes entrées mais sur une ressource fiable sont notés $\hat{i}(T, E)$. Enfin, les paramètres d'entrées de la même tâche T dans l'exécution correcte $\hat{E}$ sur des ressources fiables sont notés $\hat{i}(T, \hat{E})$.
De façon similaire, on définit les notations $o(T, E)$, $\hat{o}(T, E)$ et $\hat{o}(T, \hat{E})$ relatives aux sorties de l'exécution de la tâche T.
Une tâche peut etre victime d'une attaque, par exemple sous la forme d'un virus, d'une exécution d'un client incorrect, ou d'une intrusion lors de la lecture ou l'écriture dans le serveur de checkpoint. Il est important de noter qu'on ne peut pas decider si une tâche a été victime d'une attaque si l'on ne connait que ses paramètres d'entrée et de sortie.

Definition 2 (Tâche correcte). *Une tâche est* correcte *ssi* $i(T, E) = \hat{i}(T, \hat{E})$ *et* $o(T, E) = \hat{o}(T, \hat{E})$.

Autrement dit une tâche T qui a lu en entrée sur le serveur de checkpoint la valeur correcte $\hat{i}(T, \hat{E})$ de ses paramètres d'entrées et qui a écrit en sortie le résultat correct $\hat{o}(T, \hat{E})$ est correcte. Une tâche qui est correcte mais qui a été victime d'une attaque ne pose pas de problème et n'est pas considérée dans la suite comme falsifiée.

Definition 3 (Tâche falsifiée). *Une tâche est dite "*falsifiée*" (ou attaquée) ssi* $\Longleftrightarrow$ *elle a été modifiée (i.e. victime d'une attaque par virus, trojan etc...) et si elle n'est pas correcte.*

Ainsi, si une tâche T falsifiée a lu en entrée sur le serveur de checkpoint la valeur correcte $\hat{i}(T, \hat{E})$ de ses paramètres d'entrées, alors son résultat est incorrect: $o(T, E) \neq \hat{o}(T, \hat{E})$.
$\Longleftrightarrow$ elle est victime d'une attaque et si elle avait les bon paramètres en entrée alors elle a n'a pas fournie le bon résultat. qui a été attaquée ou modifiée par un virus, un trojan etc...
Les propositions suivantes sont équivalentes et se déduisent directement de la définition précédente:

Lemma 1. *Une tâche* T *d'une exécution* E *est falsifiée*

$$\Longleftrightarrow \ T \textit{ a été modifiée et } \begin{cases} \textit{soit } o(T, E) \neq \hat{o}(T, \hat{E}) \\ \textit{soit } o(T, E) = \hat{o}(T, \hat{E}) \textit{ mais } i(T, E) \neq \hat{i}(T, \hat{E}) \end{cases}$$

$$\Longleftrightarrow \ T \textit{ a été modifiée et } \begin{cases} \textit{soit } i(T, E) \neq \hat{i}(T, \hat{E}) \\ \textit{soit } i(T, E) = \hat{i}(T, \hat{E}) \textit{ mais } o(T, E) \neq \hat{o}(T, \hat{E}) \end{cases}$$

Dans la suite, n_F désigne le nombre de tâches falsifiées dans E. A partir de cette définition, il est possible de définir formellement une attaque massive:

Definition 4 (Attaque massive). *Une attaque massive de rapport* q *consiste à falsifier l'exécution d'au moins* $n_q = \lceil q.n \rceil$ *tâches* ($n_F \geq n_q$).

On remarquera que les notations $\hat{o}(T, E)$ et $\hat{o}(T, \hat{E})$ diffèrent. En effet, chacune indique des sorties générées par une exécution sur des ressources fiables mais alors que la première se base sur les paramètres d'entrée de E (potentiellement erronés si une tâche précédent T a été falsifiée), la seconde utilise les entrées garanties correctes de $\hat{E}$. Cette nuance permet d'établir la définition suivante:

Definition 5 (Résultat falsifié). *Le résultat d'une tâche* T *est dit falsifié si* $o(T, E) \neq \hat{o}(T, \hat{E})$,

Le fait que le résultat d'une tâche soit falsifié est décidable mais éventuellement coûteux. De plus, une tâche T falsifiée peut causer un résultat falsifié qui sera propagé à toutes les tâches qui dépendent du résultat de T.

3. Notion de certification probabiliste

L'objectif de notre approche est de fournir un algorithme permettant de déterminer l'état d'une exécution E donnée. Bien que la valeur de q (le seuil de détection) puisse être fixée arbitrairement par l'utilisateur, il convient de choisir en pratique une valeur proche du seuil de tolérance de l'application δ. Ainsi, deux cas de figures seront possibles à l'issue de la détection:

1. soit $n_F < n_q$ et l'erreur sur le résultat de l'application est dans tous les cas tolérée. Dans ce cas, une erreur sur le résultat 'CORRECT' de l'algorithme de certification n'a pas d'influence.

2. soit $n_F \geq n_q$ et le résultat de l'application n'est pas acceptable. Cette occurrence doit donc être detectée.

L'objectif est alors de détecter si une exécution $execution$ a été victime ou non d'une attaque massive. Aussi, dans toute la suite, nous considérons deux scénarios pour lesquels soit il n'y a pas eu d'attaques massives et alors toutes les tâches s'exécutent correctement ($n_F = 0$); soit n_F est suffisament grand, i.e. $n_F \geq n_q = \lceil q.n \rceil$. ce qui correspond a une attaque massive de rapport q.
Un test déterministe est coûteux puisqu'il nécessite la réexécution de $n - n_q + 1$ tâches dans le pire cas. Aussi, pour avoir une détection efficace, nous considérons une certification probabiliste. Contrairement à un algorithme déterministe, un algorithme probabiliste peut délivrer des sorties différentes pour différentes exécutions sur un même jeu d'entrée. Une analogie peut être faite au test (Monte-Carlo) de composition de Miller-Rabin (voir [12] p. 139). Ce test permet de déterminer si un nombre est premier, et assure que la probabilité de non-detection de composition est suffisamment faible (majorée par un seuil choisi par l'utilisateur). Par son efficacité, c'est ce test qui est le plus souvent utilisé pour déterminer si un nombre est premier bien que le résultat puisse être faux.
Dans toute la suite, pour déterminer si l'exécution E est correcte ou non, des tâches choisies aléatoirement sont vérifiées. Il convient d'insister sur le fait que cette approche permet de limiter le nombre de réplications de tâches tout en assurant une certification adaptable avec un seuil d'erreur ϵ fixé par l'utilisateur. Par exemple, une certification probabiliste pourra consister à ré-exécuter des tâches de E choisies aléatoirement sur des ressources fiables en comparant $\hat{o}(T, E)$ et $\hat{o}(T, \hat{E})$ pour $i(T, E) = \hat{i}(T, E)$. Si les résultats diffèrent, alors E est prouvée falsifiée Néanmoins, si E est falsifiée, une certification probabiliste à erreur ϵ assure que la probabilité de non-detection d'erreur est inférieure ou égale à ϵ.
De manière plus générale, la vérification peut être faite sur des ressources non fiables; la certification probabiliste est alors de type Atlantic City.

Definition 6 (certification probabiliste Atlantic City). *Etant donnée une exécution* E, *une certification probabiliste Atlantic City est un algorithme probabiliste prenant en entrée un seuil arbitraire* ϵ, $0 < \epsilon \leq 1$ *et une exécution* E *et qui délivre en sortie:*
(1) soit exécution correcte*;*
(2) soit exécution falsifiée*.*
De plus, soit n_F *le nombre de tâches falsifiées dans* E. *La sortie de la certification probabiliste est erronée dans les deux cas suivants:*

- *l'algorithme renvoit* exécution correcte *alors que* $n_F \geq n_q$*;*
- *l'algorithme renvoit* exécution falsifiée *alors que* $n_F = 0$*.*

La probabilité que la sortie de la certification soit erronée est inférieure ou égale a ϵ.

Si la vérification d'une tâche est faite sur une ressource fiable, alors le test de vérification fournit une réponse certaine: si une tâche est détectée falsifiée, alors $n_F \neq 0$ et la certification renvoit *exécution falsifiée*. La sortie de l'algorithme de certification probabiliste ne peut alors être erronée que dans le cas (1); la certification est de type Monte-Carlo.

Definition 7 (certification probabiliste Monte-Carlo). *Etant donnée une exécution* E, *une certification probabiliste Monte-Carlo est un algorithme probabiliste prenant en entrée un seuil arbitraire* ϵ, $0 < \epsilon \leq 1$ *et une exécution* E *et qui délivre en sortie:*

(1) soit exécution correcte*;*
(2) soit exécution falsifiée.
De plus, soit n_F *le nombre de tâches falsifiées dans* E*. La sortie de la certification probabiliste est erronée uniquement si l'algorithme renvoit* exécution correcte *alors que* $n_F \geq n_q$.
La probabilité que la sortie de la certification soit erronée est inférieure ou égale a ϵ.

La certification d'une exécution E peut être considérée par rapport aux tâches et aux résultats fournis par ces tâches. Dans le premier cas, c'est le comportement correct de la tâche qui est analysé lorsque le second cas focalise sur l'exactitude des résultats. La différence entre ces deux vues sera plus claire en §5 où le problème de la propagation de résultats au sein d'un graphe avec dépendances sera abordé.

4. Illustration dans le cas de tâches indépendantes

Dans cette section, nous considérons le cas où toutes les tâches de G sont indépendantes. Dans ce cas, la certification des tâches est équivalente à celle des résultats. Nous présentons maintenant un test Atlantic-City ACT (resp. Monte-Carlo MCT) basé sur la vérification sur des ressources non-fiables avec probabilité d'erreur majorée par $p < 1$ (resp. fiables) pour détecter si l'exécution E contient des tâches falsifiées ou non.

4.1. Algorithme $MCT(E)$ et $ACT(E)$

1. Choisir uniformément une tâche T dans E. Les entrées (resp. les sorties) de T dans E sont $i(T, E)$ (resp. $o(T, E)$).
2. Re-exécuter T sur une ressource fiable, en utilisant les entrées de E, i.e. $i(T, E)$, pour obtenir les sorties $\hat{o}(T, E)$. Si $o(T, E) \neq \hat{o}(T, E)$ renvoyer "falsifiée";
3. Renvoyer "exécution correcte".

Comme les tâches de G sont indépendantes, on a toujours $i(T, E) = \hat{i}(T, \hat{E})$[2]. Si $MCT(E)$ renvoie "falsifiée", alors une tâche falsifiée a été sélectionnée et on sait avec certitude que l'exécution E est falsifiée; dans le cas de $ACT(E)$, il n'y a pas de certitude mais seulement une probabilité supérieure à $(1 - p)$ de ne pas s'être trompé. Maintenant, si $MCT(E)$ renvoie "correcte", il ne peut y avoir eu erreur lors de la certification que si la tâche choisie n'a pas été falsifiée alors que $n_F \geq n_q$.

Lemma 2. *Soit* E *une exécution comportant* n *tâches indépendantes et* $n_F \geq n_q$ *tâches falsifiées.*
La probabilité d'erreur de $MCT(E)$ *est* $1 - \frac{n_F}{n} \leq 1 - q$.
De plus, la probabilité que la réponse soit "falsifiée" est $\frac{n_F}{n}$.

Proof. La probabilité que $MCT(E)$ choisisse une tâche falsifiée est $\frac{n_F}{n}$. Donc la probabilité que $MCT(E)$ renvoit "falsifiée" est $\frac{n_F}{n}$. Ainsi, la probabilité d'erreur (i.e. $MCT(E)$ renvoie "correcte") est $1 - \frac{n_F}{n} \leq 1 - q$. □

Le lemme suivant analyse l'erreur de l'algorithme $ACT(E)$.

Lemma 3. *Soit* E *une exécution comportant* n *tâches indépendantes et soit aucune, soit* $n_F \geq n_q$ *tâches falsifiées. Soit* $p < 1$ *une borne sur l'erreur de vérification d'une tâche. La probabilité d'erreur* e *lorsque* $ACT(E)$ *renvoie "correcte" est* $e = 1 - \frac{n_F}{n}.(1 - p) \leq 1 - q(1 - p)$.
La probabilité d'erreur lorsque $ACT(E)$ *renvoit "falisifiée" est* $e = p$.
La probabilité d'erreur est donc majorée par $1 - q(1 - p)$ *dans tous les cas.*

Proof. L'algorithme ACT ne délivre une erreur que dans les cas suivants:

- si l'algorithme renvoie "correcte" alors que l'exécution comporte $n_F \geq n_q$ tâches falsifiées: alors
  - soit une tâche correcte a été sélectionnée, ce qui a une probabilité $\left(1 - \frac{n_F}{n}\right)$;
  - soit une tâche falsifiée a été sélectionnée mais vérifiée sur une ressource non fiable: la probabilité de cet événement est majorée par $p.\frac{n_F}{n}$.

[2] Dans le cas de tâches avec dépendances, l'hypothèse qui suit ne sera plus vérifiée comme on le verra dans la suite.

Les deux événements étant indépendants, la probabilité e d'erreur est

$$e \leq \left(1 - \frac{n_F}{n}\right) + p.\frac{n_F}{n} = 1 - \frac{n_F}{n}.(1-p) \leq 1 - q(1-p).$$

- l'algorithme renvoit "falsifiée": il y a erreur uniquement si l'exécution est correcte et que la tâche sélectionnée a été vérifiée sur une ressource non fiable. La probabilité de cet événement est $e \leq p$.

Pour conclure, on a $p < 1$ d'où $p < 1 - q(1-p) < 1$. □

Le théorème suivant fournit une borne inférieure du nombre de tâches à ré-exécuter dans le cadre de $MCT(E)$ pour atteindre un ϵ donné.

Theorem 1. *Soit* E *une exécution comportant* n *tâches indépendantes. On suppose que* E *est soit correcte, soit attaquée massivement de rapport* q. *Pour tout* ϵ *fixé, le nombre d'exécutions independantes de l'algorithme* $MCT(E)$ *nécessaire pour atteindre la certification probabiliste de* E *à erreur* ϵ *est* $N \geq \lceil \frac{\log \epsilon}{\log(1-q)} \rceil$.

Proof. Considerons N exécutions indépendantes de l'algorithme $MCT(E)$. Si pendant l'une de ces N exécutions, $MCT(E)$ choisit une tâche falsifiée, alors l'exécution est prouvée falsifiée. On suppose donc que seules des tâches non falsifiées sont sélectionnées. D'après le lemme 2, cet évènement se réalise avec la probabilité $1 - \frac{n_F}{n} \leq 1 - q$. Ainsi, N applications independantes de $MCT(E)$ fournit une certification Monte-Carlo avec une probabilité d'erreur vérifiant $\epsilon \leq (1-q)^N$. Pour ϵ fixé, il est suffisant de selectionner $N \geq \lceil \frac{\log \epsilon}{\log(1-q)} \rceil$ tâches. □

Les deux courbes proposées en figures 1 et 2 illustent l'impact de ϵ et de q sur N dans le cadre d'une certification Monte-Carlo.

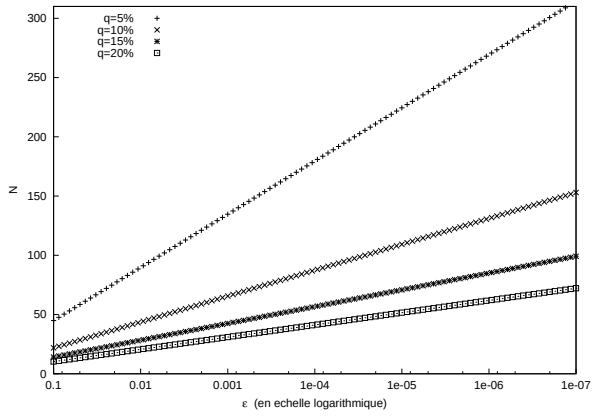

Figure 1: Impact de ϵ sur N dans $MCT(E)$.

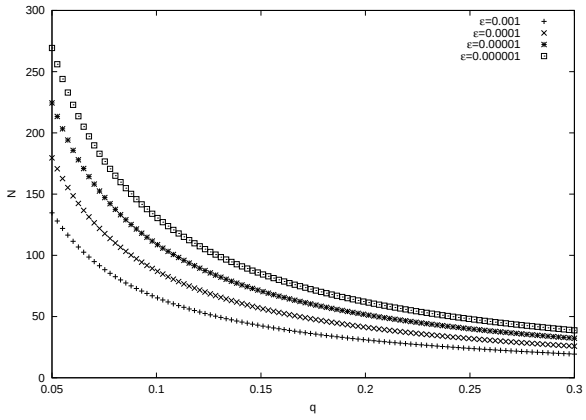

Figure 2: Impact of q on N dans $MCT(E)$.

Par ce théorème, on montre que la certification Monte-Carlo d'un graphe de $n = 10^6$ tâches attaqué massivement avec un rapport $q = 0.1$ (au moins 10% des machines sont attaquées) ne requiert que 175 tests de tâches pour obtenir une certification Monte-Carlo avec erreur $\epsilon = 10^{-8}$.
Le théorème ci-dessous, similaire au théorème 1, considère l'algorithme Atlantic-City $ACT(E)$ lorsque les vérifications sont effectuées sur des machines non fiables de probabilité d'erreur majorée par p.

Theorem 2. *Soit* E *une exécution comportant* n *tâches indépendantes . On suppose que* E *est soit correcte soit attaquée massivement de rapport* q. *On suppose de plus que la probabilité d'erreur lors de la vérification des tâches sur des ressources non fiables est majorée par* p. *Pour tout* ϵ *fixé,* $\geq \lceil \frac{\log \epsilon}{\log[(1-q.(1-p)]} \rceil$ *exécutions independantes de l'algorithme* $ACT(E)$ *suffisent pour atteindre une certification probabiliste que* E *est correcte avec une probabilité d'erreur inférieure à* ϵ.

Proof. Considerons N exécutions indépendantes de l'algorithme $ACT(E)$. Si pendant l'une de ces N exécutions, $ACT(E)$ renvoit "falsifiée", alors l'exécution est considérée falsifiée. Ces N exécutions de ACT ne délivrent une erreur que si l'algorithme renvoie "correcte" alors que l'exécution comporte $n_F \geq n_q$ tâches falsifiées: alors les N exécutions ont été erronées ce qui a une probabilité $e \leq (1 - q(1-p))^N$ d'après le lemme . D'où le résultat.
On peut cependant remarquer que la probabilité que la réponse de ACT soit "falsifiée" après ces $N =\geq \lceil \frac{\log \epsilon}{\log[(1-q.(1-p)]} \rceil$ exécutions alors que le E est correcte est que l'une au moins des N tâches sélectionnées a été vérifiée sur une ressource non fiable. La probabilité de cet événement est $e \leq 1 - (1-p)^N$, ce qui nécessite que p soit suffisamment petit. □

5. Cas des tâches dépendantes

Dans la section précédente, nous avons abordé le cas d'une certification d'une exécution E composée de tâches indépendantes. Dans ces conditions, il n'y a pas de différences entre la certification de tâches et de résultats.Plus précisément, toute ré-exécution utilise des entrées fiables, et aucune tâche n'utilise de sorties d'une autre tâche. Il en va différemment pour les exécution E composée de tâches avec dépendances. Le problème repose dans la façon dont une ressource fiable détermine si une résultat est valide ou non. Toute mesure visant à comparer ce résultat à une exécution de la tâche sur des ressources fiables est conditionné par le fait que les paramètres d'entrées utilisé dans la ré-exécution soient valides. Ainsi, étant donné une exécution E composé de tâches avec dépendances et une tâche $T \in E$, la relation $o(T, E) = \hat{o}(T, E)$ indique seulement que les résultats calculés sont les mêmes mais pas s'ils sont corrects. On notera que l'exactitude des résultats suppose que $o(T, E) = \hat{o}(T, \hat{E})$.

5.1. Tâches falsifiées et initiateurs

Les tests probabilistes utilisés dans la section 4 ne sont valides que pour la certification de tâches indépendantes. Si on devait utiliser le même raisonnement, une certification basée sur l'algorithme $MCT(E)$ supposerait que $o(T, E) \neq \hat{o}(T, E)$ pour toute tâche falsifiée. Cette hypothèse est trop restrictive car elle suppose que toute ré-exécution ayant des valeurs d'entrées (éventuellement incorrectes) fixées prouvera la falsification. Cette faible hypothèse peut être facilement exploitée par un attaquant malicieux (si E est un arbre, cet attaquant se contentera de falsifier uniquement la tâche racine par exemple). Il est donc nécessaire d'analyser plus profondément le comportement d'une tâche falsifiée.
Si $o(T, E) \neq \hat{o}(T, E)$ alors l'exécution E est falsifiée. Cependant, $o(T, E) = \hat{o}(T, E)$ indique des résultats correct seulement si les paramètres d'entrée sont corrects, c'est à dire si $i = \hat{i}$. Cela implique que T n'a pas de prédécesseur falsifié. Cela nous amène à la définition suivante:

Definition 8 (Tâche initiatrice). *Une tâche* initiatrice *(on parle aussi d'*initiateur*) est une tâche falsifiée dont aucune tâche du graphe de précédence n'est falsifiée.*

La probabilité associée au choix aléatoire d'une tâche initiatrice sera la base de la certification de résultat proposée dans la suite. Il est important de remarquer qu'il est difficile de statuer sur la capacité de détecter un résultat falsifié à partir d'une tâche qui n'est pas initiatrice: la relation $o(T, E) = \hat{o}(T, E)$ ne fournit aucune information sur le fait que la tâche a été falsifiée ou non, sauf si T est une tâche initiatrice.

5.2. Principe de la certification

Dans toute la suite, n_I désigne le nombre de tâches initiatrices dans le graphe G. La certification probabiliste est directement liée à la possibilité de sélectionner un initiateur. Il est aussi important de noter que le calcul de n_I est fortement dépendant du graphe et des noeuds qui ont été falsifiés. Nous proposons maintenant le lemme suivant, directement dérivé du lemme 2:

Lemma 4. *Soit E une exécution comportant n tâches avec dépendances Soient n_F (resp. n_I) le nombre de tâches falsifiées (resp. initiatrices) dans E. On suppose $n_I \leq n_F$. La probabilité que $MCT(E)$ renvoie "falsifiée" est au moins $\frac{n_I}{n}$ et la probabilité qu'il renvoit "correcte" est au plus $1 - \frac{n_I}{n}$.*

Proof. La probabilité que $MCT(E)$ choisisse un initiateur et donc renvoit "falsifiée" est $\frac{n_I}{n}$.
Donc $\mathcal{P}(MCT(E)=\text{"falsifiée"}) \geq \frac{n_I}{n}$ et $\mathcal{P}(MCT(E)=\text{"correcte"}) \leq 1 - \frac{n_I}{n}$ □

Le lemme suivant adapte le résultat obtenu dans le cadre du théorème 1.

Lemma 5. *Soit* E *une exécution comportant* n *tâches dépendantes et* n_I *tâches initiatrices. On suppose que* E *est soit correcte, soit attaquée massivement de rapport* q. *Pour tout* ϵ *fixé, le nombre d'exécutions independantes de l'algorithme* $MCT(E)$ *nécessaire pour atteindre la certification probabiliste de* E *à erreur* ϵ *est* $N \geq \lceil \frac{\log \epsilon}{\log(1-\frac{n_I}{n})} \rceil$.

Proof. Similaire à celle du théorème 1 est utilisant la probabilité que seules des tâches non-initiatrices soient choisies (cette probabilité étant $1 - \frac{n_I}{n}$). □

Contrairement au cas du théoreme 1, ce resultat est beaucoup plus restrictif car la valeur de n_I dépend du graphe considéré et peut être très faible dans le pire cas d'attaque comme on le verra dans la suite. Dans le reste de cet article, nous étudions l'impact d'une attaque dans le pire cas où l'attaquant peut estimer par avance la structure du graphe. Nous définissons maintenant une caractérisation des tâches ayant des résultats falsifiés pour montrer au §5.3 que le nombre d'initiateurs n_I peut être minoré en fonction de G et du rapport q de l'attaque massive, même dans un cadre pathologique.
On définit pour cela les notations suivantes: $G^<(T)$ dénote le sous-graphe des prédécesseurs de la tâche T dans le graphe. On étend cette notation à un ensemble de tâches V par $G^<(V)$.On pose également $G^\leq(T) = G^<(T) \cup \{T\}$.
De façon duale, $G^>(T)$ dénote le graphe des successeurs de T dans le graphe et $G^\geq(T) = G^>(T) \cup \{T\}$
Soit G un graphe de n tâches et F l'ensemble des tâches falsifiées de G. On définit l'*ensemble des initiateurs* noté $\mathcal{I}(F)$ comme l'ensemble des tâches T_i de F qui n'ont pas de prédecesseurs dans F, i.e:

$$\mathcal{I}(F) = \{T_i \in F : F \cap G^<(T_i) = \emptyset\}$$

Bien entendu, on suppose que les ensembles F et donc $\mathcal{I}(F)$ ne sont pas connus car sinon, la certification serait triviale. Le lemme suivant donne néanmoins une caractérisation des tâches falsifiées à partir de ces ensembles:

Lemma 6. *Soit* F *l'ensemble des tâches falsifiées de* G *et* $T \in G$. *Si les résultats de* T *sont incorrects, alors on a:*

$$G^\leq(T) \cap \mathcal{I}(F) \neq \emptyset.$$

Proof. Si les résultats de T sont incorrects alors soit T est initiatrice i.e. $T \in \mathcal{I}(F)$, soit il y a au moins une tâche initiatrice dans l'ensembles des tâches précédents T, i.e $G^<(T)$. □

Ainsi, déterminer si le résultat d'une tâche T est falsifié ou non demande de vérifier toutes les tâches de $G^<(T)$, ce qui peut correspondre à peu ou beaucoup de tâches selon le type de graphe considéré. Ce principe est utilisé dans le cadre de l'algorithme $EMCT(E)$ présenté au §5.4.
Nous fournissons maintenant une borne inférieure pour le nombre d'initiateurs d'un graphe G possedant n_F tâches falsifiées.

5.3. Nombre minimum d'initiateurs dans un graphe

Parmi les tâches falsifiées, on dit qu'une tâche est "falsifiée de manière démontrable" ou "ostensiblement falsifiée"[3] $\iff o(T, E) \neq \hat{o}(T, E)$, ce qui peut être vérifié par ré-exécution de la tâche sur une machine fiable. Le nombre minimal d'initiateurs est donc une borne inferieure du nombre de tâches ostensiblement falsifiées dans le graphe. Ces tâches sont particulièrement importantes dans la détection car elles caractérisent de façon déterministe une tâche falsifiée. Nous fournissons maintenant une borne pour le nombre minimum d'initiateurs dans un graphe. Cet borne exploite la définition suivante:

Definition 9 (Nombre minimal d'initiateurs relatif à k). *On définit le nombre minimal d'initiateurs relatif à* $k \leq n_F$, *noté* $\gamma_G(k)$, *par*

$$\gamma_G(k) = \min_{F \in G / |F| \geq k} |\mathcal{I}(F)|$$

[3] "provably falsified" ou "observably falsified" en anglais

En effet, connaissant le graphe G, un attaquant malicieux peut tenter de minimiser la visibilité d'une attaque massive de rapport q. Du point de vue de l'attaquant, il est avantageux de "cacher" les tâches falsifiées parmi le graphe des successeurs de tâches bien choisies, afin de minimiser le nombre d'initiateurs. Ainsi, $\gamma_G(n_F)$ correspond au nombre minimal d'initiateurs nécessaires pour cacher les n_F tâches falsifiées. Nous fournissons maintenant une borne inférieure à ce nombre qui correspond pourtant au cas le plus pathologique.

Lemma 7. *Soit* G *un graphe de hauteur* h *(il s'agit de la longueur d'un chemin critique) et de degré maximal* d *attaqué massivement avec un rapport* q. *Alors* $\gamma_G(n_F) = \left\lceil \frac{n_F}{\left(\frac{1-d^h}{1-d}\right)} \right\rceil$ *et le nombre minimum d'initiateurs dans* G *est* $\gamma_G(n_q)$

Proof. Soit T_i une tâche de G. La taille maximale du graphe $G^{\geq}(T_i)$ est majorée par $|G^{\geq}(T_i)| \leq 1 + d + d^2 + \ldots + d^{h-1} = \frac{1-d^h}{1-d}$. Ainsi, si T_i est une tâche initiatrice, alors T_i peut "cacher" au plus $\frac{1-d^h}{1-d}$ des n_F tâches falsifiées. Cela fera des tâches de $G^{>}(T_i)$ des tâches non-initiatrices (par définition). Donc, si chaque initiateur peut "cacher" au maximum $\frac{1-d^h}{1-d}$ tâches de F, alors le nombre minimum de tels initiateurs est $\lceil \frac{n_F}{\left(\frac{1-d^h}{1-d}\right)} \rceil$, et cette quantité est minimale pour $n_F = n_q$. □

Compte tenu de ce résultat, nous proposons maintenant l'algorithme $\mathsf{EMCT}(E)$ adapté aux graphes possédant des sous-graphes de prédécesseurs petits. Cet algorithme est très similaire à $\mathsf{MCT}(E)$ mais vérifie toutes les tâches de $G^{\leq}(T)$.

5.4. Algorithme $\mathsf{EMCT}(E)$

Soit E une exécution comportant n tâches avec dépendances.

1. Choisir uniformément une tâche T dans E. Les entrées (resp. les sorties) de T dans E sont $i(T, E)$ (resp. $o(T, E)$).

2. réexecuter T sur des ressources fiables, en utilisant les entrées de E, i.e. $i(T, E)$, pour obtenir les sorties $\hat{o}(T, E)$. Si $o(T, E) \neq \hat{o}(T, E)$ renvoyer "falsifiée"
   Sinon, pour chaque tache V dans $G^{<}(T)$ qui n'a pas déjà été vérifiée, réexecuter V sur des ressources fiables. Si $o(V, E) \neq \hat{o}(V, E)$ alors renvoyer "falsifiée".

3. Ici, Toutes les tâches de $G^{\leq}(T)$ ont été vérifiées. Renvoyer "correcte"

Soit C le coût de la certification d'une exécution par $\mathsf{EMCT}(E)$ en terme de nombres de tâches réexécutée. Alors

$$C = \frac{\sum_{T \in G} \left|G^{\leq}(T)\right|}{|G|} = \frac{\sum_{T \in G} \left|G^{\leq}(T)\right|}{n}$$

Nous nous intéresserons également dans la suite au coût en temps C_t de cet algorithme. A cet effet, nous utiliserons les notations suivantes: T_1 désignera le temps séquentiel d'exécution (i.e la somme des durées de toutes les tâches) et T_∞ la durée critique (i.e la somme maximale des durées des tâches d'un chemin critique). Donc, si $t(G)$ désigne le temps d'exécution d'un graphe G, alors $T_\infty \leq t(G) \leq T_1$.
Le théorème suivant fournit la probabilité d'erreur de $\mathsf{EMCT}(E)$ sur un graphe G.

Theorem 3. *Lors de l'exécution de l'algorithme* $\mathsf{EMCT}(E)$, *la probabilité d'erreur de certification est* $e_E \leq 1 - q$.

Proof. La probabilité que $G^{\leq}(T)$ contienne une tâche falsifiée de façon démontrable est $\frac{n_q}{n}$ dans le pire cas. On en déduit $e_E \leq 1 - \frac{n_q}{n} \leq 1 - q$ □

Dans notre contexte d'application ou les graphes considéré sont principalement des arbres, l'algorithme $\mathsf{EMCT}(E)$ est particulièrement efficace. Les sections qui suivent le démontre en détaillant le coût moyen (en nombre de tâches et en temps) de $\mathsf{EMCT}(E)$ sur ce type de graphe.

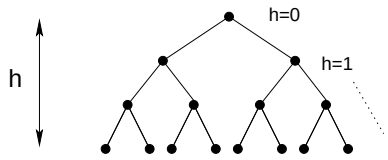

Figure 3: Arbre d'arité $a = 2$, de hauteur $h = 4$

5.4.1. Etude du coût de $\mathsf{EMCT(E)}$ sur un arbre d'arité a et de hauteur h

Nous supposons donc ici que l'algorithme $\mathsf{EMCT(E)}$ est appliqué à un arbre G d'arité a et de hauteur h. La figure 3 fournit un exemple d'un tel arbre.

Par convention, la hauteur du noeud racine est 0. Dans ce cas, $n = |G| = \sum_{k=0}^{h} a^k = \frac{a^{h+1}-1}{a-1}$.

Lemma 8. *Soit* $T \in G$. *On note* $h(T)$ *la hauteur de* T *dans l'arbre. Alors* $\left|G^{\leq}(T)\right| = h(T)+1$ *et* $t(G^{\leq}(T)) \leq T_\infty$.

Proof. En effet, $G^{\leq}(T)$ correspond au chemin de T à la racine de l'arbre. □

Le théorème suivant fournit le coût moyen de $\mathsf{EMCT(E)}$ sur un arbre:

Theorem 4. *Soit* E *l'exécution d'un arbre complet d'arité* $a \geq 2$, *de hauteur* h. *Le coût moyen en nombre de tâches de* $\mathsf{EMCT(E)}$ *est* $C = \frac{(h+1)a^{h+1}}{a^{h+1}-1} - \frac{1}{a-1} = \Theta(h) = \Theta(\log_a|G|)$
En temps, le coût moyen de $\mathsf{EMCT(E)}$ *vérifie* $C_t = \mathcal{O}(T_\infty)$

Proof. Du lemme 8, on déduit:

$$\sum_{T\in G} \left|G^{\leq}(T)\right| = \sum_{k=0}^{h} \sum_{T\in G/h(T)=k} \left|G^{\leq}(T)\right| = \sum_{k=0}^{h} a^k(k+1) = \frac{1}{(a-1)^2}\left[(a-1)(h+1)a^{h+1} - (a^{h+1}-1)\right]$$

Et donc, puisque $n = \frac{a^{h+1}-1}{a-1}$: $C = \frac{\sum_{T\in G}\left|G^{\leq}(T)\right|}{n} = \frac{(h+1)a^{h+1}}{a^{h+1}-1} - \frac{1}{a-1} \approx h = \Theta(\log_a|G|)$
Pour le coût moyen en temps, on a, toujours d'après le lemme 8:
$C_t = \frac{1}{n}\sum_{T\in G} t\left(G^{\leq}(T)\right) \leq \frac{1}{n}\left(n.T_\infty\right)) \leq T_\infty \Longrightarrow C_t = \mathcal{O}(T_\infty)$ □

Le coût en nombre de tâches est illustré dans la figure 4 où a été représenté son évolution en fonction du nombre de noeuds n de l'arbre. Une plus grande arité fait baisser le coût.

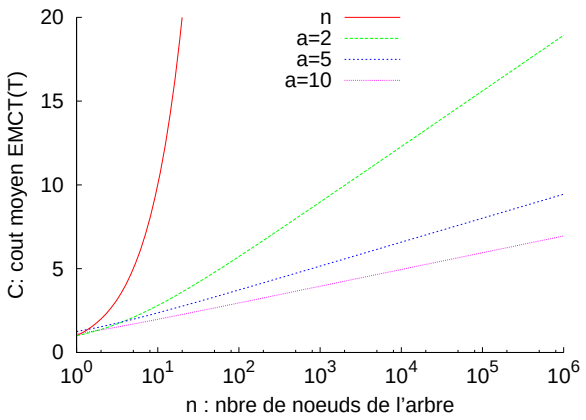

Figure 4: Cout moyen en nombre de tâches de $\mathsf{EMCT(E)}$ sur un arbre d'arité a, de hauteur h et possédant n noeuds

5.4.2. Etude du coût de $\mathsf{EMCT(E)}$ sur un arbre "inversé" d'arité a et de hauteur h

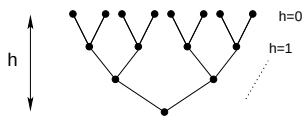

Figure 5: Arbre inversé d'arité $a = 2$, de hauteur $h = 4$

La figure 5 illustre ce que nous entendons par arbre "inversé". On suppose donc qu'on applique l'algorithme $\mathsf{EMCT(E)}$ à un tel arbre G. Nous montrons dans cette section que pour un tel arbre, le coût moyen en nombre de tâche est identique à celui trouvé au §5.4.1. En revanche, la certification est plus coûteuse en temps. Ces résultats se basent sur le lemme suivant:

Lemma 9. *Soit* $T \in G$. *On note* $h(T)$ *la hauteur de* T *dans l'arbre. Alors* $\left|G^{\leq}(T)\right| = \frac{a^{h(T)+1}-1}{a-1}$.
De plus, $t(G^{\leq}(T)) = \frac{a^{h(T)+1}-1}{a-1} + a^{h(T)}(T_\infty - h - 1)$.

Proof. En effet, $G^{\leq}(T)$ correspond à un arbre inverse d'arité a et de hauteur $h(T)$. Cet arbre contient $\sum_{k=0}^{h(T)} a^k = \frac{a^{h(T)+1}-1}{a-1}$ tâches. Pour le coût en temps, il faut considérer la pire configuration possible sur le temps respectif des tâches qui compose G. Cette configuration suppose que les tâches les plus couteuses en temps sont à coup sûr exécutées dans toute vérification. Une telle configuration est obtenue en supposant que toutes les tâches de G ont une durée de 1, sauf celles de hauteur $h = 0$ de durée $T_\infty - h$. Dans cette configuration, $t(G^{\leq}(T)) = \sum_{k=0}^{h(T)-1} a^k + a^{h(T)}(T_\infty - h) = \sum_{k=0}^{h(T)} a^k + a^{h(T)}(T_\infty - h - 1)$. □

Le théorème suivant fournit le coût moyen de $\mathsf{EMCT(E)}$ sur un arbre inversé:

Theorem 5. *Soit* E *l'exécution d'un arbre inversé complet d'arité* $a \geq 2$, *de hauteur* h. *Le coût moyen en nombre de tâches de* $\mathsf{EMCT(E)}$ *est* $C = \frac{(h+1)a^{h+1}}{a^{h+1}-1} - \frac{1}{a-1} = \Theta(h) = \Theta(\log_a|G|)$
En temps, le coût moyen de $\mathsf{EMCT(E)}$ *vérifie* $C_t = \mathcal{O}(T_\infty^2)$

Proof. Du lemme 9, on déduit:

$$\sum_{T\in G}\left|G^{\leq}(T)\right| = \sum_{k=0}^{h}\sum_{T\in G/h(T)=k}\left|G^{\leq}(T)\right| = \frac{1}{a-1}\sum_{k=0}^{h} a^{h-k}(a^{k+1}-1) = \frac{(a+1)(h+1)a^{h+1}-(a^{h+1}-1)}{(a-1)^2}$$

On retrouve ainsi sur la même formule que celle trouvée dans la preuve du théorème 4.
Pour le coût en temps, on a (en posant $T'_\infty = T_\infty - h - 1$):

$$C_t = \frac{1}{n}\sum_{k=0}^{h}\sum_{T\in G/h(T)=k} t\left(G^{\leq}(T)\right) = \frac{1}{n}\sum_{k=0}^{h} a^{h-k}\left[\frac{a^{k+1}-1}{a-1} + a^k.T'_\infty\right] = C + \frac{(h+1)a^h}{n}T'_\infty$$

Or, puisque $a^h = \frac{(a-1)n+1}{a}$, $C_t = C + \frac{(h+1)((a-1)n+1)}{a.n}T'_\infty \leq C + (h+1)T'_\infty = \mathcal{O}(h.T_\infty) = \mathcal{O}(T_\infty^2)$ □

6. Conclusion

Cet article adresse le problème de la certification de résultats d'application largement distribuées et s'exécutant dans un environnement hostile. Dans ce contexte, les tâches qui composent l'application ou les données qui stockent les paramètres (d'entrée ou de sortie) de ces tâches peuvent avoir été manipulées par un attaquant.
Alors que les approches précédentes ne considéraient que des applications composées de tâches indépendantes, nous étudions dans cet article les effets de la propagation d'erreurs qui se produit avec des

tâches dépendantes. Dans notre objectif de décider si une exécution a été massivement attaquée ou non, nous proposons une approche probabiliste ne faisant aucune hypothèse sur le comportement des fautes et de l'attaque en générale. Cela nous permet de gérer les attaques qui ne se comportent pas comme des processus de Bernouilli.

Deux algorithmes probabilistes ont été introduit pour permettre de détecter efficacement une attaque massive : le nombre de tâches à ré-exécuter sur des ressources fiables reste faible tout en garantissant une indication de résultat correct avec une probabilité d'erreur majorée par un seuil fixé par l'utilisateur. La ré-éxécution de tâche est basée sur l'utilisation de paramètres d'entrées disponibles à travers un serveur de checkpoint considéré comme sûr et sécurisé, qui stocke le graphe macro data-flow de l'exécution.

L'introduction de la notion d'initiateurs a permis de traiter judicieusement le problème associé à la propagation d'erreurs dans un graphe avec dépendances. Notre approche utilisant le concept des initateurs a permit d'aboutir à l'algorithme EMCT(E) pour lequel le surcoût de la certification reste notable pour un graphe général. Néanmoins, nous avons montrer que ce coût reste mineur dans le contexte d'applications réelles représentées par des arbres, comme c'est le cas pour les applications d'analyses d'images médicales qui motivent nos recherches.

Nos travaux actuels consistent à étendre cette approche pour prendre avantage de la détection d'attaques massives dans la correction en temps réelle de l'exécution.

Les auteurs tiennent à remercier Jang Schiltz de l'Université du Luxembourg pour sa contribution scientifique.

Bibliographie

1. CERT/CC Statistics 1988-2004, CERT Coordination Center, http://www.cert.org/stats/cert_stats.html
2. Foster, I. and Kesselman, C. and Nick, J. and Tuecke, S., *Grid Services for Distributed System Integration*, IEEE Computer, No. 6, Vol. 35, 2002
3. Foster, I. and Kesselman, C. and Tsudik, G. and Tuecke, S., *Security for Grid Services*, Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), 22-24 June, Seattle, Washington, 2003
4. Fujiwara Hideo, *Logic Testing and Design for Testability*, MIT Press, 1985
5. Germain, C. and Playez, N., *Result Checking in Global Computing Systems*, Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS 03), San Francisco, California, 23-26 June, 2003
6. Gao, L. and Malewicz, G., *Internet Computing of Tasks with Dependencies using Unreliable Workers*, 8th International Conference on Principles of Distributed Systems (OPODIS'04), Dec., 15-17, 2004, to appear in Springer Verlag LNCS.
7. Jafar S., Varrette S., Roch J.-L., *Using Data-Flow Analysis for Resilence and Result Checking in Peer to Peer Computations*, Proceedings of the 15th International Workshop on Database and Expert Systems Applications (IEEE DEXA 2004), Zaragoza, Espagne, 30th Aug. - 3rd Sep., 2004, pp. 512-516
8. Montagnat, J. and Breton, V. and Magnin, I., *Partitioning medical image databases for content-based queries on grid*, Methods of Information in Medicine, Special Issue HealthGrid04, to appear, 2004
9. Ragtime: Grille pour le Traitement d'Informations Médicales, Région Rhône-Alpes http://liris.univ-lyon2.fr/ miguet/ragtime/
10. Sarmenta, Luis F.G., *Sabotage-Tolerance Mechanisms for Volunteer Computing Systems*, Future Generation Computer Systems, Elsevier Publishing, No. 4, Vol. 18, 2002
11. Hal Wasserman and Manuel Blum, Software reliability via run-time result-checking, Journal of the ACM, Vol. 44, No. 6, 1997, pp. 826-849
12. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, C. Press, Ed. CRC Press, Inc, 1997.
13. F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line Building Data Flow Graph in a Parallel Language. In IEEE, editor, *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris, France, Octobre 1998.
14. M.C. Rinard. The design, implementation and evaluation of Jade : a portable, implicitly parallel programming language. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, Mai 1998.