La mesure de performance dans les cartes à puce

Julien Cordry

CNAM
Cedric

Département de formation doctorale en informatique

École Doctorale Informatique Télécommunication et Électronique
(EDITE)

La Mesure de Performance dans les Cartes à Puce

THÈSE

présentée et soutenue publiquement le 30 Novembre 2009

pour l'obtention du

Doctorat du Conservatoire National des Arts et Métiers
(spécialité informatique)

par

Julien Cordry

Composition du jury

Président :	Luc Bouganim
Rapporteurs :	Didier Donsez Serge Chaumette
Examinateur :	Pascal Urien
Directeur :	Pierre Paradinas
Co-encadrante :	Samia Bouzefrane

Centre d'Étude et de Recherche en Informatique du CNAM

Remerciements

Mes très vifs remerciements à monsieur Serge Chaumette, et à monsieur Didier Donsez pour m'avoir fait l'honneur d'accepter d'être les rapporteurs. La pertinence de leurs remarques et leurs expériences ont indubitablement enrichi ce document. Merci pour l'intérêt que vous y avez accordé.

Mes très vifs remerciements vont aussi à monsieur Luc Bouganim et à monsieur Pascal Urien pour avoir porté attention à mes travaux et pour avoir accepté d'encadrer ma soutenance.

Je tiens à remercier monsieur Pierre Paradinas, d'avoir accepté la responsabilité de diriger cette thèse. Il m'a permi de découvrir le milieu de la carte à puce et le milieu de la recherche. La richesse de ses points de vue, sa persévérance et sa patience ont rendu ces travaux possibles.

Je ne remercierai jamais assez madame Samia Bouzefrane pour ses efforts continus pendant des années autour de cette thèse. Il est difficile de la remercier à la hauteur de l'énergie qu'elle dépensée. Elle a joué un rôle prépondérant dans la réalisation de ces travaux. Sans elle, rien n'aurait été possible. Pour tout cela, merci.

Je voudrais aussi exprimer ma plus profonde gratitude à monsieur Éric Gressier-Soudan qui m'a accueilli au sein de son équipe. Ses conseils et ses remarques ont profondément influencé mon travail. Son entrain, sa bonne humeur, son énergie et son écoute ont grandement contribué aux bonnes conditions de travail qui ont été les miennes au cours de ces dernières années.

Un certain nombre de personnes du Cedric ont, de part leurs questions et leurs remarques, contribué à ces travaux. Je remercie donc François Anceau, Joël Berthelin, Nicolas Bouillot, Ivan Boule, Catherine Coquery, Pierre Courtieu, Séverine Demeyer, Jean-Paul Etienne, Claude Kaiser, Romain Pellerin, Gilbert Saporta, Françoise Sailhan et Jean-Ferdinand Susini pour les discutions toujours enrichissantes que j'ai pu avoir avec eux.

Je remercie tout particulièrement Stéphane Natkin et Marie-Christine Costa de m'avoir accueilli au Cedric.

Je remercie également mes collègues du projet Mesure : Corentin Boë, Gilles Grimaud, Hervé Meunier, Carine Pascal, Henri Pied, Sébastien Ronsse, Ernest Tsassong et Éric Vétillard.

Merci aussi à mes collègues et amis du Cedric : Jean-Frédéric Etienne, Patrice Krzanik, Haï-Binh Le, Patrick Jocelin, Khaled Garri, Amélie Lambert, Hélène Topart, Leïla Harfouche, Safia Sider, Henri Pugliese, Jean-Marie Schroeder, Laurent Dehoey, Guozhi Wei, Anne Wei, Samundeswari Ramachandra, Hans-Nikolas Locher, Marc Flausino, Rodrigo Almeida, Pedro Alessio, Xiangqiu Hou, Olivier Boursin, Didier Erepmoc, Vincent Roudaut, Shuohsiu Hsu, Fouad Keyrillos, Viviane Gal, José Pluquet et Mathieu Trampont.

Merci à tout le personel administratif du CNAM qui a travaillé sans compter pour rendre cette thèse possible.

Enfin, mille mercis à Jeannine, Véronique, Alain, Leslie et Morgane Cordry, ainsi qu'à Jérome Alonso, Fanny Boitier, Mylène Bouat, Ghislaine Carton, Yohan Edier, Sarah Gayrard, Raphaëlle Giardini, Patricia Hillion, Yves Keroas, Stéphanie Lebas, Cindy Meister, et Gaëlle Mooradaly.

À Chen

Résumé

La mesure de performance est utilisée dans tous les systèmes informatiques pour garantir la meilleure performance pour le plus faible coût possible. L'établissement d'outils de mesures et de métriques a permis d'établir des bases de comparaison entre ordinateurs. Bien que le monde de la carte à puce ne fasse pas exception, les questions de sécurité occupent le devant de la scène pour celles-ci. Les efforts allant vers une plus grande ouverture des tests et de la mesure de performance restent discrets. Les travaux présentés ici ont pour objectif de proposer une méthode de mesure de la performance dans les plates-formes Java Card qui occupent une part considérable du marché de la carte à puce dans le monde. Après l'introduction de méthodologies de mesures de performance pour les cartes à puce, nous choisirons les outils et les caractéristiques des tests que nous voulons faire subir aux cartes, et nous analyserons les données ainsi récoltées. Enfin une application originale des cartes à puce est proposée et permet de valider certains résultats obtenus.

Mots-clés: Java Card, *benchmark*, carte à puce, mesure de performance, test

Abstract

Performance measurements are used in computer systems to garanty the best performance at the lowest cost. Establishing measurement tools and metrics has helped build comparaison scales between computers. Smart cards are no exception. But the center stage of the smart card industry is mostly busy with security issues. Efforts towards a better integration of performance tests are still modests. Our work focused on a better approach in estimating the execution time within Java Card platforms. Those platforms constitute a big part of the modern smart card market share especially with regards to multi-applicative environments. After indroducing some methologies to better measure the performance of Java Cards, we detail the tools and the tests that we mean to use on smart cards. We will thereafter analyze the data obtained in this way. Finally, an original application for smart cards is proposed. We used it to validate some points about the results.

Keywords: Java Card, benchmark, smart card, performance measurement, test

Table des matières

Chapitre 2
État de l'art

Chapitre 3
Méthodologie pour la mesure de Performance

Table des figures

Introduction générale

Motivation

Depuis ses débuts il y a plus de 30 ans, la carte à puce est devenue un objet omniprésent de la vie quotidienne de milliards de personnes. Ses domaines d'applications sont devenus très diversifiés, puisqu'on trouve des cartes à puce dans les télécommunications, et en particulier dans la téléphonie mobile, dans le domaine bancaire, dans les transports, dans l'identité, dans la télévision payante, dans la santé, dans la fidélisation. Les volumes impliqués sont si importants que l'on peut parler de la carte à puce comme du type d'ordinateur de plus répandu au monde.

Les particularités d'une carte à puce, telles que sa fiabilité, sa facilité de transport, sa résistance aux contraintes physiques, et ses capacités cryptographiques intégrées font d'elle un outil idéal pour accéder, valider et consommer un service, pour garantir une certaine confidentialité des informations sur le porteur.

Les contraintes de sécurité sur ces objets sont souvent importantes, mais d'autres facteurs peuvent amener une entreprise à choisir une plateforme plutôt qu'une autre, notamment le prix, la taille de la mémoire et la performance. Celle-ci peut être cruciale pour certains projets, par exemple dans les transports, où une contre-performance chronique peut signifier un échec de développement.

Ce type de problème est d'autant plus exacerbé que depuis plus de 10 ans, les plates-formes multiapplications ont fait leur apparition avec notamment Java Card. Cela dénote une très nette rupture dans les techniques de programmation pour la carte à puce, car les programmeurs sont passés d'un paradigme où les concepteurs du matériel de la carte et les concepteurs logiciels étaient fortement liés et où la programmation pouvaient se faire dans un langage approprié à la plateforme (et souvent dans un langage bas niveau), à un paradigme où les applications devenaient accessibles à la programmation par des personnes ne connaissant pas exactement les spécifications de la carte. Les cartes à puce devaient alors intégrer des machines virtuelles qui étaient souvent estimées comme étant beaucoup trop volumineuses et trop gourmandes en calcul, et ce pour des cartes plus coûteuses tout en gardant leurs propriétés de sécurité et sans permettre à une application de compromettre la sécurité des données d'une autre.

Aujourd'hui, ce type de plates-formes participe activement à l'essor des cartes à puce. Cette industrie est florissante et s'ouvre sur de nombreux sujets de recherche liés à ces caractéristiques multiapplicatives et sécuritaires. L'adoption des plates-formes Java Card est maintenant une chose acquise dans l'industrie. Mais après plusieurs années d'évolution et de perfectionnement, il est devenu indispensable de disposer d'un outil de référence dans la mesure de leurs performances.

L'apparition de programmes de benchmark est souvent un signe de la maturité d'une technologie, et l'état de ces benchmarks montre l'évolution des besoins par rapport à cette technologie. Les travaux sur la mesure de performance pour Java Card reflètent le besoin d'un outil de benchmark plus complet, et surtout qui puisse être utilisé de manière transversale dans les différents secteurs de l'industrie. Il est évident, d'après les travaux publiés, et d'après les conversations que

nous avons pu avoir que les acteurs principaux de l'industrie sont avides de connaître les performances de leurs cartes, ou des cartes de certains fabricants, qui peuvent être leurs concurrents ou leurs partenaires.

Il est fondamental de souligner que les performances d'une carte à puce ne sont pas tout. La sécurité sur une carte peut elle aussi être une qualité première. Cela dépend avant tout du domaine d'application dans lequel cette carte à puce va servir. Un compromis doit être trouvé entre plusieurs facteurs, dont la performance et la sécurité.

Le projet MESURE

Le projet MESURE a été financé par l'Agence Nationale pour la Recherche (ANR) avec pour but de palier au manque d'outils de mesure de performance dans les cartes à puce. Le projet a été sélectionné par l'ANR en 2005, il a commencé en mai 2006 et s'est terminé en mars 2008, et il a impliqué trois partenaires : le CNAM-Cédric situé à Paris, comme équipe directrice du projet, la société Trusted Labs de Sophia Antipolis et l'équipe POPS (Petits Objets Portables et Sécurisés) de l'INRIA à Lille.

Le but de ce projet est de combler le manque perçu d'outils de benchmark qui soient acceptés par la communauté de la carte à puce. Le projet se propose de fournir, entre autres, un programme source qui couvre la mesure de performance sur la carte à puce. L'essentiel étant de mesurer ce qui a de la pertinence pour le monde de la carte à puce. Le projet a aussi été pensé pour être facilement adaptable pour tous les systèmes d'exploitation et pour cibler la plupart des cartes.

L'idéal pour le projet serait de ne pas proposer un outil qui soit trop encré dans un environnement de test spécifique. De nombreux travaux concernant la sécurité sont axés autour d'outils qui fournissent des résultats très précis pour procéder à des attaques par canaux cachés sur les cartes à puce. Le but du projet n'est pas de fournir un tel outil, mais simplement d'évaluer aussi exactement et précisément que possible les performances des cartes à puce en fournissant un outil de mesure portable et utilisable simplement sans avoir besoin d'un environnement de mesure conçu spécifiquement pour les besoins de la mesure. La philosophie derrière ces outils est qu'on doit pouvoir les créer indépendamment les uns des autres, mais qu'ils puissent fonctionner en complémentarité, et qui permettent à partir de mesures élémentaires de retrouver les données pertinentes de la performance d'une carte. Comme tout benchmark, il nous faudra discuter de la note attribuée au résultat final et qui sera calculée en fonction d'une référence correspondant aux performances en général des cartes à puce utilisées dans le projet.

Puisque la sécurité des cartes à puce n'est pas notre objectif, et pour conserver un secret relatif concernant les cartes mesurées, les notes et les performances brutes des cartes utilisées sont entièrement anonymisées. En effet, si des résultats très précis étaient collectés de manière empirique par des utilisateurs cela pourrait être utilisé pour mettre à jour des attaques sur certaines cartes à puce. Le but du projet est également de proposer un benchmark qui soit accepté comme étant un référentiel de la performance des cartes à puce par l'industrie. Si le projet expose des vulnérabilités dans les cartes de certains fabricants, l'outil pourrait être rejeté par ces mêmes fabricants, et ce rôle de référentiel serait compromis. En effet, si une carte a des performances anormalement élevées dans seulement une des caractéristiques testées, on pourrait penser que la portion de code qui correspond à cette caractéristique s'exécute dans un environnement dégradé qui n'offre pas toutes les qualités de sécurité attendues.

Très peu de travaux dans le monde de la carte à puce sont développés avec une licence libre. Le projet MESURE fait exception à cette règle. Afin de ne pas empêcher la diffusion de l'outil une licence CeCILL v2 a été utilisée. Cette licence permet de partager et de réutiliser le projet

dans un autre contexte. L'idée est que les entreprises intègrent MESURE à d'autres projets.

Les travaux entrepris ici ont été centrés autour des méthodologies à utiliser pour isoler les performances de parties de code intéressantes.

Un certain nombre d'outils ont été conçus sans avoir été nécessairement prévus au départ. Un exemple est un outil permettant de calibrer les mesures, un autre permettant d'extraire les résultats obtenus avec des mesures brutes. Un besoin indispensable existe derrière toutes les mesures : l'analyse. Il est nécessaire de caractériser les résultats et d'en tirer un sens. Un autres aspect abordé est la représentativité des mesures. Les mesures doivent avoir un sens vis-à-vis de l'usage qui est fait des applications. Notre famille d'outils doit aussi être capable de résumer les performances d'une carte à puce avec des notes de synthèse. Le sujet de cette thèse correspond à la problématique de mesure dans la carte à puce. Le projet MESURE a permis de financer un peu plus d'un an de cette thèse.

Contributions

Une des tâches premières de cette thèse a été de faire l'état de l'art de la mesure de performance dans les cartes à puce. Les travaux concernés sont globalement divisés en deux catégories : ceux qui tentent de se focaliser sur la faisabilité d'une application sur une famille de cartes données et ceux qui tentent d'être plus généraux dans les mesures effectuées, mais qui s'éloignent d'un contexte d'application réaliste.

J'ai pu mettre au point une méthodologie de la mesure de performance qui se fonde en partie sur certains travaux provenants d'autres auteurs et qui tentent d'être les plus généraux possible. L'idée principale derrière ces travaux est celle de l'isolement du temps d'exécution d'un fragment de code considéré comme étant digne d'intérêt sur la carte. J'ai réutilisé cette idée mais en apportant plusieurs choses. J'ai en particulier raffiné la granularité du fragment de code en question, ce qui nous permet de nous concentrer sur des points plus précis d'un code dont la performance nous interresse.

Il est essentiel, quand on fait des mesures de performance, d'en tirer une note de synthèse qui représente la plateforme testée et qui soit réaliste dans un contexte d'application donné. Il est donc important de présenter une méthode pour intégrer les mesures et pour en extraire des informations pertinentes. J'ai pu relier les mesures individuelles avec des applications de certains domaines. Ainsi, j'ai fait l'étude d'une application du domaine public, MCardApplet. Cette application a été développée dans une optique d'identification du porteur. Pour une utilisation "normale" de cette applet, j'ai pu tirer un modèle d'utilisation de chaque bytecode et de chaque appel à une méthode de l'API.

Depuis la fin du projet MESURE en mars 2008, les recherches ont porté sur la validation des mesures en utilisant un lecteur de précision et des méthodes statistiques pour corroborer les résultats obtenus avec des lecteurs et des systèmes moins exacts. Les bruits perçus pendant les mesures de performance sont un problème que l'on ne peut en générale pas entièrement éliminer. Un problème majeur dans la conception d'un outil de mesure de performance est de savoir si l'outil est capable d'estimer le bruit et de présenter des mesures significatives malgré le bruit.

J'ai par ailleurs eu l'occasion de développer une application originale dans le cadre d'un concours. Cette application porte sur les jeux multijoueurs ubiquitaires, ce qui est un domaine où les cartes sont rarement utilisées. Cette application m'a permis de mettre en pratique MESURE pour vérifier la faisabilité d'une telle application sans avoir un environement de test complet, et pour mettre en relation les mesures de temps d'exécution de codes spécifiques et celle d'une application plus générale.

Plan du document

Cette thèse est organisée de la façon suivante :

- **Le chapitre 1**
  Ce chapitre présente les différentes étapes de l'histoire de la carte à puce et les caractéristiques principales de ces plates-formes. Cela inclut les standards internationaux, les caractéristiques physiques des cartes à puce, quelques détails sur les microprocesseurs, la communication avec une carte à puce, et le cycle de vie d'une carte. De plus Java Card est introduit, ainsi que quelques plates-formes et outils communément utilisés dans un contexte orienté vers les cartes.
- **Le chapitre 2**
  L'état de l'art est présenté dans ce chapitre. On y parle auparavant de ce qui est fait en termes de benchmarking dans un contexte général sur ordinateur. Les caractéristiques de l'évaluation de performance y sont évoquées et on y parle des types de benchmarks, de métriques de performance et d'évaluation de la mesure. Puis on évoque plus en détails ce qui est fait sur les plates-formes Java Card.
- **Le chapitre 3**
  Ce chapitre présente une méthode d'isolation d'une portion de code avec des mesures de temps d'exécution. Cette méthode est appliquée à une certaine partie des bytecodes. Même si la totalité des bytecodes ne peut pas être couverte avec cette méthode, les bytecodes non couverts peuvent être isolés par paires. L'API Java Card présente elle même des méthodes dont le temps d'exécution est isolable par une méthode similaire. On présente quelques résultats sur des bytecodes d'arithmétiques et on vérifie que les résultats résistent à un passage à l'echelle.
- **Le chapitre 4**
  Les outils du projet MESURE sont présentés dans ce chapitre. MESURE a été développé sous la formes de plusieurs modules indépendants. Certains de ces modules ont été rajoutés en cours de projet, d'autres se sont révélés peu utiles à la complétion du projet. Ce caractère modulaire s'est donc révélé crucial.
- **Le chapitre 5**
  Une analyse des données obtenues pendant le projet est présentée ici. Ce type de travail est classique dans la mesure de performance. Le but est de caractériser les résultats. Des problèmes de correction des mesures sont aussi explorés ici. On compare les résultats obtenus expérimentalement sur PC et des résultats obtenus sur une plateforme dédiée.
- **Le chapitre 6**
  Ce chapitre présente un type d'application particulièr qui sort du cadre d'application classique des cartes à puce. Le développement de cette application est l'occasion de présenter une utilisation de MESURE dans un contexte nouveau et de montrer l'impact de la mesure de performance sur l'étude de faisabilité d'une application. On y présente aussi une validation des résultats de MESURE.
- **Le chapitre 7**
  Dans ce chapitre, on résume les contributions du projet MESURE, et les efforts qui ont été faits du point de vue recherche. Une partie sur Java Card 3.0 présente quelques éléments de recherche laissés en suspend essentiellement par manque de matériel pour effectuer des tests.
- **Les Annexes**
  On y parle des travaux de recherche qui ont donné lieu à des publications, des stagiaires pris en charge pendant la thèse. Des résultats sont présentés pour illustrer l'intérêt d'utiliser

un lecteur de précision dans le chapitre 5. On y présente aussi les résultats obtenus par le profilage de l'application MCardApplet qui est présentée dans le chapitre 1. Quelques distributions d'un même test sont aussi présentées, afin d'approfondir les problèmes rencontrés pendant l'analyse des données.

1

Les cartes à puce

Sommaire

La technologie des cartes à puce a considérablement évolué depuis ses débuts. Les capacités individuelles des cartes se sont améliorées en termes de facultés matérielles et de technologie logicielle. De nouveaux usages sont continuellement introduits et développés. Le but de ce chapitre est de présenter l'histoire et les principales caractéristiques de la technologie des cartes à puce. On y discutera en outre du cycle de vie d'une carte, des différents systèmes d'exploitation et plates-formes et en particulier des plates-formes Java Card.

1.1 Histoire

Cette section présente brièvement les étapes de développement qui partent de l'émergence de simples cartes en plastique aux cartes à puce multi application qui sont communes aujourd'hui.

Le développement de matériaux synthétiques a permis la production de cartes en plastique à bas coûts dès le début des années 50. Ces cartes offraient une alternative durable aux cartes faites jusque là en papier ou en carton. C'est vers cette époque que Diners Club a émis aux États Unis une carte permettant à son propriétaire d'effectuer des paiements qui avait pour seule valeur la force de son nom. Elle était acceptée dans des hôtels et dans des restaurants et devint un symbole de statut pour son propriétaire. Le concept à été accepté et amélioré par VISA et MasterCard qui répandirent les cartes de crédit aux États Unis. L'Europe et le reste du monde suivirent peu après. Le passage de cartes de membre de clubs réservées à une élite à des cartes de crédit largement répandues s'est fait en amenant des bénéfices et des inconvénients. Les nouvelles cartes de crédit permettaient à leurs utilisateurs de payer de manière flexible dans le monde entier sans se préoccuper de la devise ou de la disponibilité de liquide. La sécurité était basée sur les détails imprimés concernant l'émetteur, et les détails en relief concernant le porteur de la carte, ainsi que sa signature. L'entité acceptant la carte était responsable de vérifier l'authenticité de la carte et d'identifier le propriétaire grâce à sa signature.

Ce système de sécurité était inadéquat et d'autres développements sécuritaires devinrent nécessaires. Une bande magnétique fut ajoutée sur le revers de la carte pour permettre aux détails de devenir lisibles en machine. Cela pour comporter un PIN (Personal Identification Number - numéro personnel d'identification), pour identifier le propriétaire. Les cartes de crédit pouvaient alors servir à des transactions sur papier avec la signature et les détails en reliefs ou à des transactions électroniques en utilisant le code PIN et les informations stockées sur la bande magnétique. Aujourd'hui encore les cartes avec une bande magnétique et des reliefs sont toujours les plus utilisées pour les transactions financières [100]. Cependant les informations sur une bande magnétique peuvent être lues, altérées ou suprimées en ayant accès à un équipement approprié. Toutes les informations confidentielles telles que le PIN durent être enlevées des cartes pour être référencées depuis un endroit sûr via une transaction en ligne. La sécurité s'en trouvait améliorée, de même que la complexité et cela demandait des moyens de réaliser des transactions électroniques d'avoir accès à un moyen de communication en ligne avec une entité centrale. Il devenait nécessaire de développer une solution hors ligne.

Les années 1970 ont apporté de nombreux changements dans le domaine de l'ingénierie des semi-conducteurs et des puces de quelques millimètres. La carte à puce est née lors du dépoî de quelques brevets clés qui décrivent l'incorporation d'une puce dans une carte en plastique :

- Le brevet déposé par les inventeurs allemands Jürgen Dethloff et Helmut Grötnipp en 1968,
- Le brevet déposé par le japonais Kunitaka Arimura en 1970 pour Arimura Technology Institute,
- Ceux du français Roland Moreno : 47 brevets dans 11 pays entre 1974 et 1979.

Les premiers produits impliquant la technologie des cartes à puce ont été commercialisés vers

la fin des années 1970 quand CII-Honeywell-Bull (maintenant le Groupe Bull) a introduit des cartes à microprocesseurs. Les premiers essais eurent lieu en France et en Allemagne dans le début des années 80 où les cartes de téléphone prépayées et les cartes de crédit sécurisées furent testées. Les qualités de résistance aux attaques furent démontrées et les développements continuèrent. Les cartes de téléphone se répandirent très vite avec quelques millions en circulation en France en 1986, 60 millions en 1990, et plusieurs centaines de millions dans le monde en 1997 [100]. Une autre application des cartes à puce survint en 1988 quand la poste allemande choisit d'employer des cartes à puce comme outils d'authentification dans un réseau de téléphones mobiles analogique. Cela déboucha à l'incorporation de cartes à puce dans la technologie émergeante du réseau GSM mobile (Global System for Mobile Communications - Système global pour les communications mobiles) qui compte maintenant plus de 600 millions d'utilisateurs dans plus de 170 pays.

Un autre domaine d'application expérimentait dans cette technologie mais à un rythme plus lent : l'industrie financière était tout aussi attentive aux progrès en cryptographie qu'en cartes à puce. Les mathématiques derrière la cryptographie passèrent, durant cette période, du domaine réservé du secret des gouvernements vers les universités et l'industrie [73]. Les cartes à puce devinrent un support idéal pour exécuter des algorithmes liés à la cryptographie et pour stocker des clés. En outre, la nature portable des cartes à puce avait pour conséquence que ces outils cryptographiques pouvaient être utilisés partout par le porteur de la carte. Les banques françaises firent les premières avancées en 1984, et dès 1994, toutes les banques utilisaient le code PIN pour authentifier les transactions. L'Autriche fut le premier pays à proposer des cartes de paiement multi-fonction en 1996 [100]. En 1994 est publié EMV (Europay, MasterCard, Visa), ce qui servira à garantir la compatibilité des trois cartes de crédit les plus répandues [36] [37]. Les avancées en cryptographie, en conception matérielle, et logicielle ont permis aux cartes à puce de poursuivre leur diffusion.

1.2 Caractéristiques

Cette section introduit les caractéristiques de la technologie des cartes à puce : les standards pertinents, l'apparence physique, les détails du microprocesseur, le protocole de communication et le cycle de vie.

1.2.1 Les standards

Il y a un certain nombre d'entités qui publient des normes et des spécifications pour les cartes à puce. L'ISO (International Organization for Standardization - organisation internationale pour la standardisation) est la plus importante d'entre elles pour définir les caractéristiques de la technologie. De nombreux autres standards ont émergé de l'utilisation des cartes à puce pour garantir leur compatibilité vis à vis de certains domaines industriels et de certaines applications.

L'ISO et l'IEC (International Electronic Commission - commission internationale électronique) définissent les propriétés de base des cartes à puce. Les groupes de travail (Working Groups - WGs) travaillant sur les standards internationaux dédiés aux cartes à puce sont divisés en deux sections : financière (ISO TC68/SC6) et les applications générales (ISO/IEC JTC1/SC17). Une famille majeure des standards est l'ISO/IEC 7816 [53] [52] [51] qui définit les caractéristiques des cartes à puce avec contacts, ce qui inclut les caractéristiques physiques, l'emplacement des contacts, les protocoles de transmission.

L'ETSI (European Telecommunications Standards Institute - institut européen des standards de communication) définit un ensemble de standards pour l'utilisation des cartes à puce dans le domaine des télécommunications. Le GSM à ce titre définit des standards concernant les

télécommunications en Europe, mais ces standards sont couramment utilisés dans le monde entier. Le successeur de GSM est l'UMTS (Universal Mobile Telecommunication System - système universel de télécommunication mobile). Il est spécifié par le 3GPP (3rd Generation Partnership Project - projet de partenariat pour la 3ème génération) et est publié par l'ETSI. L'UMTS nomme les cartes à puce du domaine des UICC (Universal Integrated Circuit Cards - cartes à circuits intégrés universelles) ou des USIM (Universal Subscriber Indentity Module - module universel d'identité de l'abonné).

Europay, MasterCard et Visa (EMV) ont publié les spécifications EMV pour garantir la compatibilité des cartes du domaine financier en se basant sur ISO 7816. Ce fut un grand pas en avant qui permit la prolifération des cartes de crédit à travers le monde. GlobalPlatform, qui commença sous le nom de Visa OpenPlatform, définit les spécifications à la fois des cartes et des terminaux pour mettre en place une plateforme commune pour le développement de cartes à puce multiapplications.

1.2.2 Propriétés physiques

La taille des cartes à puce a été définie avant l'idée d'embarquer un microprocesseur. La taille standard est définie par l'ISO/IEC 7810 [50]. Elle est appelée ID-1, et elle peut être trouvée dans la plupart des portes-feuilles. Le standard a été amélioré pour définir une carte à puce comme ayant une épaisseur de 0.76 mm et intégrant un microprocesseur avec des contacts à des endroits spécifiés.

La taille ID-1 d'une carte à puce est adaptée pour plusieurs applications, mais d'autres formats sont nécessaires pour d'autres utilisations. Par exemple, l'ETSI a défini un format appelé ID-000 et qui est utilisé dans les téléphones portables GSM. La particularité de cette utilisation est que la carte à puce n'est pas souvent déplacée ou remplacée. Le format correspondant est plus difficile à manipuler que le format ID-1, mais, étant plus petit, il laisse plus de liberté pour les concepteurs de téléphones mobiles. Avec le développement des téléphones portables, ID-000 est devenu trop restrictif, pour la conception de téléphones. L'ETSI a ainsi défini un format plus petit appelé 3FF (3rd Form Factor) [2], qu'on appelle mini-UICC. Ce dernier format n'est pas très répandu, mais les prochaines générations de cartes SIM se conformeront à ce standard. De plus, un format appelé ID-00 a été développé, et est d'une taille comprise entre celle d'ID-1 et celle d'ID-000. Ce format offre d'autres bénéfices en termes de manipulation mais il est relativement nouveau et très peu répandu (cf 1.1).

1.2.3 Microprocesseur

Le microprocesseur dans une carte à puce standard est fait de plusieurs éléments :

- *CPU*
  Le CPU (Central Processing Unit) est typiquement 8-bits, 16-bits et parfois 32-bits. Les microcontrôleurs utilisent des jeux d'instructions 8-bits, comme Motorola 6805 ou Intel 8051 [100], et des jeux d'instructions 32 bit RISC (Reduced Instruction Set) sont parfois utilisés. La fréquence d'horloge varie entre 4MHz et 40MHz, et les opérations normales ont lieu à 5MHz.
- *ROM*
  La mémoire ROM (Read Only Memory) contient des données qui resteront inchangées tout au long du cycle de vie de la carte. On peut y trouver le système d'exploitation et les algorithmes utilisés par la carte. La majorité des informations est placée sur la carte à puce durant la fabrication de celle ci durant un processus appelé *masking*. Une carte typique

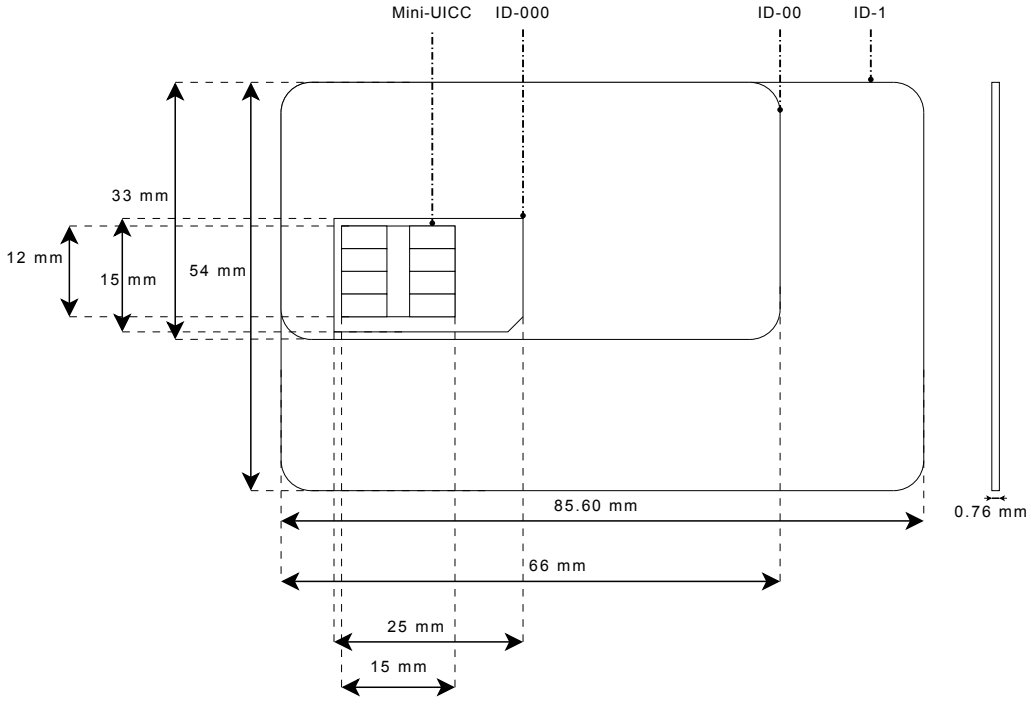


FIG. 1.1 – Les formats de cartes à puce

contient quelques kilo-octets de ROM.

– *RAM*
  La mémoire RAM (Random Access Memory) n'est pas persistante et quand l'alimentation électrique est retirée au microprocesseur, les informations stockées sont perdues. Les données dans la RAM peuvent être manipulées un nombre infini de fois. Les données en question sont souvent pertinentes pour les opérations en cours d'exécution sur la carte. Une carte typique contient de 128 octets à 2 ko de RAM.
– *EEPROM*
  L'EEPROM (Electricaly Erasable Programmable Read Only Memory) est une autre forme de mémoire. Cette mémoire peut être réécrite et son contenu persiste sans alimentation électrique du microprocesseur. Ce type de mémoire est utile pour les informations qui doivent être persistantes, mais qui doivent être uniques au porteur de la carte. Les différents types d' EEPROM ont chacun leur propre temps d'accès, et leur propre durée de vie. L' EEPROM d'une carte typique peut subire 500.000 opérations d'écriture ou d'effacement et garder les informations stockées pendant 10 ans [100]. Bien que lire dans l' EEPROM prend en général autant de temps que dans la RAM, l'écriture est 1000 fois plus lente. La mémoire EEPROM flash est très similaire à de l' EEPROM classique, mais le temps d'écriture est plus rapide qu'avec une mémoire EEPROM standard (10 μs au lieu de 2-10 ms) [100]. Une carte typique contient quelques kilo-octets d' EEPROM.

Ces quatre éléments sont communs à la plupart des cartes à puces mais on peut parfois rencontrer des éléments additionnels. On trouve ainsi des coprocesseurs cryptographiques et des mécanismes de détection d'intrusion [100]. Les microprocesseurs de cartes à puce sont sophistiqués mais fonctionnent dans des environnements contraints. Ainsi, ils n'ont accès qu'à une quantité limitée de mémoire, qu'elle soit RAM, ROM ou EEPROM, et à une puissance de calcul restreinte. Ces points s'améliorent avec les développements matériels. Mais la majorité des cartes à puce ne

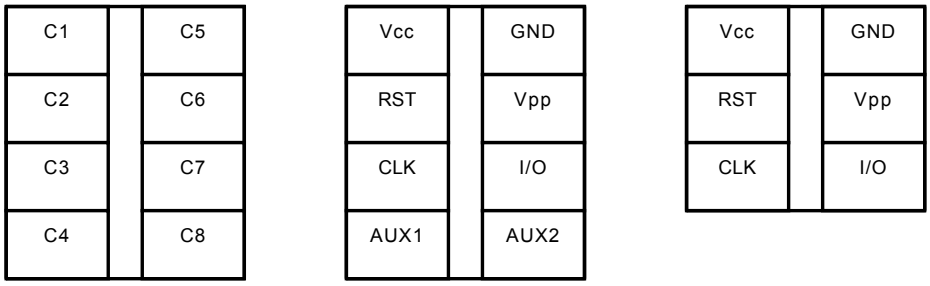


FIG. 1.2 – Points de contact

Contact	Nom	Fonction
C1	Vcc	Tension d'alimentation
C2	RST	Signal Reset
C3	CLK	Signal d'horloge
C4	AUX1	Réservé pour usage futur
C5	GND	Connexion à la masse
C6	Vpp	Tension de programmation de la carte (obsolète)
C7	I/O	Entrées/Sorties
C8	AUX2	Réservé pour usage futur

TAB. 1.1 – ISO 7816-2 Fonctions des points de contact

comptent pas leur mémoire en méga-octets ou en giga-octets.

1.2.4 Communication

Une carte à puce se conformant à ISO 7816-2 doit avoir des contacts sur sa surface qui permettent la communication avec le microprocesseur. Il y a huit points de contact définis dans l'ISO/IEC 7816-2 (cf la figure 1.2 et le tableau 1.1, d'après [100]).

- *Vcc*
  Le microprocesseur est alimenté en électricité par ce point de contact.
- *RST*
  Ce point de contact est utilisé par le microprocesseur pour initier une séquence de *reset*.
- *CLK*
  Le terminal peut fournir au microprocesseur un signal d'horloge avec ce point de contact. Le signal d'horloge peut contrôler la vitesse des opérations et servir de référence pour la communication de données entre une application hôte et le microprocesseur.
- *GND*
  Ce point de contact fournit la masse électrique au microprocesseur.
- *Vpp*
  La tension de programmation était autrefois utilisée pour fournir l'électricité nécessaire à la programmation de l'EEPROM. Aujourd'hui, ce point de contact est obsolète.
- *I/O*
  Une application hôte et le microprocesseur peuvent communiquer en mode *half-duplex* grâce à ce point de contact.
- *AUX1* et *AUX2*
  Ces points de contact étaient au départ réservés pour un usage ultérieur. Ils sont maintenant utilisés à des fins d'encapsulage dans une couche USB pour certaines cartes [51].

Couche Application (APDU))
Couche Transport (TPDU)
Couche physique

FIG. 1.3 – Modèle OSI adapté aux cartes à puce

Les communications avec une carte à puce prennent une forme *half-duplex* (les échanges entre la carte et le terminal se font en un seul sens à un moment donné). Cette caractéristique entraîne un besoin de définir un rapport maître/esclave dans la communication, et pourra engendrer des problèmes de collisions de données.

Lorsqu'une carte à puce est insérée dans un lecteur (Card Acceptance Device - CAD), celui ci lui fournit l'électricité nécessaire à son fonctionnement, et il initie la communication avec la carte. Les communications avec la carte utilisent un modèle de pile de protocoles aligné sur le modèle OSI, comme le montre la figure 1.3. Un registre de données au niveau transmissions est appelé un TPDU (Transport Protocol Data Unit [53]), tandis qu'un registre de données au niveau application est appelé un APDU (Application Protocol Data Unit).

L'ATR (Answer To Reset) est un tableau d'octets qui est envoyé par la carte au terminal après avoir détecté un *reset* de la part du CAD. Entre autres, l'ATR fournit au CAD des informations sur les protocoles de transmission et les débits supportés par la carte à puce. La durée d'un moment élémentaire - appelé ETU pour Elementary Time Unit - sur le contact CLK est défini par l'ATR. L'ATR définit, dans un octet TA1, un nombre F (facteur de conversion pour l'horloge) et un nombre D (facteur de conversion pour le débit) dont le rapport $F/(D \times f)$ définit un nombre de cycles par ETU pour une fréquence f. Cette dernière fréquence est celle du signal d'horloge du lecteur et elle est comprise entre 1MHz et 5MHz. La vitesse de transmission va en général de 9.600 bits/s à 111.600 bits/s.

Chaque octet prend 10 ETU (1 par bit plus 1 bit d'initialisation et 1 bit de parité) pour transiter entre l'émetteur et le recepteur à plus ou moins 0,2 ETU. Un "temps de garde supplémentaire" (EGT - Extra Gard Time) entre chaque octet permet au transmetteur et au receveur de se préparer pour le prochain octet. Celui-ci est défini par un octet TC1 dans l'ATR. Ce temps permet par exemple au receveur de vérifier le bit de parité et de transmettre un signal d'erreur en cas de problème. La fréquence à laquelle le receveur vérifie le contact I/O doit être de moins de 0,2 ETU.

La figure 1.4 présente des débits de quelques cartes à puce pour un CAD donné fonctionnant à 3.57MHz.

Le standard ISO 7816 présente des protocoles de transmissions et des commandes pour échanger des données entre la carte et le terminal (cf figures **??**, 1.3). Les protocoles de transmissions définissent les processus de communication entre le terminal et la carte à puce, et les mécanismes à utiliser pour manipuler correctement les erreurs de transmission détectées. Les protocoles de transmission T=0 et T=1 sont ceux couramment employés dans le cadre d'une communication avec une carte à puce. Ils sont utilisés presque sans exception avec toutes les cartes à puces à contact [99]. Il y a quelques types de cartes à puce qui supportent un protocole USB, notamment pour des applications qui nécessitent une grande mémoire. Dans le cas des cartes à puce sans contact, les protocoles les plus employés sont ISO/IEC 14443 Type A ou B.

Le protocole de transmission T=0 est le plus ancien et le plus répandu. C'est un protocole de transmission orienté octet. Les commandes APDU de type 4 sont impossibles en T=0. Le terminal doit à la place faire envoyer une commande de type 3 puis une commande de type 2 (une commande GET RESPONSE), pour récupérer les données que la carte à puce renvoie au

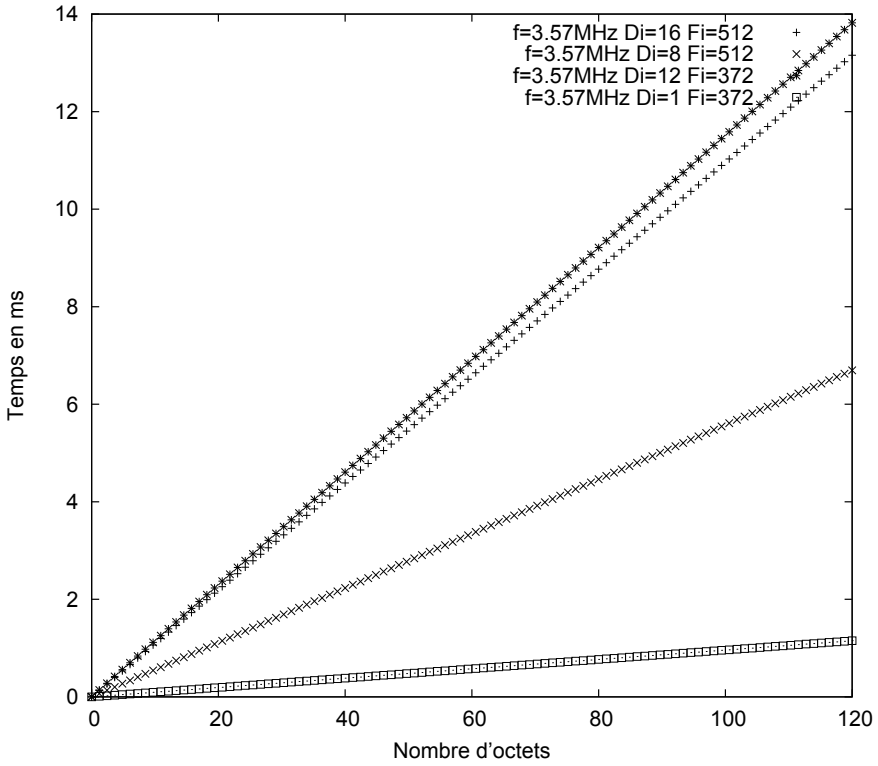


FIG. 1.4 – Débits pour quelques cartes à puce avec un CAD fonctionnant à 3.57MHz

Commande APDU	**Entête**				**Corps Optionnel**		
Type 1	CLA	INS	P1	P2			
Type 2	CLA	INS	P1	P2	LE		
Type 3	CLA	INS	P1	P2	LC	DATA	
Type 4	CLA	INS	P1	P2	LC	DATA	LE

TAB. 1.2 – Commandes APDU ISO 7816-4

terminal. L'utilisation du protocole T=0 n'en est pas affectée pour autant. C'est le protocole standard pour la plus répandue des applications de la carte à puce dans le monde : les cartes SIMs et USIMs utilisées dans les systèmes de télécommunication mobile GSM et UMTS.

Le protocole T=1 est orienté blocs, et tous les types d'APDUs sont utilisables directement. T=1 a une structure beaucoup plus compliquée que T=0, mais il est aussi plus robuste, car il met en place des processus de détection d'erreur et de renvois de données défectueuses. T=1 est souvent utilisé dans le monde sur les cartes de paiement et les cartes d'identité. Ces avantages par rapport à T=0 ne sont pas suffisants pour rendre T=0 obsolète.

L'ISO 7816-4 spécifie les commandes à envoyer à une carte à puce et les réponses qui y sont associées (voir les tableaux 1.2 et 1.3). Une commande APDU est constituée d'un entête et d'un corps. L'en-tête est obligatoire, mais le corps est optionnel. Une réponse APDU est constituée d'un corps et de deux octets appelés le mot de statut. Seul le mot de statut est obligatoire dans la réponse.

Une commande APDU est faite d'un entête de quatre octets : l'octet de classe CLA, l'octet d'instruction INS, et deux octets de paramètres P1, P2. La classe sert à indiquer le standard dans

Corps optionnel	Mot de statut obligatoire	
DATA	SW1	SW2

TAB. 1.3 – Réponses APDU ISO 7816-4

lequel la commande est spécifiée. L'octet d'instruction définit la commande qui doit s'exécuter et les deux paramètres P1 et P2 fournissent des informations additionnelles sur la commande. La section additionnelle contient trois éléments suplémentaires. LC précise la taille du champ de données DATA qui est envoyé à la carte. Enfin, LE détermine la taille de la réponse attendue. LC et LE ont tous les deux une longueur de 1 octet.

La réponse APDU contient un champ optionnel DATA, dont la taille doit correspondre au LE de la commande APDU, et un mot de statut obligatoire sur deux octets (SW1 et SW2). Ces derniers servent à donner le code de retour de la commande. Ils peuvent ainsi prendre plusieurs valeurs dont le sens est fixé par la norme, tels "Succès", "Échec", "Instruction non comprise" [52] ... Un développeur d'application peut définir des mots de statut non standards. La taille maximum d'un champ de données DATA est de 254 octets. Il peut varier d'une carte à l'autre, et un mécanisme permet de recevoir ou d'envoyer des données plus grandes en plusieurs réponses APDU.

1.2.5 Horloge

Un processeur de carte à puce n'a en général pas de générateur d'horloge interne. Une horloge externe doit nécessairement être fournie à travers le CAD. Cette horloge sert de référence pour les débits de données.

Le signal de l'horloge appliqué au contact CLK n'est pas nécessairement le même que l'horloge interne du processeur. Certains microcontrôleurs ont un diviseur ou un multiplicateur d'horloge qui peut s'insérer entre l'horloge externe et l'horloge interne. Cela permet aux oscillateurs présents dans les terminaux d'être utilisés comme source du signal d'horloge pour la carte à puce. La plupart des microcontrôleurs permettent au signal d'horloge d'être désactivé lorsque le microprocesseur est en sommeil. L'économie d'énergie qui en découle est de quelques micro ampères, ce qui peut être utile dans certaines applications [100].

Il est à noter que la consommation de courant électrique en milli ampère croit linéairement avec la fréquence d'horloge quand le processeur opère en mode normal [100].

1.2.6 NFC

La technologie NFC [85] (Near Field Communication - communication en champ proche) permet à des appareils d'interagir à courte portée (moins de 10 centimètres) par un signal haute fréquence. Cette technologie est une extension du standard ISO 14443 [54] (cartes sans contacts, RFID). La fréquence de fonctionnement est de 13,45 MHz, avec un débit de 106, 212, 424 ou 848 kbits/s et une communication *half-duplex* entre les appareils compatibles.

Les cartes à puce NFC sont donc des cartes à puce accessibles à courte distance. Une carte à puce de ce type tire son énergie soit d'un couplage capacitatif (soit une batterie) soit d'un couplage inductif (collecté par l'antenne). Le couplage inductif fonctionne sur le principe du transformateur où une bobine dans le lecteur induit un courant dans une autre bobine (c'est à dire l'antenne dans la carte). La puce est capable, en changeant sa résistance, de transmettre un signal qui est capté par le lecteur et interprété comme un signal de données.

Certaines des interactions visées par la technologie NFC sont les interactions à partir d'un téléphone. NFC offre une possibilité d'interagir plus facilement qu'avec du Bluetooth (pas de

configuration) avec une portée plus courte (ce qui amène un peu plus de sécurité), un débit plus faible mais surtout une consommation électrique beaucoup plus faible.

Certaines cartes à puce possèdent à la fois une capacité de communication sans contact et avec contact. Ces cartes sont nommées des cartes **dual interface**.

1.2.7 Cycle de vie

Toutes les cartes à puce vont suivre des cycles de vie similaires malgré la diversité des usages qu'on peut en faire. L'ISO 10202-1 tente de représenter ce cycle de vie commun. Ce standard est centré autour du domaine bancaire mais décrit toutes les cartes à puce de leur conception à leur fin de vie [100].

Le cycle de vie est constitué de cinq phases et chaque transition entre deux phases sont décrites avec précision. Différents acteurs interviennent durant les phases :

1. **Phase 1 - Production**
   Le **fabricant de puce** met en place certaines parties strictement matérielles (on choisit le microprocesseur et la mémoire). Un système d'exploitation rudimentaire et quelques données sont figées dans la mémoire ROM par une opération de **masquage** qui consiste en des dépôts successifs de couches conductrices et de couches isolantes sur une galette de silicium pour obtenir un circuit tridimensionnel.
   Le fabricant de cartes découpe les galettes de silicium, en puces individuelles qui vont être testées. Il met en place les contacts (pour une carte à contact) et le format plastique.
2. **Phase 2 - Préparation de la carte**
   Le fabricant de cartes charge des fonctionnalités dans l'EEPROM, en particulier les éléments communs aux applications qui vont être installées.
3. **Phase 3 - Préparation de l'application**
   À ce stade toutes les cartes sont identiques. Souvent, le fabricant de cartes développe une ou plusieurs applications pour le compte de l'**émetteur de carte** et les installe sur la carte. Une étape de **personnalisation** de la carte vis-à-vis du porteur a lieu.
4. **Phase 4 - Utilisation de la carte**
   C'est la phase la plus connue des **porteurs**. Éventuellement, on pourra réinstaller des applications, les personnaliser et repasser à une phase d'utilisation. Dans ce cas, on peut avoir plusieurs transitions successives entre les phases 3 et 4. Les applications peuvent aussi être désactivées et les cartes peuvent bloquer d'autres installations.
5. **Phase 5 - Fin de vie**
   C'est une phase finale de l'utilisation. La carte et les applications sont désactivées. Cela peut arriver par accident, destruction, vol et plus rarement retour à l'émetteur pour destruction.

1.3 Plates-formes

1.3.1 Les plates-formes Java Card

Schlumberger développa la première carte à puce capable d'exécuter des programmes écrits en Java en 1996. Cet effort n'était cependant pas le premier effort vers la réalisation d'une carte à puce à architecture ouverte (OTA par Europay en est un exemple).

Autour de Sun, se créa en 1997 une conférence, appelée aujourd'hui Java Card Forum, réunissant les acteurs du monde de la carte à puce. Le Java Card Forum (JCF) se fixa comme but

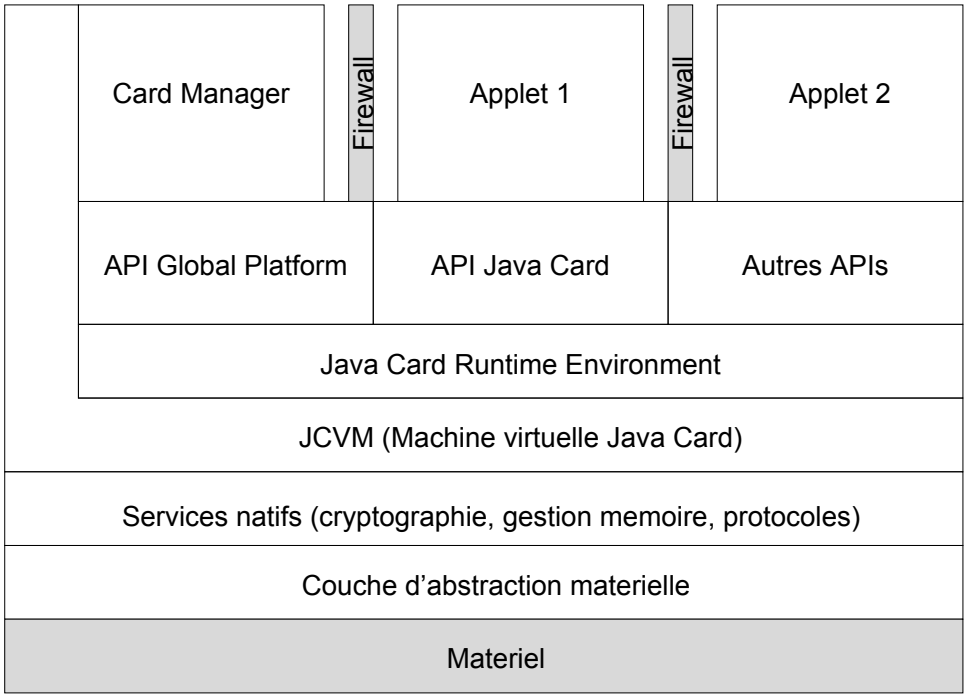


FIG. 1.5 – Architecture de Java Card

de définir un standard international pour faire du Java dans les cartes à puces. Ainsi le JCF définit :

- le sous ensemble du langage Java qui est utilisé dans les cartes à puce,
- l'interpréteur de *bytecode* (code objet) Java, c'est à dire la machine virtuelle Java ou JVM,
- des APIs qui soient générales à tout type d'application ou spécifiques (aux transactions financières par exemple).

Ces APIs forment l'interface entre le système d'exploitation de la carte à puce et Java.

Aujourd'hui, la norme consiste en :

- la spécification de la machine virtuelle Java Card (JCVM),
- la spécification de l'environnement d'exécution Java Card (JCRE),
- l'API Java Card.

Une carte à puce Java a une machine virtuelle Java qui est activée pendant la fabrication de la carte et désactivée à la fin du cycle de vie de la carte. Quand une carte à puce n'est plus alimentée en électricité, la machine virtuelle reste active mais suspend ses activités.

L'applet dans la carte peut être sélectionnée par un AID (Application IDentifier) en utilisant une commande SELECT. Après la sélection d'une applet, celle-ci recevra tout les APDUs ultérieurs. Le code d'une applet peut évaluer les commandes APDUs, effectuer des accès en mémoire et générer une réponse. Cette approche permet de faire coexister sur une même carte plusieurs applets qui répondront à une commande donnée de manière mutuellement indépendante.

Le développement d'une application Java Card est illustré par la figure 1.6. Pour développer un programme Java pour carte à puce, on doit tout d'abord écrire les sources du programme à l'aide d'un éditeur. On peut ensuite compiler le code source pour générer du *bytecode* qui est indépendant de la machine hôte. Jusqu'ici, il n'y a pas de différence entre du Java pour PC et du Java pour la carte à puce. Les bytecodes sont ensuite transférés au Java Card Converter (étape

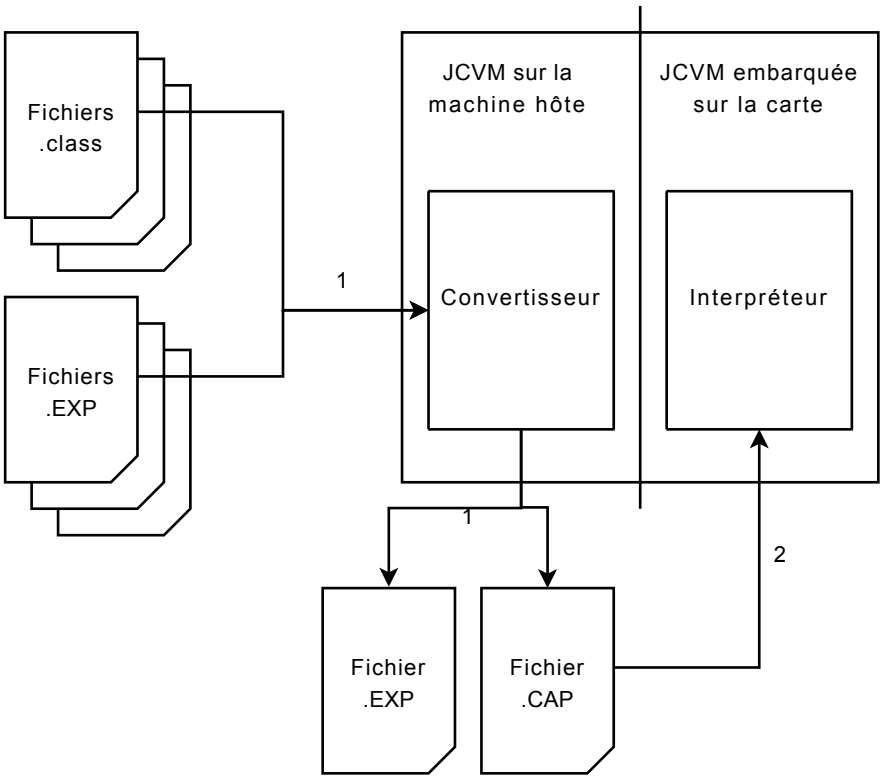


FIG. 1.6 – La machine virtuelle de Java Card

1 sur la figure 1.6), c'est à dire le convertisseur Java Card, qui fait le travail d'une partie de la machine virtuelle Java, mais sur le PC et non sur la carte. À ce titre, il teste le format, la syntaxe et les références de champs dans le programme. À la suite de ces tests, il génère un fichier CAP (Card Application File), qui n'est autre qu'un JAR (archive java compressée) contenant le bytecode de l'application, ainsi qu'une éventuelle signature pour authentifier l'application et sa provenance. On peut alors charger ce fichier CAP sur la carte à puce, en utilisant les mécanismes de *GlobalPlatform* (cf 1.3.3) - étape 2 sur la figure 1.6. La machine virtuelle embarquée dans la carte à puce va pouvoir lire le bytecode ligne par ligne et générer des instructions machines pour le processeur.

Une critique fréquente contre l'utilisation de Java dans une carte à puce est la vitesse d'exécution. Il est difficile de faire des comparaisons pertinentes entre un programme en assembleur et un programme en Java. Par exemple, programmer un algorithme cryptographique en Java n'a aucun sens. Un programme Java typique va par contre utiliser intensivement les méthodes de l'API Java Card qui sont codées en partie par des instructions natives propres au processeur [100].

L'accès au ramasse miette est particulier. Une méthode `JC.System.requestObjectDeletion()` est implémentée dans certaines cartes. Cette méthode fait appel au ramasse miette. En dehors de cet appel, le ramasse miette n'est pas lancé de manière intempestive par le JCRE. Une récupération de la mémoire aura typiquement lieu à l'effacement d'une application.

Le succès de Java Card est indéniable, et il est lié aux succès en général de l'industrie de la carte à puce. Java est devenu de facto, le langage standard de la programmation Java Card [100]. Chaque année, 1 milliard de cartes à puce Java sont produites alors que 6 milliards de Java Cards sont en circulation et que 90% des cartes SIMs aux USAs, dans l'UE et en Amérique du

Sud sont des Java Cards [68].

Le développement du standard Java Card a accompagné la popularité des cartes ces dernières années. Étant capables d'exécuter un sous ensemble du populaire et portable langage Java, les plates-formes Java Card rendent plus accessible aux programmeurs le monde de la carte à puce. Ceci permet aussi de réduire le temps pour commercialiser les applications pour cartes à puce [12]

1.3.2 Java Card 3.0

En mars 2008, le Java Card Forum a publié la version 3.0 de la norme Java Card [59]. Cette nouvelle version est un pas en avant majeur dans la conception d'applications pour cartes à puce en proposant une édition "Classic" qui améliore la version précédente de Java Card (et qui est compatible avec celle-ci) et une édition "Connected" qui propose de concevoir des applications pour cartes à puce avec une approche orientée "serveur web". D'un point de vue de la facilité de programmation, cette dernière édition permet de s'abstraire un peu plus de la plateforme pour s'approcher d'un Java plus traditionnel (les types tels que `String` `int` sont supportés) qui pourrait s'apparenter plus à du J2ME qu'à du Java Card 2.X. Une notion de multi-threading est également introduite, et même si les mécanismes de manipulation d'APDUs pour gérer les entrées/sorties sont toujours présents, le développeur peut s'en passer pour utiliser des méthodes utilisées dans les réseaux (HTML, SOAP). Un autre apport de Java Card 3.0 est la plus grande facilité qu'il y a à créer des objets volatiles (ce qui implique un mécanisme de ramasse miettes élaboré).

La conséquence de ces améliorations est que cette plateforme est beaucoup plus ouverte aux développeurs traditionnels de Java. Le développement d'applications Java Card est moins un travail de spécialiste avec cette version. Une autre conséquence est que la complexité accrue des mécanismes systèmes dans l'environnement d'exécution et dans la machine virtuelle entraîne la nécessité d'utiliser des cartes à puce plus performantes et plus coûteuses.

Il n'y a pas à l'heure actuelle de cartes à puce mises en service utilisant Java Card 3.0. La version actuelle est Java Card 3.0.1 et date du milieu de l'année 2009. Une autre version 3.0.2 est attendue pour la fin de l'année 2009.

1.3.3 GlobalPlatform

La fin des années 90, l'industrie de la carte à puce a commencé a proposer des cartes multi-application (la première version de Java Card date de 1997). Cette technologie impose de ce fait de devoir gérer les applications sur une carte, en tenant compte des droits de chacune. Cependant, Java Card ne proposait pas de norme pour la gestion d'application embarquée. Avec ses activités dans le monde de la carte à puce, Visa International était confronté relativement tôt au fait de devoir faire coexister dans une même carte plusieurs applications provenant de plusieurs sources. Ceci entraîna l'apparition de la spécification Visa Open Platform (VOP). Open Platform définit une interface dans le système d'exploitation d'une carte à puce pour gérer les applications. Depuis 1999, un comité décide des évolutions de cette spécification. Elle est aujourd'hui appelée GlobalPlatform. Cette spécification est indépendante de tout système d'exploitation, ce qui permet de supporter tous types de systèmes d'exploitation pour carte à puce. Ainsi Multos et Java Card supportent cette spécification. En pratique, elle est le standard pour charger et gérer les applications Java dans les plates-formes Java Card. Par exemple, l'ETSI GSM 03.19 considère GlobalPlatform comme l'interface standard pour charger des applications.

L'architecture de GlobalPlatform est faite d'un certain nombre de composants qui fournissent

des interfaces neutres du point de vue du matériel. Le Card Manager est l'élément clé qui gère une carte GlobalPlatform.

1.3.4 D'autres plates-formes

Il existe un certain nombre d'autres plates-formes que nous ne détaillerons pas.

Multos

La plateforme Multos [75] a été développée par le consortium MAOSCO qui compte comme membres American Express, Dai Nippon Printing, Mondex International Ltd, Siemens, Fujitsu, Hitachi, Motorola, MasterCard International, Keycorp. C'est la première carte à puce ouverte multi-applicative. Les applications sont développées de manière indépendante du matériel. Ces cartes sont compatibles ISO 7816 et EMV. Un langage, MEL (Multos Execution Language), basé sur du Pascal est utilisé avec l'aide d'une machine virtuelle. Les développements ont lieu en Java, en C, en Basic ou en Modula-2 et les compilateurs peuvent générer du MEL. Multos est un concurrent à Java Card.

Windows for Smart Card (WSC)

Développée en 1998 par Microsoft, cette plateforme multi-applicative compatible avec l'ISO 7816 n'est pratiquement plus utilisée aujourd'hui [105]. WSC permettait de charger dynamiquement des applications, contenait un système de fichier FAT, comportait une machine virtuelle et proposait une API faite à partir d'une version réduite de l'API Windows.

Smart Card .NET

Cette plateforme propose un support pour .NET sur carte à puce. L'initiative vient de Hive-Minded. Beaucoup de langages sont supportés : C#, C++, Visual Basic, J# et Java Script. D'autres langages sont supportés dans certaines implémentations : Perl, Python, Eiffel, Forth et Cobol. L'architecture permet les appels à distance de .NET, ce qui permettrait une plus grande connectivité entre la carte et le terminal.

Basic Card

La société allemande ZeitControl propose des plates-formes Basic Card depuis 1996 et depuis 2004, cette plateforme est multi-applicative. Les cartes à puce Basic Card sont des cartes avec ou sans contact et implémentent des algorithmes tels que DES, 3DES, AES, RSA, SHA-1 et des algorithmes à courbes elliptiques. Les applications sont relativement peu complexes et sont très performantes [108]. Ces cartes à puce sont par ailleurs peu coûteuses, ce qui explique en partie la résistance de cette plateforme face à des standards soutenus par des entreprises beaucoup plus grandes.

1.4 Les APIs d'entrées/sorties

Une partie indispensable de toute application pour les cartes à puce est la partie cliente sur la machine hôte. Quand la machine hôte est un ordinateur, les programmeurs ont besoin d'un moyen d'envoyer des APDUs à la partie serveur sur la carte à puce de l'application. Différentes APIs implémentées de différentes manières permettent aux programmeurs de définir des APDUs et de les envoyer à travers un service et à l'aide du pilote du CAD vers la carte à puce.

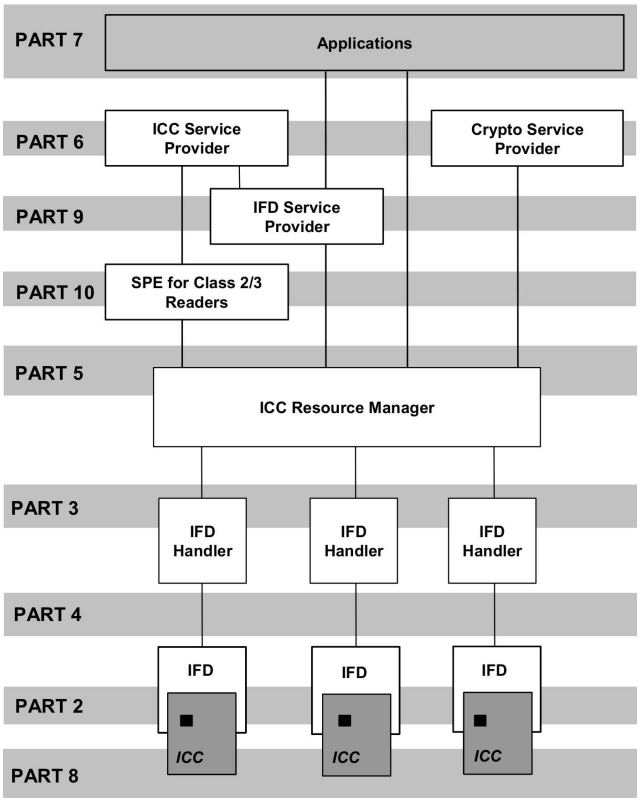


FIG. 1.7 – L'architecture de PC/SC 2.0.1

1.4.1 PC/SC

La spécification PC/SC [91] (Personal Computer/Smart Card) est dédiée à l'intégration des cartes à puce dans le contexte d'un ordinateur. La spécification provient d'un groupe d'industriels, PC/SC Workgroup, comprenant entre autres Gemalto, Infineon, Microsoft et Toshiba. La figure 1.7 présente l'architecture globale de PC/SC. Dans cette architecture, des ICCs (Integrated Chip Card - cartes à puce) sont insérées dans des IFDs (Interface Device - des lecteurs où CADs). Les caractéristiques physiques supportées correspondent aux normes ISO 7816 et ISO 14443. Les IFDs handlers, c'est à dire les drivers des lecteurs doivent être programmés par les constructeurs des lecteurs en se conformant aux APIs existantes (dont celle de Microsoft). L'architecture de PC/SC engendre un besoin système de fournir un "ICC Ressource Manager", autrement dit un service qui va gérer les ressources systèmes relatives aux cartes à puce et qui va contrôler l'accès aux lecteurs.

Deux implémentations majeures de ce standard existent. PC/SC est implémenté dans les systèmes Windows 200x/XP de Microsoft et est disponible sous Windows NT/9x.

Dans ce modèle, chaque constructeur de lecteurs, pour chacun de ses lecteurs, devait donc fournir un pilote (c'est à dire IFD Handler) respectant l'API Microsoft pour le support des lecteurs (par exemple une API semblable à celle définie dans la partie 3 des spécifications PC/SC). De plus, ce pilote devait passer un ensemble de tests du Windows Hardware Quality Labs afin d'être homologué.

En effet, au niveau de la partie 4 des spécifications, différents lecteurs pouvant utiliser un protocole spécifique même s'ils utilisent le même média (par exemple l'USB), il fallait un pilote par lecteur ou tout au moins par famille de lecteur utilisant le même protocole pour le même

média. Toutefois, afin de résoudre ce lourd problème d'installation de pilotes, une initiative visant à standardiser le protocole sur le média USB a été lancée au sein de l'USB Implementers Forum et a développé la spécification CCID 1.0 (c'est à dire USB Chip/Smart Card Interface Device). Ainsi, Microsoft a implanté cette spécification dans un pilote qu'il distribue avec ses systèmes d'exploitation permettant ainsi à n'importe quel lecteur compatible CCID de fonctionner sans requérir l'installation de pilote supplémentaire lors de son branchement.

Une implémentation libre de PC/SC, `pcsc-lite` est disponible sous Linux et Mac OS X [103]. Elle a été développée par le groupe MUSCLE (Movement for the Use of Smart Card in a Linux Environment), qui compte comme développeurs, David Corcoran, Ludovic Rousseau et Damien Sauveron.

Le cœur de ces différentes implémentations est une API en C permettant d'interagir avec un service (un démon PC/SC pour `pcsc-lite`) : c'est le gestionnaire de ressource décrit plus haut.

La plus part des APIs d'entrées/sorties suivantes ne sont que des *wrappers* (encapsulateurs) de PC/SC.

1.4.2 OCF

L'OpenCard Framework (OCF) est un intergiciel pour carte à puce implémenté en Java. Il est à noter qu'OCF n'est pas un wrapper PC/SC, mais a une architecture propre. Il permet d'accéder aux cartes à puce en utilisant des APDUs définis dans l'ISO7816-4. Á l'origine ce projet a été mené par l'OpenCard Consortium qui réunissait Gemplus et IBM entre autres. Le projet a été arrêté dans sa version 1.2. À la suite de la publication de cette version, l'OpenCard Consortium s'est dissous, plongeant le projet dans un stade dormant, et aboutissant à la fermeture du site web en 2007. Effectivement le projet était déjà non maintenu depuis 2000 [105]. De nombreux programmes utilisent encore aujourd'hui OCF, mais il peut être considéré comme obsolète aujourd'hui.

1.4.3 JPC/SC

JPC/SC est une API en Java développée par Marcus Oestreicher de IBM BlueZ Secure Systems et encapsulant l'API PC/SC. Un programme utilisant cette librairie va communiquer avec les ressources PC/SC disponibles sur la machine (par exemple avec le démon PC/SC sous Linux).

1.4.4 JSR268

La JSR 268 [66] est une API d'entrées/sorties pour communiquer avec une carte à puce en Java. La version finale de cette API date de Décembre 2006. Une librairie implémentant cette API est distribuée avec le JDK 1.6 de Sun. Elle est de facto, la norme pour communiquer avec une carte à puce en Java en utilisant des APDUs.

1.5 Applications

De nombreux domaines appliquent les technologies issues des cartes à puce en général et de Java Card en particulier. Parmi ceux ci, certains se démarquent :

- **Les télécommunications**.
  Ce domaine est très largement le plus gros demandeur de technologie Java Card, en particulier pour les opérateurs GSM/UMTS. En Europe, près de 100% des cartes SIMs sont des Java Cards [68]. Depuis quelques temps, certaines intègrent des applications de télévision

payante. Les applications SIMs sont particulières puisqu'elles sont compatibles avec SIM Application Toolkit (STK). STK est un standard de l'ETSI défini en 1994 qui concerne les services à valeur ajoutée et les transactions de e-commerce sur les téléphone GSM (standard GDM 11.14 [39]). Cela permet de donner un rôle proactif à la carte SIM. Celle-ci peut piloter l'interface du téléphone mobile, contrôler l'accès au réseau, et permettre à l'utilisateur d'établir un échange interactif avec une application réseau. Dans les téléphones 3G, les cartes USIM (Universal Subscriber Identity Module) utilisent une extension de ce standard : l'USIM Application Toolkit (USAT) [1]. Un autre aspect de ce domaine d'application le distingue des autres : il est très courant dans les télécommunications de charger des applications après la mise en service d'une carte. C'est ce que l'on appelle la gestion OTA (Over The Air).

- **Le domaine bancaire**.
  Le secteur bancaire a été actif dès la définition de Java Card et de GlobalPlatform qui avait commencé sa vie comme Visa Open Platform. L'adoption de Java Card a été long dans le secteur. En effet, le secteur bancaire est attiré par le faible coût des cartes, mais la flexibilité de Java Card n'a pas d'intérêt évident pour une carte bancaire. Cependant, la baisse des prix a permis aux acteurs de ce domaine de proposer des implémentation en Java. Les applications bancaires sont souvent bâties sur une notion de transaction, ce qui représente la très grande majeure partie des cas. Certaine cartes bancaires proposent des services de porte-monnaie électroniques. Les calculs cryptographiques sont souvent assez importants, pour l'authentification du terminal, et la signature des données transactées. La norme EMV [36] [37] propose des protocoles d'authentification avec des clés RSA allant jusqu'à 1984 bits, et met fin aux cartes à bandes magnétiques.
- **La télévision payante**.
  Le secteur multimédia est assez spécifique. Il est hautement compétitif, il n'y a pas de standard admis par toute l'industrie et les cartes sont souvent attaquées par des techniciens qualifiés et avec des moyens conséquents. Les informations sur les applications sont confidentielles et celles-ci sont souvent développées en code natif. Java Card est considérée comme une couche non contrôlable et donc relativement indésirable. L'arrivée de la télévision mobile, l'industrie a adapté ses standards à ceux du secteur télécom, Java Card inclus. Les niveaux de sécurité dans les applications en Java Card sont assez haut par rapport aux autres secteurs, mais relativement bas par rapport aux applications natives du domaine. Les applications ne sont pas disponibles car très protégées. Cependant, nous savons qu'il y a des applications de DRMs (Digital Rights Management - Gestion des droits numériques) qui utilisent intensément les opérations cryptographiques des cartes.
- **Les transports**.
  Les applications de transport sont des applications pour des cartes sans contacts. Les contraintes de performance sont assez élevées, car les transactions doivent être très rapides. Il y a peu de temps, il était considéré comme étant impossible d'utiliser une Java Card qui satisfasse les performances minimales. Les dernières générations de Java Cards ont des performances nettement meilleures et cette barrière n'existe plus. Certaines plates-formes Java Card ont même des performances similaires à celles des applications natives, en utilisant un matériel différent. Les applications de transport sont utilisées pour gérer des droits et les contraintes de sécurité sont relativement élevées, comparables à celles du secteur bancaire. L'essentiel est d'empêcher un rechargement de la carte sans payer. Un exemple de spécification pour les transports est Calypso qui est utilisé par la SNCF et la RATP à Paris [72].
- **L'identité**.

Dans le secteur de l'identité, les applications sont plutôt simples et doivent surtout gérer une grande quantité de données dont il faut garantir l'authenticité. Les besoins en termes de sécurité varient grandement d'une application à une autre et il est difficile de faire une analyse de risque. La valeur du bien à protéger et les dommages possibles sont difficiles à évaluer, ce qui justifie des extrémités dans les deux sens pour le niveau de sécurité et pour la performance.

1.5.1 Un exemple d'application : MuscleCard

MuscleCard est une application d'identité libre (licence BSD).

Le projet MuscleCard fut initié, comme `pcsc-lite`, par David Corcoran au sein de MUSCLE [104]. Il a pour ambition de fournir à l'utilisateur final des services cryptographiques pour lui permettre de s'authentifier sur sa machine, de signer ses messages, etc.

L'architecture de MuscleCard est présenté dans la figure 1.8.

MuscleCard a initialement été développé comme une solution pour Mac OS X et pour Linux afin de pallier l'absence d'un support complet de PC/SC dans ces environnements. Depuis, il a aussi été porté dans les environnements Windows de Microsoft.

Depuis son commencement, un des buts du projet a été d'être indépendant du fournisseur de cartes pour offrir les services cryptographiques. C'est donc tout naturellement que les développeurs ont pensé à utiliser la technologie Java Card. Ainsi, ils ont développé une applet Java Card connue sous le nom de MCardApplet et destinée à être chargée sur la Java Card de toute personne souhaitant utiliser la plate-forme MuscleCard.

Cette applet permet de stocker des informations de diverses natures : PINs, clés cryptographiques, certificats et tout autre donnée que l'on souhaite garder confidentielle. Bien évidemment, cette applet gère également des notions de droit en lecture et en écriture. Enfin, suivant les capacités cryptographiques des Java Cards sur lesquelles elle est chargée, cette applet est capable d'accéder à un certain nombre de mécanismes cryptographiques comme le chiffrement ou le déchiffrement de données, la signature ou la vérification de signature, le hachage de données.

Pour résumer, grâce à cette applet, le projet MuscleCard touche plusieurs dizaines de modèles de Java Cards, ce qui est loin d'être négligeable.

L'essence de cette application est de centraliser certains aspects sécuritaires d'un utilisateur de système informatique dans une carte à puce. Ces aspects peuvent être un code PIN, une clé ou une paire de clés d'un algorithme cryptographique supporté par la carte à puce, ou une donnée biométrique. La partie de l'application située sur la machine hôte peut interagir avec des programmes et appeler la partie de l'application sur la carte pour gérer des besoins de chiffrement, de mots de passe ... Elle utilise une partie CSP (Crypto Service Provider) qui est utilisée dans Windows comme une API cryptographique qui va éventuellement faire le lien vers les cartes à puce insérées dans les lecteurs de la machine et qui les identifie par leurs ATRs. Un module PKCS#11 permet de signer des messages, et de s'authentifier auprès des sites web avec Mozilla par exemple. Cela constitue un exemple de *Single-Sign on*.

1.6 Conclusion

En quelques décénies, les cartes à puce sont devenues des objets d'usage quotidien presque banaux pour des milliards de personnes. L'évolution matérielle des cartes a poussé les performances vers le haut, ce qui a permis des progrès logiciels. Il y a quelques temps, elles devaient être dédiées à une seule tâche. Aujourd'hui, les plates-formes multi-applications sont courantes

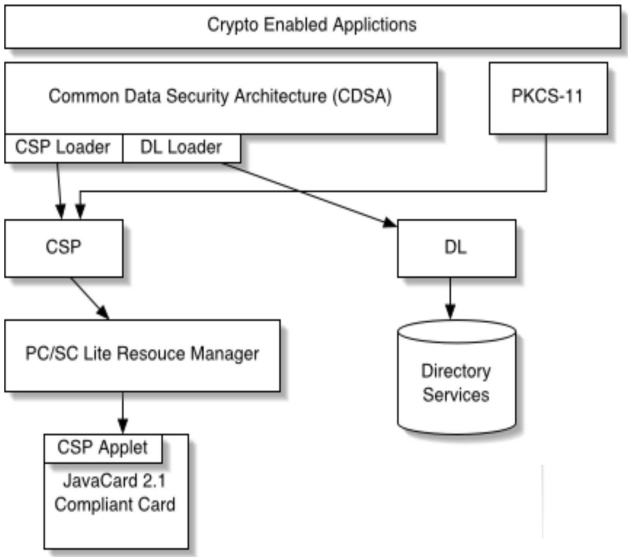


FIG. 1.8 – L'architecture de MuscleCard

avec notamment Java Card, et elles incorporent des mécanismes de sécurité complexes. La spécification Java Card 3.0 amène avec elle de nouvelles innovations et une nouvelle complexité dans la programmation. Dans ce contexte, relativement peu de travaux ont été menés pour étudier ouvertement la performance des cartes à puce.

Les cartes à puce sont en général livrées en très grands volumes, de l'ordre du million d'exemplaires. Il serait donc compliqué et coûteux pour les fabricants de remédier à un éventuel problème de performance ou de sécurité.

Il est donc crucial pour les fabricants de s'interroger sur les performances des cartes à puce produites tout en gardant à l'esprit les applications qui sont prévues sur ces cartes.

2

État de l'art

Sommaire

2.1 Introduction

Dans le cycle de vie d'un système informatique, la maturité est souvent marquée par l'apparition d'une référence standard de performance [56]. Les plates-formes Java Card existent depuis 1997, et pourtant il y a relativement peu d'initiatives dans le domaine. Il y a un besoin d'outils qui puissent être utilisés pour démontrer l'efficacité d'une machine virtuelle Java Card et qui puissent fournir des critères de comparaison des performances des plates-formes Java Card, permettant ainsi aux acteurs de la communauté Java Card de prendre des décisions sur les environnements qui peuvent correspondre à leurs besoins.

En effet, s'il existe de nombreux benchmarks développés pour mesurer les performances des machines virtuelles Java, très peu de travaux se sont intéressés aux cartes à puce.

Quelque soit le domaine d'application, tous les développeurs et tous les utilisateurs sont intéressés par la performance des systèmes car leur but est d'utiliser le système le plus performant pour le prix le plus bas. Dans le cadre des cartes à puce, ce besoin est contrebalancé par l'importance de la sécurité. Pour une carte à puce une bonne performance peut être atteinte au détriment de la sécurité. Un compromis doit donc être trouvé entre ces deux facteurs. La table 2.1 résume l'importance de différents critères qui interviennent dans la sélection d'une plateforme suivant le domaine d'application.

2.2 La mesure de performance

2.2.1 Généralités

La mesure de performance est généralement comprise comme l'une des méthodes de l'évaluation de performance aux côtés de l'utilisation de modèles analytiques et de simulations [56] [74] [62]. Une autre famille d'outils permettant d'évaluer les performances d'un système sont les moniteurs.

La table 2.2 d'après Lilja [74] et Jain [56] présente différents critères à prendre en compte pour concevoir une application qui évalue la performance d'un système. La mesure de performance est donc un type d'évaluation de performance où il faudra instrumenter. La précision des mesures peut être relativement variable, ce qui entraîne un besoin de validation des mesures.

Les métriques de performance incluent :

- **Le temps de réponse**
  La figure 2.1 fournit deux définitions du temps de réponse réaliste. Dans tous les cas, le temps de réponse est un intervalle de temps. Tous benchmark utilisant cette métrique doit se conformer à l'une de ces définitions.
- **Le débit/taux**

Industrie	Performance	Sécurité	Mémoire	Prix
Télécom	+	–	++	-
Bancaire	-	+	–	++
Multimédia	+	++	-	–
Transports	++	+	–	+
Identité	++	+	+	+

TAB. 2.1 – Critères relatifs de sélection des plates-formes dans différents domaines industriels. Source [20]

Un cas classique est le nombre d'instructions par seconde en millions (MIPS) ou le nombre d'opérations flottantes par seconde (Mflops) ou encore la bande passante en bits par seconde.

- **L'efficacité**
  C'est le ratio entre un débit maximal théoriquement atteignable et le débit achevé. Par exemple, si un réseau local en 1000 Mbps (méga bit par seconde) atteint un débit maximum de 850 Mbps, l'efficacité sera de 85
- **La disponibilité**
  C'est la fraction de temps pendant laquelle le système est disponible aux requêtes des utilisateurs.
- **La fiabilité**
  Celle-ci est souvent représentée en termes de secondes sans erreurs.

Quand on mesure une performance de manière expérimentale, on est éventuellement sujet à du bruit récolté pendant la mesure - des erreurs expérimentales. Il faut donc des moyens de pouvoir rapprocher la mesure qui peut être faussée d'une valeur de référence que l'on essaie d'approcher par cette mesure.

Des mesures peuvent être plus ou moins **précises** suivant la dispersion des différentes valeurs expérimentales obtenues. Par analogie avec un jeu de fléchettes, les fléchettes sur une cible peuvent être relativement groupées vers un point de la cible si elles ciblent précisément ce point, ou relativement dispersées.

Une mesure peut être plus ou moins **exacte** suivant qu'elle est proche ou non de la valeur de référence. Pour reprendre l'analogie avec le jeu de fléchettes, les fléchettes sur une cible peuvent être plus ou moins proches du double centre (ou *bull's eye*) suivant que le joueur ait lancé avec exactitude ou non (cf figure 2.2).

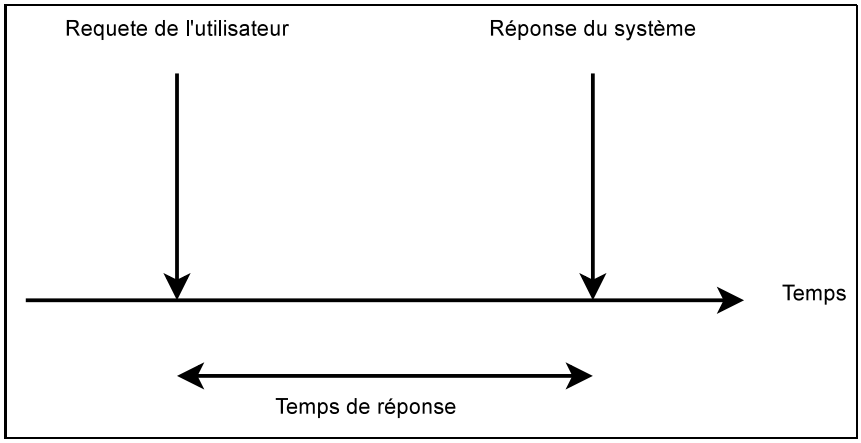
Finalement, la **résolution** des mesures est le plus petit changement détectable entre deux mesures.

Étant donné une famille de mesures similaires, un benchmark devra trouver une valeur qui représente ces mesures. Plusieurs données peuvent être tirées des mesures : les moyennes arithmétiques, géométriques et harmoniques, l'écart type, la valeur médiane. Dans le cas d'une mesure temporelle, la moyenne arithmétique est indiquée [74]

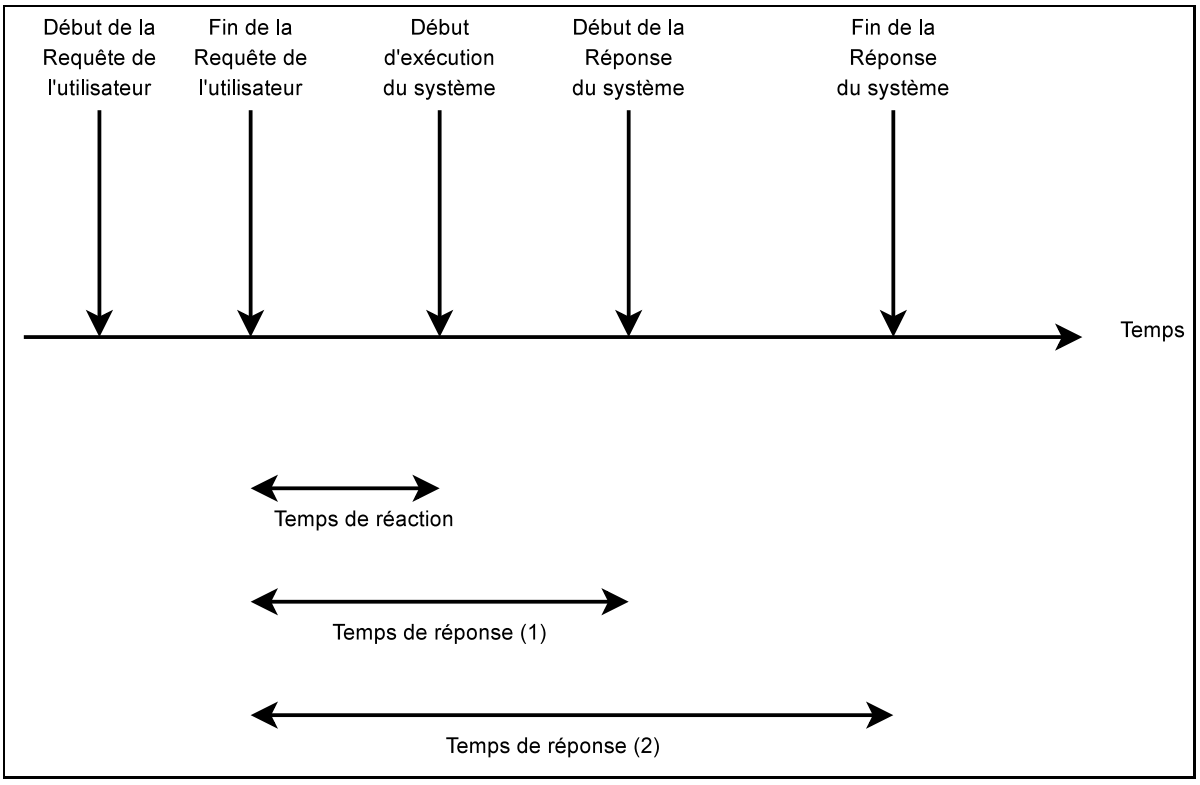
Pour une mesure de performance temporelle sur ordinateur, la précision des mesures va dépendre de la dispersion des valeurs mesurées. Sur un nombre assez large de valeurs (plus de 30), on s'attend à une distribution normale des mesures [62] [74]. Cette distribution sera effectivement normale si les facteurs qui causent les bruits sont indépendants. L'exactitude des mesures va dépendre de l'écart entre la moyenne des mesures faites et une valeur "réelle" qui correspond au temps d'une exécution sans bruit (cf la figure 2.3). Par la suite un calcul d'intervalle de confiance permet de filtrer les valeurs peu probables [62].

Critère	**Modèles analytique**	**Simulation**	**Mesures**
Stade	Tous	Tous	Après le prototypage
Temps nécessaire	Faible	Moyen	Variable
Outils	Analyse	Langages de programmation	Instrumentation
Exactitude	Faible	Modérée	Variable
Coût	Faible	Moyen	Élevé

TAB. 2.2 – Critères pour sélectionner les techniques d'évaluation



Requête et réponse instantanées.



Requête et réponse réalistes.

FIG. 2.1 – Définitions du temps de réponse

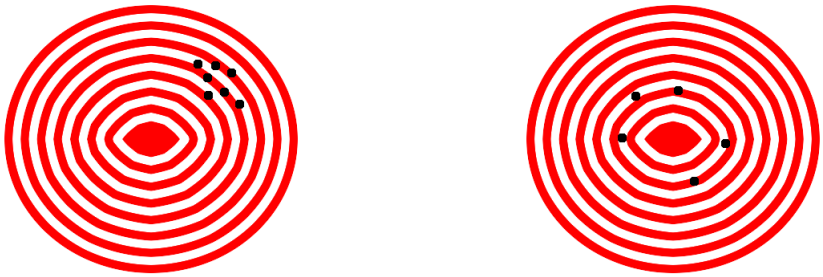


FIG. 2.2 – Un lancé précis mais peu exact, une autre exact mais peu précis - analogie du jeu de fléchettes

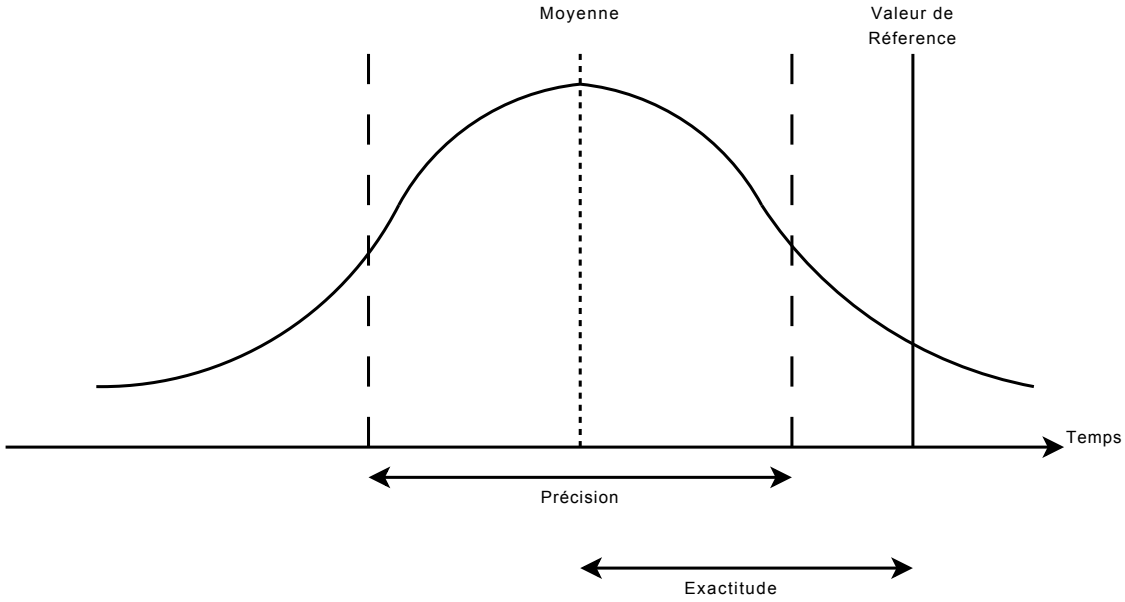


FIG. 2.3 – Distribution normale de données mesurées expérimentalement et estimation de l'exactitude et de la précision

2.2.2 Types de benchmarks

Jain [56] cite certains types de benchmarks. On peut retrouver dans l'ordre historique d'apparition :

- de simples **additions** (pour les tous premiers benchmarks),
- des **combinaisons d'instructions** représentatives,
- des **noyaux**, c'est à dire des *services* en fonctionnement, comme des algorithmes de tri, ce qui inclut des combinaisons d'instructions et non pas des instructions seules,
- des **programmes synthétiques**, qui sont des boucles représentatives de codes utilisables sur une machine, ce qui inclut des entrées/sorties,
- des **benchmarks d'applications**, qui utilisent directement les applications utilisées sur le système.

Les problèmes de représentativité sont cruciaux pour les programmes de mesure de performance. Ainsi les noyaux donnent une bonne idée de la performance du CPU sans être nécessairement représentatifs de ce qui va intéresser les développeurs d'applications sur la plateforme. Les programmes synthétiques mélangent les entrées/sorties aux performances du CPU, mais les proportions exactes des uns et des autres sont difficiles à évaluer à priori pour tous les systèmes. De plus ils peuvent ignorer certains mécanismes comme le cache.

2.2.3 Quelques benchmarks

Au cours de ces 40 dernières années, de nombreux programmes de mesure de performance sont apparus et ont acquis au moins pour un temps un statut de benchmark de référence. En voici une liste courte :

- **Le crible d'Ératosthènes** permettant de trouver les nombres premiers, à été utilisé pour comparer les performances de processeurs, d'ordinateurs et de langage de programmation [56].
- **La fonction d'Ackermann**, de par sa nature récursive, a été très tôt utilisée pour mesurer la performance des mécanismes d'appels de procédure en ALGOL [56].
- **Whetstone** [29] a été spécifiquement conçu pour la mesure de performance dans les programmes 949 ALGOL. C'est un programme synthétique relativement ancien (1975 - le premier programme de mesure synthétique date de 1969 [11]), c'est à dire qu'il tente de représenter les performances globales pour un total de 11 modules, où chaque module est représentatif des opérations fréquemment effectuées dans les programmes ALGOL. Les résultats sont présentés sous la forme de KWIPS (milliers d'instructions Whetstone par seconde).
- **LINPACK** [28] [32] a été développé par Jack Dongarra en 1983 pour l'Argonne National Laboratory et fait suite à des travaux similaires (**EISPACK**) datant des années 1970. Ce benchmark est constitué d'un certain nombre de programmes qui tentent de résoudre des équations linéaires. Les valeurs mesurées mettent en avant le calcul sur des nombres flottants, étant donné la quantité importante d'opération flottantes appelées. Les résultats sont présentés sous la forme de MFLOPS (millions d'instructions flottantes par seconde). Ces travaux et ceux qui ont suivi ont été développés en Fortran. Une part importante de ces benchmarks porte sur MPI (Message Passing Interface) qui est un mécanisme propre aux clusters et aux superordinateurs. Par conséquent, les résultats de ces tests ont été publiés très régulièrement portant sur les 500 ordinateurs les plus performants au monde [115] [31], qui font référence.
- **Dhrystone** [121] a été développé par Reinhold Weicker pour Siemens en 1984. Il est

constitué de nombreux appels de procédures. À l'origine, il fut développé en Ada, mais il est disponible en C, en Pascal et en Ada, et la version en C est la plus employée [56]. Les résultats sont présentés sous la forme de DIPS (instructions Dhrystone par seconde).

- **Les Boucles de Livermore** [77] consistent en 24 programmes de calcul scientifique. Ce benchmark a été conçu en FORTRAN spécifiquement pour les super-ordinateurs, mais il est utilisable sur des ordinateurs personnels. Les résultats, présentés en MFLOPS sont difficilement interprétables car ils ne se réduisent pas à une seule note synthétique. Les résultats unitaires sont présentés à l'aide de trois moyennes (arithmétique, géométrique et harmonique), un maximum et un minimum. Entre 40 et 60% du temps de calcul est pris par des opérations sur des nombres flottants [56].
- **SPEC** (Systems Performance Evaluation Cooperative) est une organisation fondée en 1988 regroupant des acteurs de l'industrie informatique qui tente de développer un ensemble de benchmarks standardisés [35]. La version 1.0 de SPEC est apparue en 1990 [24] et comporte : GCC (compilation de 19 programmes), Espresso (automate de conception électronique sur 7 modèles), Doduc (simulations de Monte Carlo), NASA7 (opérations sur les nombres flottants), LI (problèmes des 9 reines), eqnttot (résolution d'équations logiques), Matrix300 (utilisation de LINPACK sur des matrices 300x300), Fpppp (équations de chimie quantique en FORTRAN) et Tomcatv (calcul sur des polyèdres).
  Ces benchmarks sont focalisés sur la performance du CPU et mettent de côté les performances des entrées/sorties. Une note globale est donnée par rapport à un système de référence (à l'origine, un VAX-11/780) sous la forme : "*SPECthruput*"n@r où n est le nombre de processeurs sur la machine testée et r est le ratio entre la performance de la machine testée et la performance de référence.
- **Phoronix Test Suite** [95] est un benchmark développé par Phoronix Media en 2004 et un certain nombre d'acteurs de l'industrie informatique. Ce benchmark à été conçu en utilisant des logiciels libres pour des ordinateurs personnels uniquement. Les tests incluent MEncoder, FFmpeg, lm_sensors et des jeux OpenGL. Étant dédiés aux logiciels libres, ces tests fonctionnent sous Linux, Mac OS X, OpenSolaris, et les OS de la famille BSD.

SPEC est aujourd'hui considéré comme étant une référence pour la mesure de performance sur les ordinateurs personnels. C'est en partie du à la diversité des tests, à la grande part que prennent les applications réelles dans les tests, et aussi par le fait qu'il provient d'un effort de standardisation de la mesure de performance par un consortium composé de nombreux acteurs majeurs de l'industrie informatique et d'un nombre d'universités influentes. Dans sa version actuelle, cette famille de benchmarks est composée de plusieurs membres : SPEC CPU2006 (calcul sur des entiers et des flottants) [111], SPECviewperf 10 (opérations OpenGL) [26], SPEC MPI2007 (Message Passing Interface - pour superordinateurs), SPECjAppServer2004 (client/serveur en Java), SPECmail2009 (serveur e-mail), SPECsfs2008 (NFS), SPECpower_ssj2008 (consommation énergétique), SPECsip (SIP), Virtualization (travail sur plusieurs machines virtuelles), SPECweb2009 (serveur web).

Chacun de ces benchmarks est décomposé en sous modules. Par exemple SPEC CPU2006 [25] est composé de CINT2006 (calcul sur les entiers) et CFP2006 (calcul sur des flottants). CINT2006 est à son tour divisé en plusieurs programmes en C ou en C++. GCC fait toujours partie de CINT2006. On y trouve aussi des programmes d'intelligence artificielle (jeu de go), de compression vidéo (H264), de compression de données (bzip2) entre autres [25].

2.2.4 La mesure de performance en Java/J2ME

Il existe de très nombreux benchmarks pour les plates-formes Java sur ordinateur personnels.

Un premier exemple est **SPECjvm2008** [27] qui contient sept programmes (compression, base de données, compilation, décodeur MP3, parseur, génération d'image 3D et système expert), ainsi que des tests pour mesurer la performance du JRE, des entrées/sorties, des appels de librairies ... Chaque programme est exécuté avec différentes tailles de données en entrée.

Un autre exemple typique est le **Java Grande Forum Benchmark** [60] [61] qui a été développé en grande partie à l'université d'Édimbourg et l'université d'Adélaïde. C'est un benchmark synthétique qui est divisé en plusieurs groupes de programmes :

- Une partie séquentielle.
- Une partie orientée multithreading.
- Une partie MPI (pour super-ordinateurs).
- Une partie comparaison avec C (reprenant les même programmes implémentés en C.

Chaque groupe est lui même divisé en plusieurs classes d'application suivant la complexité de ces applications, de la mesure de performance sur les opérations élémentaires comme les appels de méthode jusqu'à la résolution d'équations de Euler pour la partie séquentielle. Un grand nombre de ces programmes proviennent d'autres benchmarks comme LINPACK, et sont un portage de ces programmes en Java. Les résultats sont présentés en termes de nombre d'instructions par seconde.

Du fait de la nature de Java (c'est à dire, l'utilisation de machine virtuelles, de ramasses miettes et de compilateurs Just In Time), des benchmarks comme **SciMark** [97] font du calcul scientifique avec à l'esprit de mettre en marche le compilateur JIT. Les résultats, en Flops, sont fortement influençables par le déclenchement ou non du compilateur JIT.

Morphmark [80] est un exemple de benchmark synthétique pour Java Micro Edition [55] avec une approche "jeux en Java pour mobile". Ce benchmark permet de décider si un téléphone est adapté pour jouer à un jeu. Les tests portent sur le graphisme du téléphone et sur les entrées/sorties de la machine virtuelle.

Grinderbench est un autre benchmark pour J2ME. développé par l'EEMBC (EDN Embedded Microprocessor Benchmark Consortium) [34]. Ce programme se distingue par le fait qu'il n'est pas un benchmark synthétique, mais qu'il inclut du code provenant d'applications CLDC 1.0 du "monde réel". Les algorithmes testés ne sont pas des portages de code C, mais des applications représentatives de l'usage qui est fait de J2ME dans les téléphones. Ainsi, nous y trouvons : un décodeur d'images PNG, un jeu d'échec, un parser XML, un programme utilisant des fonctions cryptographiques, un test de parallélisme. En dehors de GrinderBench, d'autres benchmarks ont été développé par l'EEMBC sur une thématique embarquée. La plus part d'entre eux sont des applications réelles mais il y a aussi quelques benchmarks synthétiques.

2.3 La mesure de performance en Java Card

2.3.1 Castellà

La première initiative a été celle de Castellà-Roca et al. dans [13] qui étudient les performances des cartes de paiement Java Card sans PKI. Ce qui est mesuré est la performance temporelle de différentes cartes pour des opérations de signature. Les auteurs argumentent leurs recherches en précisant que les applications visées sont des applications de micro paiement. En effet les applications de micro paiements consistent uniquement à autoriser des paiements par carte de montants très faibles et successifs, pour lesquels une forte authentification n'est pas nécessaire. Dans ce cadre, l'utilisateur est le débiteur d'une carte de paiement. Les mesures sont effectuées de bout en bout, c'est à dire qu'on n'essaie pas d'analyser les différents éléments mesurés individuellement.

Ce travail est resté ponctuel et n'a pas donné suite à d'autres initiatives concernant la mesure de performance.

Il y a dans ces travaux une comparaison des performances de différentes cartes sur une application d'utilité réelle, et il est possible d'ordonner les cartes de la plus performante à la moins performante. Cependant, le travail effectué est très partiel, puisqu'il ne concerne que la mesure de performance d'une application de signature.

2.3.2 Markantonakis

Markantonakis considère dans [76] que la performance de la couche communication est suffisamment importante pour éclipser la partie cryptographie des cartes qui est en général considérée comme étant la plus gourmande en temps de calcul. Le problème, d'après lui, n'est pas que les fonctions cryptographiques soient peu coûteuses, mais qu'on peut grandement améliorer les performances d'une carte en considérant les performances de l'API d'entrée/sortie. Il présente à cet effet les performances de plusieurs micro processeurs utilisés dans les cartes à puce concernant des algorithmes cryptographiques comme DES, SHA ou RSA.

Les entrées/sorties sur les cartes à puce doivent utiliser une des APIs standardisée pour communiquer avec la carte depuis une machine hôte. L'auteur présente deux de ces APIs : PC/SC qui est de facto la norme sous Windows, et OCF qui propose une API en Java et qui est indépendant de l'architecture et de l'OS de la machine hôte.

Des tests sont effectués sur deux types de cartes de type Java Card 2.0 (des cartes GemXpresso de Gemplus et des cartes SmartCafé de Giesecke & Devrient), et en utilisant deux types de CAD (GCR410, PCT-200), en utilisant un adaptateur logiciel (wrapper) Java pour PC/SC et OCF, le tout sous Windows NT.

Dans un premier test, la carte ne reçoit pas de données de l'hôte et lance une exception, ce qui renvoie juste un SW conforme à la norme ISO 7816. Dans un deuxième test, la carte ne reçoit pas de données du terminal mais renvoie n octets de données en retour. La carte reçoit n octets de données du terminal mais fait une exception et renvoie un SW dans un troisième test. Enfin, lors d'un dernier test, la carte reçoit n octets de données du terminal et en renvoie m, avec $n, m \in \{10, 20, 30, 40\}$.

Les résultats sont donnés sous forme de moyenne et d'écart type. On peut y voir que PC/SC est en moyenne 18,6% plus rapide que OCF en général, et que OCF est particulièrement contre-performant dans les phases d'initialisation de la connexion avec la carte (tests Connect, Select), mais même en écartant ces résultats, PC/SC est toujours plus performant que OCF quelque soit la carte et quelque soit le lecteur.

OCF donne des performances moins dispersées que PC/SC sur le CAD GCR, mais les résultats sont opposés quand on considère les résultats sur le CAD PCT.

Il y a une linéarité manifeste des résultats en fonction du nombre d'octets envoyés et reçus. Cependant, avec OCF, avec le CAD PCT-200, la carte GemXpresso subit de très fortes contre-performances sur les tests de type 3, c'est à dire sans envoyer d'octet de donnée, mais en recevant n octets en retour. L'auteur considère ces résultats là comme étant dûs à une mauvaise interprétation des APDUs par le CAD PCT et conseille de ne pas les prendre en compte.

De manière générale, cet article met simplement le doigt sur le fait qu'un calcul cryptographique moyen est équivalent en termes de temps de performance à l'envoi et à la réception de 10 octets avec des APIs communes. Les résultats sont malheureusement légèrement datés, puisque OCF n'est plus vraiment utilisé aujourd'hui, et que Java Card 2.0 a près de 10 ans.

L'applet qui permet de faire les mesures n'est pas explicitement donnée, mais il est facile de la reconstituer.

L'auteur se contente de donner les performances théoriques des micro processeurs sans pour autant faire des tests avec les cartes qui servent à mesurer la performance des entrées/sorties, ce qui biaise quelque peu la thèse de l'article selon laquelle la cryptographie n'est pas le goulot d'étranglement au niveau performance dans une carte à puce. Ici, l'auteur ne intéresse à rien d'autre qu'a la performance combinée de deux APIs avec deux CADs et deux cartes. Il ne s'intéresse pas à la performance des cartes seules, ni à d'autres fonctionnalités que les mécanismes d'entrée/sortie. Aucun travail ultérieur ne viendra compléter ce papier.

2.3.3 SCCB

SCCB (Smart Card CNAM Benchmark) est un projet à l'initiative du CNAM-Cedric démarré en 2003 [33]. Le but déclaré de ce projet est de fournir un outil pour mesurer la performance des Java Cards. Une page web est consacrée à ce projet :
`http ://deptinfo.cnam.fr/ paradinas/sem/sccb/`.

Le projet est divisé en deux parties :

- `SCDevice` contient les procédures de mesure de performance qui ont trait aux caractéristiques physiques et techniques de la carte. Le package `TechnicalFeatures` permet de retrouver des information comme l'AID, la durée d'un cycle, etc ... Ces données sont utilisées pour affiner l'analyse des performances. Le package `PhysicalFeatures` contient des modules qui évaluent les caractéristiques physiques de la carte : tailles des mémoires volatiles et non volatiles, temps d'écriture dans chaque type de mémoire, taille de la pile ...
- `SCSystem` contient tous les tests de la couche logicielle, c'est à dire des bibliothèques Java Card et Global Platform. On y fait la distinction entre ce qui relève de la machine virtuelle Java Card (JCVM) et ce qui relève de l'API Java Card.

Les tests sont effectués sous Windows XP avec des CADs USB et avec plusieurs cartes. Les cartes sont des Java Card provenant de différents fournisseurs et satisfaisant différentes versions des normes Java Card et Global Platform. Les résultats obtenus sont cohérents d'une machine à l'autre avec une même carte et un même CAD. Il n'y a pas de traitement statistique des résultats, et il n'y a pas non plus de mécanisme efficace d'isolement des performances. La performance d'une opération simple comme l'addition est en fait toujours mesurée sur des variables globales, et on ne gère pas les bytecodes mis en œuvre. La conséquence est que toutes les opérations arithmétiques ont le même temps de performance. Le projet n'a pas été largement diffusé, et est resté confidentiel, ce qui l'a empêché de fournir une référence globale de performance, de même qu'il n'est pas devenu un standard. Toutefois, il a permis d'adresser clairement la problématique et d'avoir une idée plus précise de la manière de mesurer des plates-formes Java Card, des principes de base utilisés plus tard dans le projet MESURE qui sera décrit plus loin dans ce document.

2.3.4 Erdmann

Le travail, relativement récent, de Erdmann [38] distingue différents domaines d'applications et mesure des opérations d'entrée/sortie, des fonctions cryptographiques, de consommation d'énergie, de JCRE. Les résultats sont présentés sous forme de diagrammes en araignées avec des degrés de ramifications. L'outil n'est malheureusement pas disponible D'autre part, les plates-formes considérées sont exclusivement celles de cartes de Infineon Technologies.

L'auteur classifie les différentes mesures traitées de la manière suivante :

- les opérations arithmétiques et logiques,

```
//[...]
 public class Appletname extends Applet {
     final static byte LOOP1 = (byte) ISO7816.OFFSET_INS+5;
     final static byte LOOP2 = (byte) ISO7816.OFFSET_INS+6;
     final static byte LOOP3 = (byte) ISO7816.OFFSET_INS+7;
     //[...]
for (i1=0;i1<buffer[LOOP1];i1++){
    for (i2=0;i2<buffer[LOOP2];i2++){
         for (i3=0;i3<buffer[LOOP3];i3++) {
             Util.arrayCopyNonAtomic(EEdata1,(short)5,
                         RAMdata1,(short)8,(short)256);
             // Anweisungsabarbeitung
          }
    }
 }
```

FIG. 2.4 – Boucle contenant la partie a isoler : une copie de données en EEPROM vers la RAM

```
for (i1=0;i1<buffer[LOOP1];i1++){
    for (i2=0;i2<buffer[LOOP2];i2++){
        for (i3=0;i3<buffer[LOOP3];i3++) {
            //[...] // keine Anweisungsabarbeitung
         }
    }
 }
```

FIG. 2.5 – Boucle permettant d'isoler les mesures

- l'écriture et la lecture en mémoire RAM et EEPROM,
- l'appel de méthode,
- les opérations sur les tableaux,
- les branchements,
- le mécanisme de transaction,
- la sécurité (chiffrement, signature),
- les spécificités de la norme Visa Open Platform.

Erdmann utilise une technique d'isolation des mesures pour observer le comportement des cartes sur des points précis.

Ainsi, il est question de faire des boucles imbriquées dans lesquelles elle laisse le code a isoler (cf figure 2.4). La performance de cette boucle est ensuite comparée à la performance d'une boucle vide (cf figure 2.5), pour isoler la performance du code en question. Le temps de communication est écarté, intéressant pas l'auteur.

Dans tous les cas, la moyenne des mesures est retenue comme étant la valeur représentative de la performance. Il n'est pas fait état du nombre de mesures effectuées.

Des travaux sont conduits sur trois machines hôtes et deux lecteurs, et au moins 9 cartes différentes (qui ne sont pas explicitement nommées).

L'auteur soutient, chiffres à l'appui, que les résultats sont peu sensibles à la plate-forme de

mesure utilisée. Ainsi elle teste pour une carte et un lecteur donnés, trois différentes machines hôtes, sur une série de tests. Se basant sur l'apparente similarité des résultats, l'auteur conclut que la machine hôte n'a pas d'incidence sur la mesure. Il est a noter que les mesures ne sont effectuées que sur une seule boucle, et que la différence entre deux mesures (avant isolation du bruit) varient en fait de parfois 20 pour cent d'une machine à l'autre. Ainsi on peut douter de l'exactitude de cette conclusion.

D'autres mesures de performance, toujours sur des opérations de copie de tableaux, sont effectuées avec une machine hôte et une carte donnée, et en comparant les résultats obtenus avec deux lecteurs différents. Les tests sont effectués avec différentes tailles de boucles. Encore une fois, l'auteur conclut que les résultats sont suffisamment similaires d'un lecteur à l'autre pour que le CAD n'ait pas d'incidence significative sur les mesures.

Dans d'autres tests, l'auteur tente de déterminer la taille d'une page d'EEPROM en faisant varier la taille des tableaux copiés et en observant différentes tailles de boucles. Seule une des cartes testées permet d'identifier la taille des pages d'EEPROM de cette manière. En effet, elle seule donne de manière consistante des résultats indiquant une contre-performance à chaque fois qu'on copie 16 octets de plus qu'au test précédent, ce qui semble indiqué une taille de 16 octets pour les pages d'EEPROM. L'impossibilité à déterminer la taille des pages dans les autres cartes provient, d'après l'auteur, de l'impact d'un cache sur la mesure. Par la suite, l'auteur a tenté sans succès de contourner le problème de cache en utilisant des copies de grands tableaux sur quatre cartes.

Les mesures suivantes concernent le chiffrement DES. L'auteur compare les mesures effectuées avec les temps de chiffrement spécifiés pour un microcontrôleur donné mais l'écart entre les deux étant important (d'un facteur de 1000), l'auteur conclut que les cartes ne font pas directement appel à une fonction native, et que du bruit, provenant de la communication CAD - carte et de la JCVM perturbe les mesures.

Ensuite, l'auteur a testé des éléments du langage Java, comme `if`, `else`, `for`, `break`, `continue`, `switch`, `while`. Certaines cartes se sont avérées non testables, d'autres n'ont pu être testées que sur une seule machine hôte, et un seul lecteur. Ce genre de problème pourrait expliquer pourquoi l'auteur voulait, dans un test précédent, expliquer que les machines hôtes et les CAD sont équivalents en terme de mesures. D'ailleurs l'auteur fait la même affirmation sur les tests qui ont pu être effectués ici avec une même carte dans des circonstances différentes.

Des tests ont été effectués avec une applet de porte monnaie. Les mesures ont consisté à mesurer le temps nécessaire pour effectuer une des commandes complexes sans tenter d'isoler le bruit des mesures. Ces mesures ne sont pas utilisées pour noter la performance des cartes, mais seulement à titre indicatif pour une application "représentative".

Finalement, des mesures sont effectuées avec un oscilloscope. Cependant l'auteur n'essaie pas de recouper les mesures faites à l'oscilloscope avec celles faites avec juste une machine et un CAD.

L'attribution de notes tient en compte les performances en terme de chiffrement, JVM, et de mémoire.

Ce travail, est relativement complet par rapport aux autres travaux concernant la mesure de performance dans les cartes à puce, et il donne une idée de l'état des benchmarks sur carte à puce dans des entreprises comme Infineon.

2.3.5 Fischer

Le travail de Fischer dans [40] prolonge les travaux de Erdmann.

Toujours dans le cadre d'un benchmark pour les cartes produites par Infineon, il compare les performances d'un code simple (une boucle cf 2.6) écrit en Java et son équivalent en langage

```
// PARAMETER ( LOOP *) VOM CCTERMINAL EMPFANGEN
for ( i1 =0; i1 < buffer [ LOOP1 ]; i1 ++)
{
  for ( i2 =0; i2 < buffer [ LOOP2 ]; i2 ++)
  {
    for ( i3 =0; i3 < buffer [ LOOP3 ]; i3 ++)
    {
       b01 =( byte ) i3 ;           // variable to variable
       b02 =( byte )127;            // constant to variable
       s01 =( short ) i3 ;          // variable to variable
       s02 =( short )127;           // constant to variable
    }
}
// ANTWORT AN CCTERMINAL SENDEN
```

FIG. 2.6 – Boucle mesurée par Ficher

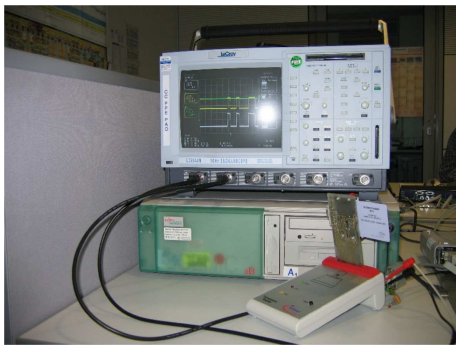


FIG. 2.7 – Les mesures de Fischer avec un oscilloscope

d'assemblage natif propre à la carte. La partie en Java est détaillée au niveau bytecode. Les mesures sont effectuées sur deux cartes.

Pour la première carte, les temps mesurés en Java sont environ 22 fois plus long que les temps mesurés en natif. Le facteur entre les durées d'exécution du programme natif et du programme Java est de 14 pour la seconde carte qui est par ailleurs légèrement plus rapide que la première.

L'auteur constate cependant que le fait d'utiliser une machine telle qu'un PC pour mesurer la performance provoque un bruit qui se fait particulièrement sentir sur les petites tailles de boucles. Par conséquent, l'auteur propose l'utilisation d'une plate-forme temps - réel : JNut. L'argumentation est que les bruits impactent moins les mesures ainsi. JNut est une plate-forme dérivée du projet Ethernut, dont le but est de produire des petites plates-formes embarquées accessibles via Ethernet. Le but premier de JNut est de fournir une plate-forme dédiée aux applications Ethernet. Mais la plate-forme comprend Nut/OS, un OS embarqué temps-réel. Ici, l'auteur utilise un PC, un CAD, JNut et des outils Infineon pour "intercepter" les APDUs qui transitent au niveau des entrées/sorties de la carte. Le boîtier JNut est relié à un second PC et permet de garder trace des échanges en temps réel.

D'après l'auteur, la précision des mesures est améliorée grâce à ce système.

Cependant, l'auteur ne détaille pas ce qui le pousse à invalider les mesures effectuées directe-

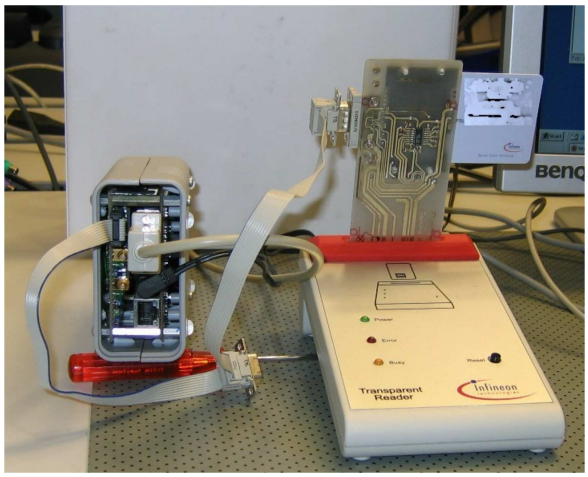


FIG. 2.8 – Le dispositif de Fischer avec JNUT

ment avec un PC.

Aucune mesure n'est fournie avec ce système.

Bien qu'a priori inadapté à la mesure de performance sur carte à puce, le système de mesure mis en place avec JNut permet néanmoins de se rendre compte du besoin de précision dans la mesure de performance sur les cartes. Le matériel utilisé devrait rendre ces mesures difficiles à réaliser à grande échelle.

2.3.6 Rehioui

Un autre travail intéressant est celui mené par le groupe IBM BlueZ secure systems [102] pour tester les performances de l'algorithme de cryptage DES, des opérations de lecture/écriture dans les mémoires RAM et EEPROM.

Les cartes concernées ici, sont essentiellement des cartes IBM JCOP 41 qui implémentent la version 2.1 de Java Card.

L'auteur fait état d'une certaine dispersion des résultats, et argue qu'un certain nettoyage statistique doit avoir lieu pour obtenir des données plus fiables.

Ainsi, un algorithme de nettoyage itératif des valeurs obtenues est présenté. Il consiste à éliminer successivement les valeurs qui s'éloignent de la moyenne de plus d'une fois l'écart type. L'algorithme s'arrête quand l'échantillon de valeurs atteint une proportion donnée du nombre de mesures initiales. Ce dernier nombre est fixé par l'utilisateur. L'idée est de finir en se rapprochant des valeurs les plus fréquentes, c'est à dire, du pic le plus élevé dans la distribution des valeurs. Cependant, en cas de distribution "en peigne", cet algorithme pourrait ne rien donner d'acceptable.

Une applet est présentée pour estimer le temps de communication entre la machine hôte et la carte (cf 2.9). Un calcul permet d'isoler le temps d'exécution de ce qui se déroule sur la carte et de mettre de côté le temps de communication et les bruits dans la mesure sur la machine hôte. L'auteur propose une table de bruits de communication en entrée et en sortie pour différents APDUs envoyés et reçus. Ces tables servent à toutes les mesures subséquentes pour déduire des mesures effectuées tout ce qui est indésirable.

Une équation est proposée :

$$communicationoverhead = ohIn + ohOut - setupoverhead$$

```
byte[] buf = apdu.getBuffer();
switch(buf[ISO7816.OFFSET_INS]){
case INS_NOOP: //do nothing
   return;
case INS_READ_IN: //read data from apdu
   apdu.setIncomingAndReceive();
   return;
case INS_COPY_OUT: //send data in apdu
   apdu.setOutgoingAndSend((short)0, Util.getShort(buf,ISO7816.OFFSET_P1));
   return;
default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
}
```

FIG. 2.9 – Applet de mesure du overhead

avec $ohIn$ et $ohOut$ les bruits causés respectivement par l'entrée et par la sortie. Cependant $setupoverhead$ n'est pas défini formellement et rien ne permet de le calculer.

Par la suite, des mesures sont présentées concernant un calcul de chiffrement/déchiffrement avec DES et triple DES sur des tableaux de différentes tailles. La performance en triple DES est meilleure que la performance en DES simple sur les cartes JCOP 41, sans que l'auteur n'explique pourquoi.

De la même manière, des mesures sont effectuées sur des copies de données d'un tableau à un autre, en discriminant la nature de la mémoire qui contient les données (RAM ou EEPROM). Les mesures sont faites sans transaction, puis lors de transactions, et les résultats sont comparés. Ces mesures en particulier permettent de donner des indices quant à la taille des pages mémoire, grâce à certaines tailles de copies pour lesquelles les performances ne sont pas linéaires avec les autres. Toutefois, l'auteur ne détaille pas les déductions en question, ni la taille des pages. Les mesures sur la RAM sont aussi effectuées dans une transaction mais ne sont pas plus détaillées.

Enfin, l'auteur présente une méthodologie pour mesurer la performance de bytecode individuel. En effet, si on fait une mesure sur une commande pendant laquelle n bytecodes $c_i, 0 \leq i < n$ sont exécutés dans la machine virtuelle, alors on peut tenter de faire une mesure avec n' bytecodes $c'_i, 0 \leq i < n'$. Ces deux mesures posent pour nous le début d'un système d'équations qui, une fois résolu, nous permettra de déduire la performance individuelle de chacun des bytecodes. Aucune mesure n'est effectuée par l'auteur, et aucun système n'est explicitement décrit. De plus aucune méthodologie n'est présentée pour concevoir des applets qui satisfassent ces équations.

2.3.7 Papapanagiotoy

Papapanagiotoy et al. évaluent dans [89] les performances de deux protocoles de validation et de révocation de certificats en ligne : OCSP et SCVP qui sont les plus utilisés dans l'associations de certificat pour les transactions sécurisées dans les cartes à puces. Les certificats testés sont des certificats X.509 [116]. Ceux ci sont en effet les plus utilisés aujourd'hui.

Ils implémentent ces protocoles dans deux plates-formes Java Card différentes pour pouvoir déterminer quel protocole est le plus efficace pour l'utilisation des cartes à puce.

Cela implique le développement d'une applet qui récupère un certificat X.509 et qui ait besoin de déterminer sa validité. L'applet n'est pas proposée. Les cartes testées sont deux cartes se

conformant à la norme Java Card 2.1.1, mais leur nature exacte n'est pas révélée. Les commandes testées concernent par exemple l'envoi de certificat, la création/l'envoi de requête, l'obtention d'une réponse, etc.

Les mesures sont faites depuis une machine hôte. On mesure le temps pris entre l'envoi d'une commande APDU et le retour de la réponse APDU. Chaque test consiste en une série d'envois et de réceptions d'APDUs (en effet, il faut utiliser plusieurs APDUs pour envoyer un seul certificat, par exemple). Chaque mesure est effectuée cinq fois, et on observe la moyenne arithmétique, la valeur médiane et l'ecart type. Une des difficultés liées à ce test est que les cartes en viennent vite à manquer de mémoire, ce qui entraîne une réinstallation de l'applet.

Les tests présentés ne comportent que la partie OCSP. Les résultats obtenus permettent d'établir une performance de référence concernant une applet d'OCSP sur deux cartes. Aucune mention n'est faite de l'influence du CAD, ou de l'OS, et on se restreint à la mesure de performance sur les certificats. Ainsi, comme les mesures sont menées de bout en bout, c'est à dire qu'on n'essaie pas de décomposer les performances individuelles des mécanismes utilisés, la seule appréciation des cartes qu'on peut avoir est celle concernant l'applet d'OCSP développée par les auteurs, et elle n'est pas dévoilée en détails.

2.3.8 Chaumette-Sauveron

Chaumette et al. [14] [5] ont présenté une plateforme de grilles Java Card. Cette plateforme propose de distribuer une application complexe en s'aidant d'un cluster de cartes à puce. L'idée derrière l'utilisation de grille de cartes à puce est qu'une application distribuée peut faire usage de cartes à puce pour garantir la sécurité des opérations effectuées. Plus particulièrement, cette publication propose une évaluation des performances de cette architecture. Les calculs effectués consistent à prendre un cluster de 9 cartes d'un type donné et de faire des calculs de fractale distribués sur l'ensemble du système. Les auteurs n'ont pas tenté de calculer le sur-coût du à la plateforme utilisée, mais ils affirment qu'un autre test incluant l'ouverture d'un canal sécurisé a permis de montrer que le sur-coût est négligeable.

Un estimation de mise à l'échelle des performances est aussi présentée. Sur un type de carte donné, les auteurs montrent que l'amélioration des performances en augmentant le nombre de cartes traitant le programme est presque linéaire, même si les résultats ne sont pas optimaux.

Les auteurs estiment aussi que les résultats ne sont pas satisfaisants en termes de performance, en particulier si on compare ces résultats à ceux qu'on pourrait obtenir dans le cadre d'un cluster plus traditionnel, mais ils arguent que le but ici est la sécurité et non la performance.

2.3.9 Guyot

Guyot et al. ont décrit dans [46] comment gérer la mobilité des sessions en gardant des information sur une carte à puce. Dans ce contexte particulier, on évalue les performance des cartes à puces, en implémentant des services réels et en observant le temps que les cartes prennent à suspendre et à reprendre une session. Deux expérimentations ont été entreprises sur dix plates-formes Java Card (de Gemplus, Oberthur et G&D). La première d'entre elles mesure la durée écoulée entre la validation du code PIN utilisateur par la carte et le moment où la session précédente est restaurée d'une manière utilisable à l'écran. La seconde mesure la durée de chargement d'une session dans une carte à puce.

Guyot décrit aussi dans [46] d'autres expérimentations concernant la mesure de performance dans des plates-formes Java Card. L'auteur évalue tout d'abord les lecteurs en entrées/sorties en proposant un programme qui reçoit un APDU d'une taille arbitraire et qui répond simplement

par un Status Word, ou qui reçoit un APDU composé uniquement d'un entête et qui répond par un APDU de taille correspondant aux paramètres (P1 et P2) de la commande APDU. Un temps en secondes est donné pour plusieurs couples de CAD et de cartes à puce, mais il n'est pas fait mention de ce qui est pertinent dans ce temps. Aucune mesure d'isolation n'est prise. On fait simplement s'exécuter un programme sur plusieurs cartes à puce et plusieurs CADs. L'influence du lecteur n'est par conséquent pas vraiment démontré. Par exemple, dans les chiffres fournis, il semble que les CADs branchés en PS/2 soient toujours plus "mauvais" pour les mesures (les mesures prennent plus de temps) qu'avec des CADs USB. Il n'y a pas de notion de temps de communication entre le CAD et la carte à puce exprimé.

Ensuite, l'auteur propose :
- un programme qui fait appel à une fonction de hachage
- un programme qui fait un chiffrement ou un déchiffrement
- un programme qui fait des copies en mémoire (RAM, EEPROM)
- un programme qui génère des nombres aléatoires
- un programme qui mesure la taille de la pile
- un programme qui teste les INS acceptés par la carte à puce
- un programme qui teste les CLA acceptés par la carte à puce

Il n'est, encore une fois, jamais question d'isoler les parties de code qui nous intéressent. Seul OCF est utilisé comme moyen de communication avec les cartes à puce. Les programmes mesurés sont sans doute assez éloignés de programmes utilisables dans une application "réelle". Les codes sources de ces programmes sont en partie donnés. Il est à noter qu'aucune valeur représentant la dispersion des mesures n'est donnée.

2.3.10 Poll et al.

Poll et al. décrivent en 2007 les résultats d'une comparaison entre plusieurs cartes à puce compatibles Java Card [81]. Les comparaisons entre cartes sont d'ordre général, comme les versions de Java supportées, la quantité de mémoire RAM ou s'il y a des applets pré-installées, autant que les temps d'exécution de certains programmes Java Card. Les travaux présentent une méthode d'isolation du temps d'exécution avec une granularité non spécifiée. Cette isolation permet d'ignorer les entrées/sorties entre autres. Aucune estimation de bruit ou de l'exactitude des mesures n'est fournie. Les tests portent sur huit cartes anonymisées et comportent des tests de signature, de chiffrement symétriques et asymétriques.

La plupart des tests sont simplement des tests pour vérifier l'existence d'une méthode optionnelle en Java Card. Les résultats sur les mesures de chiffrement et de MAC pour DES, 3DES et AES avec différents *bourrages*. De la même manière, différentes tailles de clés RSA sont testées pour faire des signatures SHA-1 et MD5.

Les temps sont donnés en millisecondes.

2.3.11 Tews

Les travaux de H. Tews et B. Jacobs [114] sont relativement récents (2009). Ces travaux concernent les cartes à puce utilisées dans les transports en argumentant que de nombreux systèmes modernes ont des défauts de sécurité. Dans le cadre de OV-Chipkaart (aux Pays Bas), les auteurs se proposent d'utiliser des protocoles à bases de clés publiques qui mettent plus en avant le secret de l'identité du porteur mais qui sont aussi plus gourmands au niveau du calcul sur la carte à puce.

Sur une carte à puce donnée, les auteurs utilisent des algorithmes cryptographiques à clés publiques (RSA et un algorithme à courbe elliptique) sur des grands nombres (de tailles variables, mais codés dans la plateforme Java Card avec des entiers 32 bits). Le protocole utilisé est celui suggéré par Stefan Brands [10], qui décrit une approche des infrastructures à clés publiques permettant entre autres de sélectionner et de limiter les informations circulant par les usagers des contrôleurs de l'infrastructure.

Les auteurs ont procédé à des tests de performance sur des cartes à puce Athena IDProtect et des cartes NXP JCOP31 (toutes deux compatibles Java Card 2.2.2) en utilisant d'un côté au maximum le coprocesseur cryptographique des cartes à puce et de l'autre, pour référence, en implémentant les algorithmes en Java Card.

Les mesures sont effectuées sans tentative d'isolement des parties concernées, et elles incluent par conséquent un maximum de bruits. S'ils ne tentent pas d'isoler les temps d'exécution de ce qui les intéresse particulièrement, les auteurs présentent cependant plusieurs mesures effectuées individuellement sur chacune des parties du programme qui les intéresse. En outre les mesures utilisent de manière systématique un lecteur CCID piloté depuis un PC sous Linux.

Une partie des mesures concernent les algorithmes cryptographiques sur des clés de 512 bits à 2048 bits, et une partie démontrent les temps d'exécution d'une librairie de calculs de multiplication et de modulo sur les grands nombres (ce qui sert dans le cadre du protocole présenté par Brands), en faisant par exemple varier les tailles des nombres pour ces différentes opérations.

Les cartes à puce utilisées sont toutes les deux relativement récentes (2008). Malgré cela, certains des calculs mesurés sont extrêmement coûteux en termes de temps d'exécution. Les calculs effectués pour l'exécution du protocole au complet durent de 5 à 20 secondes en fonction de la taille des clés (de 512 à 2048) RSA en utilisant le coprocesseur. Comme on peut s'y attendre les calculs sont extrêmement longs en Java Card pur (environ 75 minutes pour une clé de 512 bits).

En conclusion, les auteurs sont relativement sceptiques et pessimistes quant à l'utilisation du protocole avec une clé de 1280 bits. Ce dernier calcul, qui est leur objectif initial, prend environ 10 secondes de temps d'exécution. Un autre calcul sur une clé de même taille mais en révélant des informations sur le porteur de la cartes à puce, le protocole s'exécute dans un temps d'environ 5 secondes. Pour certains calculs utilisés comme une multiplication de Montgomery (ce qui n'est pas disponible dans l'API Java Card), le temps de calcul prends 25 secondes en Java Card pur, mais par une astuce de programmation, les auteurs parviennent à utiliser le coprocesseur cryptographique pour effectuer cette opération, ramenant ainsi le temps d'exécution à environ 0.3 secondes.

Les temps de calculs impliqués sont totalement irréalistes dans le cadre d'une application destinée aux transports. Pour les auteurs, la seule solution pour parvenir à intégrer ce protocole sur des plates-formes Java Card, consisterait à étendre l'API Java Card aux opérations sur les grands nombres (qui par ailleurs sont souvent implémentées de manière native sur les cartes à puce, mais indisponibles au niveau de l'API).

De manière générale, les auteurs ne se concentrent pas du tout sur la mesure de performance mais sur la réalisabilité de l'implémentation du protocole de Brands. Les méthodes utilisées concernant la mesure de performance ne sont pas présentées. Les auteurs parviennent à cibler le cœur de l'intérêt d'un outil de mesure de performance : l'utilisabilité d'une application dans un contexte de vie réelle, en tentant de garantir au maximum le respect de la confidentialité des informations. Sans doute le protocole est il réalisable en utilisant une carte à puce encore plus récente et en utilisant une bibliothèque sur les grands nombres implémentée de manière native. Les méthodes de conception de l'implémentation sont par ailleurs peu pratiques à l'heure

actuelle : ils utilisent des entiers sur 32 bits, des calculs en grande partie codés en Java Card, ou plutôt, nécessitant d'utiliser une version alternative de Java Card et le tout, pour des cartes sans contacts, ce qui suppose un coût matériel relativement important pour les cartes à puce. L'issue de ce travail dépend donc fortement des avancées dans les technologies entourant les cartes à puce dans les années à venir.

2.3.12 Attaques

Bien que n'étant pas axés sur la mesure de performance, un certain nombre de travaux portant sur les attaques et les contre - mesures correspondantes sur les cartes à puce peuvent montrer ce que les notions de temps et de performance d'une carte à puce peut apporter. Voici trois de ces travaux.

Les travaux de Sauveron [105] [15] portent sur la sécurité des cartes à puce. Les cartes à puce sont potentiellement vulnérables à des attaques par canaux cachés. Par exemple, on peut observer le temps d'exécution d'une fraction de code, la consommation en courant ou les émissions électromagnétiques émises par la carte à un moment donné pour en déduire une signature du code. Sauveron identifie une nouvelle classe d'attaque pour déterminer les signatures de ces portions de code. Elle est composée de deux attaques. On peut utiliser des *glitches* sur le canal d'entrée/sortie ou répéter une portion de code. Par exemple on peut faire appel à un mécanisme décrit dans l'ISO 7816-3 [53] qui permet d'obtenir un délai supplémentaire au lecteur pour éviter un *timeout*. En T=0, cela revient à envoyer un octet `0x60` (Null) sur la couche transport. En Java Card, on peut obtenir cet effet en utilisant une méthode `apdu.waitExtension()`. Une chose intéressante à faire est d'encadrer une portion de code intéressant par deux envois d'octet NULL. Cependant, dans beaucoup de cartes à puce modernes, cette méthode Java Card est tout simplement désactivée. Une solution alternative consiste à utiliser `apdu.setOutgoing()` et `apdu.setOutgoingLength(...)` d'un côté du code à isoler et `apdu.sendBytes(...)` de l'autre. Ces méthodes permettent d'envoyer une réponse APDU en temps normal dans une utilisation classique de Java Card. Mais les utiliser de part et d'autre d'un code pertinent peut être utilisé pour isoler ce code car elles entourent le code pertinent de *glitches*.

Il est à noter que ce travail nécessite d'être très précis et très exact dans la mesure et éventuellement de pouvoir observer la couche transport, ce qui veut dire qu'il faut soit espionner le contact I/O à l'insu du lecteur (avec un oscilloscope par exemple cf 2.3.5), soit modifier le driver du CAD (si les sources sont disponibles) pour marquer les entrées/sorties de la manière la plus précise possible. Il faut aussi garder en tête que tout appel à des entrées/sorties va générer du bruit dans les mesures.

Les travaux de Sirrett [108] font état de plusieurs types d'attaques sur les cartes à puce utilisant le manque de notion de temps propre aux cartes à puce. Ainsi, il est possible de concevoir une attaque qui partage une carte à puce pour la télévision payante entre plusieurs personnes n'ayant pas nécessairement souscrit au service payant. Une contre-mesure à cette attaque consisterait à introduire la notion de temps dans la carte à puce (*timestamp*) et à contrôler le comportement du client en comparaison avec son comportement attendu. Du point de vue de la conception des applications, cela forcerait le développeur à considérer les performances respectives de la carte à puce et de la machine hôte associée. L'auteur présente une utilisation pratique des contre-mesures proposées en mesurant le temps de réponse *de bout en bout* notamment sur certaines solutions qui voient s'échanger plusieurs APDUs. Les résultats tendent à montrer la réalisabilité de la solution.

Les travaux de Vermoen et al. [119] montrent un type d'attaque que l'on peut employer en analysant la consommation d'une carte à puce. Si celle-ci exécute un code que l'on connaît,

on doit pouvoir analyser la courbe de consommation électrique à l'aide d'un oscilloscope pour retrouver les signatures de chaque bytecode et de chaque appel au coprocesseur cryptographique. Si par la suite, la même carte à puce exécute du code inconnu, on doit pouvoir reconstituer le code exécuté. Les résultats sont souvent très approximatifs, ce qui est du à des problèmes de répétabilité. On obtient des probabilités d'occurrence d'un bytecode à un moment donné. Les auteurs ne retrouvent pas tout à fait deux fois les mêmes courbes mais celles-ci peuvent se ressembler plus ou moins. Par recoupement les auteurs peuvent reconstituer le code "probable" qui s'est exécuté. Ces travaux typiques des attaques par DPA (Differential Power Analysis - attaque par analyse de la consommation) nécessitent une très grande précision.

2.3.13 Conclusion

Parmi les travaux présentés ici, seuls SCCB, Rehioui, Poll et Erdmann tentent de créer un outil de mesure de performance généralisé à toutes les plates-formes Java Card.

Les travaux de Rehioui se concentrent sur les cartes IBM JCOP, les tests présentés ne concernent que DES et les entrées sorties, il n'y a pas de tests menés sur les éléments de la machine virtuelle Java Card, les méthodes pour obtenir une mesure correcte sont douteuses.

Les travaux de Poll et al. sont ouverts sur plusieurs cartes et portent sur des fonctions cryptographiques et sur les versions supportées. La méthodologie de la mesure reste sommaire.

Les travaux de Erdmann sont concentrés sur les cartes Infineon. Les mesures de Erdmann ont été faites sans véritablement avoir en tête une véritable applet Java Card dans son utilisation classique. Les mesures ont été limitées dans le degré de précision qui a été apporté à ce qu'il fallait où non mesurer. Ainsi, les bytecodes qui sont factuellement mesurés pour les opérations arithmétiques sont arbitraires.

SCCB est resté incomplet : peu de mesures effectuées, toute la partie bytecode manque d'une approche générale et systématique. On pourrait aussi reprocher à SCCB un manque de portabilité, et une non gestion de l'éparpillement des résultats.

Les travaux de Chaumette et al. ne couvrent qu'une application très particulière, pour ainsi dire, très exceptionnelle dans le monde de la carte à puce : une application distribuée. Les travaux de Markantonakis ne couvrent que les entrées/sorties sur un nombre de cartes réduit, ceux de Papapanogiotou ne couvrent que les certificats, ceux de Guyot ne couvrent que les sessions, ceux de Castellà que les micro-paiements, ceux de Ficher se contentent de préciser quelques points ignorés par Erdmann. Les travaux de Tews concernent un protocole précis, et ne tentent pas de présenter une méthodologie de la mesure.

De manière générale, aucun travail n'a en profondeur vérifié que les mesures sont totalement indépendantes des outils de mesure. De plus aucun travail présenté ici n'est disponible publiquement sous la forme de code source, ou de binaire.

Cependant, on peut souligner la volonté, plus ou moins dissimulée des entreprises du domaine (IBM et Infineon en particulier, ici) de mesurer les performances de leurs produits. Il y a donc un véritable besoin, et il est sans doute également présent dans les entreprises qui sont clientes de ces fournisseurs. De même certains groupes de chercheurs liés à des universités font manifestement usage d'outils de mesure sans nécessairement l'afficher clairement (Tews). Il n'y a pas d'outils de référence général et propre à toute l'industrie.

On pourrait aussi souligner que ce manque de résultat officiel est peut être lié à un certain goût du secret. Des travaux de Chaumette et al. et de Vermoen et al. tendent à utiliser des mesures de grande précision pour attaquer les cartes en retrouvant les signatures énergétiques et temporelles de certains bytecodes et d'appels au coprocesseur cryptographique.

3

Méthodologie pour la mesure de Performance

Sommaire

3.1 Introduction

Le but de cette thèse est de fournir des programmes de mesure de performance qui soient représentatifs de l'utilisation qui est faite des cartes à puce. Il est donc important de fournir des **benchmarks d'applications** considérés comme représentatifs. Un problème auquel on serait confronté si on essayait de mesurer directement la performance d'applications serait les très courts temps de réaction que la plupart des commandes prennent usage quotidien d'une carte à puce. Ces temps de réaction peuvent être suffisamment courts pour que les bruits des systèmes utilisés pendant la mesure de performance puissent couvrir complètement les mesures elles mêmes. De plus on est très vite confronté à des problèmes de répétabilité de ces tests. En effet, avoir un temps de réaction très court pour un bruit trop important entraîne une grande dispersion des mesures et un flou sémantique quant aux temps mesurés.

Les travaux sur les mesures de performance qui ne s'intéressent qu'à une application en particulier sont soit excentrés par rapport aux domaines d'applications habituels des cartes à puce [15], ou bien considèrent des temps de réaction très longs [114], où encore manquent de proposer une véritable méthodologie générale pour pratiquer les mesures [13] [89]. Plus généralement, ces travaux servent avant tout à montrer la réalisabilité ou non d'une application, plus qu'à véritablement mesurer les performances générales des cartes à puce.

Un certain nombre de publications [102] [40] [38] [33] proposent d'utiliser des boucles contenant des portions de codes estimées comme étant dignes d'intérêt que l'on pourrait éventuellement isoler. Ces travaux sont systématiquement plus complets en termes de méthodologie, mais il reste cependant à analyser le degré de granularité des mesures et à replacer ces mesures isolées dans le contexte d'applications réalistes.

Concernant les métriques, nous nous intéresserons exclusivement à des applications réalistes dans des cas de figures où les opérations sont effectuées correctement et sans erreur. Par conséquent, les métriques intéressantes pour nous incluent le temps nécessaire à une seule exécution ou le taux d'instructions exécutées par seconde. Il se trouve que les applications réalistes ne fonctionnent pas généralement avec des boucles où une portion de code significatif est répétée un grand nombre de fois. Par conséquence, seul le temps d'exécution est retenu comme métrique. Il faut aussi garder à l'esprit que ces temps d'exécution doivent être utilisés pour fournir une note finale représentant les performances d'une carte à puce paticulière.

Nous allons présenter une méthodologie de mesure de performance. Cette méthodologie est basée sur l'isolement du temps d'exécution d'un sous ensemble du code. Nous allons d'abord présenter les principes généraux de l'isolement. Les bytecodes d'arithmétique vont ensuite servir d'exemple d'isolement du temps d'exécution d'un bytecode simple. Ceux-ci vont nous permettre entre autres de vérifier la correction de notre méthodologie en faisant varier les tailles de boucle. Nous allons ensuite étendre cette méthodologie à un maximum de bytecodes. Ensuite, nous tenterons d'utiliser une méthodologie similaire étendue aux appels des méthodes de l'API Java Card.

3.2 Principe général

3.2.1 Introduction

Dans les approches de la mesure de performance dans Java Card (cf 2.3), on peut distinguer plusieurs manières de procéder :

– en prenant une application *réaliste*, et en mesurant le temps pris par chaque échange d'APDU, éventuellement sur plusieurs cartes à puce, plusieurs CAD, et plusieurs machines

hôtes. Les résultats obtenus sont significatifs de l'application mesurée, mais limités à celle ci.
- En isolant le temps d'exécution de certains éléments pertinents de la performance de la carte. Cependant aucun travail ne tente de relier cette mesure à une application réaliste. Par contre les mesures peuvent être relativement complètes.

Le projet RNTL MESURE propose aux utilisateurs de mesurer les performances des cartes à puce en fournissant du code portable et sans que l'utilisateur n'ait besoin de connaître les détails de la carte à puce utilisée, comme le microcontrôleur utilisé ou la quantité et la qualité de la mémoire. Ces aspects sont par ailleurs gardés relativement secrets par les fabricants. Il ne peut donc pas s'agir purement d'une évaluation de performance par un modèle analytique ou par des simulations, mais de la mesure expérimentale de la performance. Si l'on voulait produire un modèle ou ne faire que des simulations, il faudrait développer un JCRE et une JCVM. Cela poserait déjà un problème pour distribuer les outils du fait de la licence Java Card. Il faudrait également produire un modèle de chacun des microcontrôleurs généralement utilisés dans la carte à puce ainsi que la partie OS propre à la carte et que tous les mécanismes de sécurité incoporés, comme les mécanismes de vérification de code. Les outils résultants seraient complexes et introduiraient par ailleurs un "flou" quant aux performances par manque de précision et d'exactitude [56]. La complexité et l'imprecision associées à de tels outils, ainsi que la difficulté, voire l'impossibilité, d'obtenir des spécifications précises de la part des fabricants, sont un frein au développement de tels modèles.

Nous avons aussi fait le choix de ne pas proposer un benchmark synthétique, qui pourrait refléter par exemple les performances brutes du microcontrôleur ou du coprocesseur cryptographique, ce qui n'aurait pas vraiment de sens vis-à-vis d'une application typique pour les cartes à puce. Les benchmarks synthétiques peuvent avoir un sens sur des ordinateurs personnels car ceux-ci ont de nombreuses utilisations possibles. Mais cela serait sans doute moins pertinent dans le cadre d'un système embarqué et en général mono-tâche comme une carte à puce.

La particularité des cartes à puce Java pour la mesure de temps d'exécution est que nous ne disposons pas d'une horloge interne accessible directement. Tout d'abord parce que les cartes à puce, pour la plupart, n'ont pas d'horloge interne, la seule horloge disponible provenant du terminal. Ensuite parce que l'API Java Card ne fournit aucune méthode d'accès à l'horloge. Cela rend difficile le fait de mesurer de manière strictement interne, tout en écrivant un outil de mesure qui soit portable.

Ce que l'on peut mesurer, c'est donc le temps pris entre l'envoi d'une commande APDU et la réception de la réponse APDU correspondante. La figure 3.1 illustre la problématique des temps écoulés pour la mesure de performance sur les cartes à puce. Une grande partie du temps mesuré peut être considéré comme n'étant pas pertinent.

Pour créer des benchmarks complets, il va donc nous falloir isoler le temps d'exécution des "éléments pertinents" de la performance d'une carte. Ensuite, nous pourrons relier ces temps mesurés à des applications utilisées couramment dans les domaines d'utilisation d'une carte à puce.

Pour mesurer la performance d'une Java Card, il nous faut mesurer :
- les performances de la machine virtuelle, c'est à dire, qu'on doit pouvoir mesurer le temps d'exécution de chaque bytecode,
- les performances de l'API Java Card, ce qui fait le plus souvent appel à des fonctionnalités natives (cryptographiques par exemple),
- les performances de la mémoire, suivant son type.

Les opérations arithmétiques sont un exemple simple d'éléments d'une machine virtuelle, qu'on peut vouloir mesurer. Comment mesurer la performance d'opérations arithmétiques dans

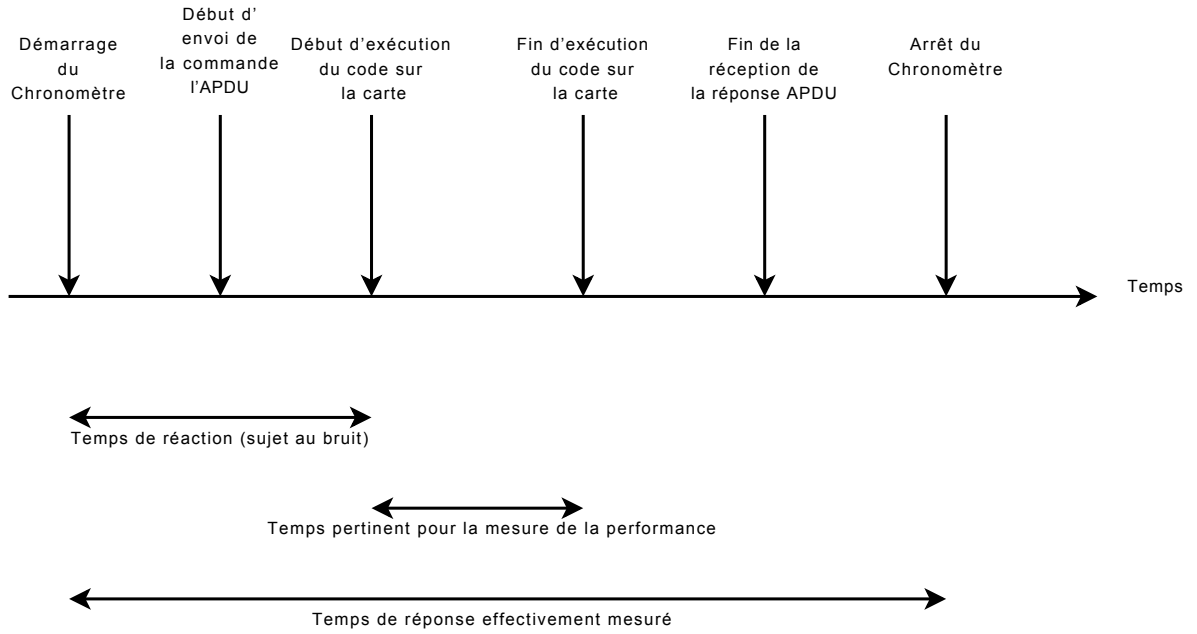


FIG. 3.1 – Temps de réponse mesurable

une carte à puce Java ? Nous avons développé des benchmarks sous Eclipse, en utilisant JDK 1.6 avec la JSR268 comme API.

L'outil de mesure de performance contient deux parties : une partie script et une partie applet. Tout est écrit en Java. Ces deux parties sont développées en parallèle : à chaque script correspond une applet. Tous les scripts héritent d'une classe `TemplateScript` et toutes les applets héritent d'une classe `TemplateApplet`. Ces deux classes couvrent l'ensemble des outils utiles pour les scripts ou les applets, et le fichier CAP correspondant au package `TemplateApplet` sera toujours le premier chargé sur la carte à puce et le dernier à être désinstallé une fois les tests effectués.

Chaque script contient une méthode `run()` qui va être exécutée à chaque fois qu'on va effectuer une mesure. À cet effet, c'est cette méthode qui va démarrer un chronomètre, lancer la commande APDU, réceptionner la réponse APDU et arrêter le chronomètre.

Depuis le JDK 1.5, l'API Java standard propose une méthode `System.nanotime()` pour obtenir un temps par rapport à un repère arbitraire (dans le passé ou le futur - d'où des temps négatifs parfois) qui correspond à une horloge système. Ce temps est codé sur un `long`, ce qui nous donne 2^{63} nano secondes (soit 292 ans) comme temps maximal d'écart entre deux dates mesurées avec cette méthode. Ce temps a une précision d'une nano seconde, ce qui ne veut pas dire qu'il y a une exactitude d'une nano seconde.

L'applet hérite de la classe `TemplateApplet`. À ce titre elle doit implémenter des méthodes `setUp()`, `run()` et `cleanUp()`. `TemplateApplet`, qui implémente la méthode `process(APDU apdu)` et qui va recevoir toutes les commandes APDU destinées à notre applet. Elle vérifie l'octet `INS` de la commande APDU qu'elle reçoit, et suivant la valeur de celle-ci, elle va appeler les méthodes implémentées par l'applet.

- La méthode `setUp()` effectue des allocations mémoire qui sont nécessaires pour certains tests.
- La méthode `run()` est utilisée pour lancer les tests qui vont être mesurés.
- La méthode `cleanUp()` est utilisée après le test pour nettoyer la mémoire.

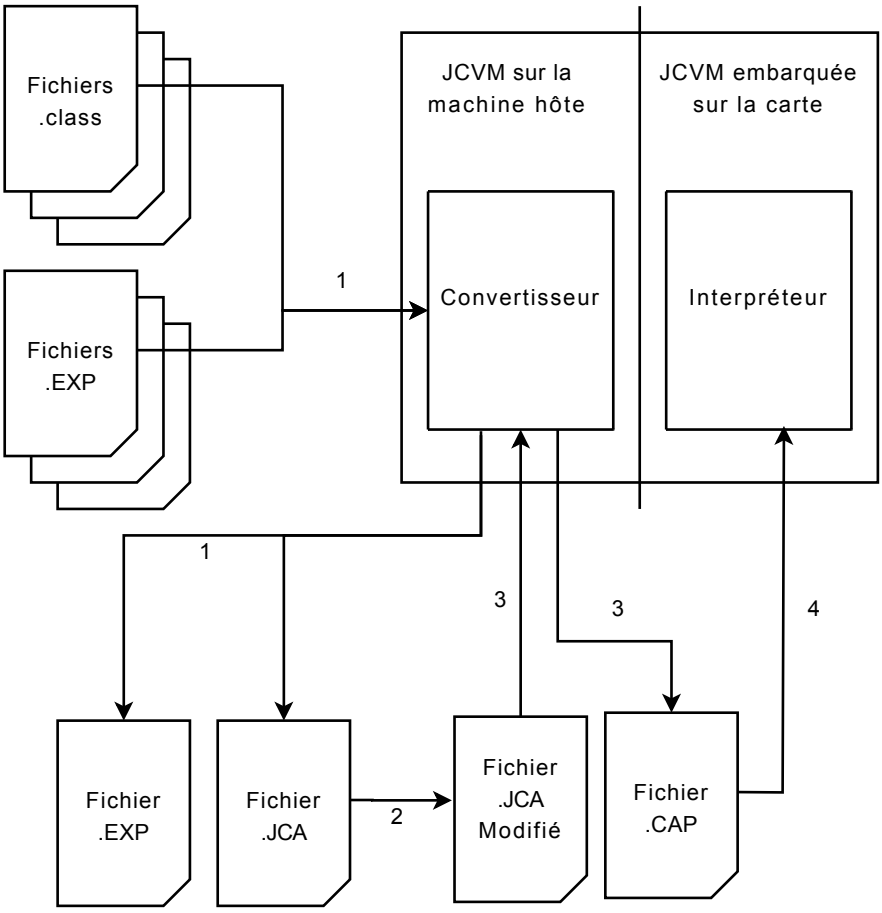


FIG. 3.2 – La conversion modifiée pour introduire des bytecodes

L'applet testée est capable de reconnaître tous les cas de tests et de lancer un test particulier avec sa méthode `run()`. La figure 3.2 illustre les nouvelles étapes pour convertir une applet. L'environnement Eclipse utilisé permet d'utiliser les outils de conversion des applet de Sun MicroSystems, ce qui permet de convertir une applet Java en fichier JCA dans un premier temps (étape 1). On va pouvoir utiliser un script pour modifier ce fichier JCA de manière à y insérer les bytecodes que nous souhaitons tester (étape 2). Une troisième étape permet de convertir le fichier JCA en fichier CAP qui va pouvoir ensuite être installé sur la carte à puce (étape 4).

3.2.2 Isoler le temps d'exécution d'un bytecode

Notre environnement Eclipse intègre l'outil Converter de Sun Microsystems, que l'on peut utiliser pour générer des fichiers CAP (Card Application). On peut aussi utiliser cet outil en plusieurs étapes. Lors d'une première étape, on va utiliser l'outil pour générer un fichier JCA (Java Card Assembly). Ce fichier représente le bytecode de l'applet sous une forme compréhensible pour un humain. Ensuite on peut convertir ce fichier JCA en fichier CAP. Si on veut mesurer la performance des bytecodes, il nous faut prendre en compte le fichier JCA produit, le modifier éventuellement pour utiliser les bytecodes qui nous intéressent, puis générer le fichier CAP que l'on va utiliser pour faire nos mesures.

Mesurer le temps d'exécution d'un bytecode dans une plateforme Java Card nécessite de trouver un moyen d'obtenir un résultat qui soit suffisamment précis. Le résultat doit refléter le

```
public void process(APDU apdu) throws ISOException {
  byte[] buffer = apdu.getBuffer();
  if (selectingApplet()) return;

  if (buffer[ISO7816.OFFSET_CLA] != CLA)
    ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

  switch (buffer[ISO7816.OFFSET_INS]) {
  case INS1 :
    run(apdu);
    break;

  default :
    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
  }
}
```

FIG. 3.3 – Extrait d'une classe Java

temps d'exécution isolé. Les mesures effectuées doivent prendre en compte le temps qui commence à s'écouler dès le démarrage du chronomètre, et l'exécution du bytecode en question. Du temps s'écoule aussi entre la fin de l'exécution du bytecode et l'arrêt du chronomètre. Ce temps parasite notre mesure et il est non prédictible.

L'appel de l'exécution du bytecode doit traverser plusieurs couches logicielles et matérielles avant que le bytecode ne s'exécute. Tout d'abord, l'outil de mesure sur la machine hôte utilise le langage Java. Comme Java utilise une machine virtuelle, celle ci va avoir un impact sur les performances notamment entre le démarrage du chronomètre et l'envoi de la commande APDU, et vice versa entre la réception de la réponse APDU (ce qui donne lieu à une allocation mémoire dans la machine virtuelle) et l'arrêt du chronomètre. La machine virtuelle Java sur la machine hôte va communiquer avec un service (comme le démon PCSC sous Linux) qui va prendre en charge la commande APDU et la transmettre au driver du CAD. On doit traverser aussi une couche matérielle, en utilisant le port PS/2 ou USB pour communiquer avec le CAD. Il y a un temps de transmission qui est non négligeable. Du côté de la carte à puce, les commandes APDU doivent passer par des couches matérielle (le microcontrôleur) et logicielles (le système d'exploitation, l'API Java Card). Et notre programme de test doit analyser l'APDU reçue avant d'exécuter le bytecode, ne serait-ce que pour vérifier que l'instruction (`INS`) est correcte et consiste bien à lancer un `run()`. Une fois `run()` appelé, il faut aussi prendre en compte tous les bytecodes, qu'on n'essaie pas de mesurer, mais qui vont être exécutés. Par exemple pour faire une opération simple comme une addition sur des entiers de type `short`, il nous faut au moins empiler deux de ces entiers. De plus, puisque nous voulons fournir un programme qui soit portable, nous supposons qu'il n'y a pas d'optimisation de la machine hôte pour en faire une machine dédiée aux mesures. Ainsi le niveau de priorité de notre processus de mesure est laissé inchangé.

Pour minimiser les effets de ces interférences, il nous faut isoler le temps d'exécution du bytecode qui nous intéresse plus particulièrement, tout en s'assurant que le temps d'exécution du bytecode en question est suffisamment important par rapport à la mesure.

Si on veut maximiser le temps d'exécution du bytecode sur lequel on se concentre, il nous

```
.method public process(Ljavacard/framework/APDU;)V 7 {
        .stack 2;
        .locals 1;

        .descriptor     Ljavacard/framework/APDU;       0.10;

L0:     aload_1;
        invokevirtual 4;// getBuffer()[B
        astore_2;
        aload_0;
        invokevirtual 5;// selectingApplet()Z
        ifeq L2;
L1:     return;
L2:     aload_2;
        sconst_0;
        baload;
        bspush -128;
        if_scmpeq L4;
L3:     sspush 28160;
        invokestatic 6;// javacard/framework/ISOException.throwIt(S)V
L4:     aload_2;
        sconst_1;
        baload;
        stableswitch L6 16 16 L5;
L5:     aload_0;
        aload_1;
        invokespecial 8; // SampleTestApplet.run(Ljavacard/framework/APDU;)V
        goto L7;
L6:     sspush 27904;
        invokestatic 6;// javacard/framework/ISOException.throwIt(S)V
L7:     return;
}
```

FIG. 3.4 – Extrait d'un fichier JCA

Applet	Cas de test

`process() {` (pseudocode)
`  i = 0`
`  While i <= L`
`  DO {`
`  run()`
`  i = i+1`
`  }`
`}`
	`run() {` (bytecodes)
	`  op`$_1$
	`  op`$_2$
	`  ⋮`
	`  op`$_n$
	`  op`$_0$
	`}`

TAB. 3.1 – Applet de mesure pour isoler le bytecode op_0

faut programmer l'applet avec une structure de boucle qui peut avoir une large borne supérieure, afin d'exécuter le bytecode un grand nombre de fois.

D'autre part, pour pouvoir complètement isoler le temps d'exécution de notre bytecode, il nous faut estimer le temps pris par tous les parasites. Entre autres, il nous faudra retrouver le temps d'exécution de tous les bytecodes auxiliaires qui nous sont utiles pour exécuter le bytecode qui nous intéresse. Cela veut donc dire qu'il y a un lien de dépendance entre les mesures de bytecodes. Par exemple, si on veut mesurer le temps d'exécution de `sadd`, on doit pouvoir mesurer et isoler le temps d'exécution d'un bytecode qui met un entier de type `short` sur la pile, comme `sspush` ou `sload`. Ensuite, pour obtenir le temps d'exécution isolé, il nous faut déduire le temps de la boucle à vide, du bruit et des bytecodes auxiliaires de la mesure contenant notre bytecode.

La figure 3.1 présente ce qui est exécuté dans un appel à la méthode `run()`. Les bytecodes auxiliaires et le bytecode qui nous intéresse sont tous exécutés dans la méthode `run()`. Cela nous garantit la libération de la pile après chaque appel à `run()`. C'est à dire que de la mémoire sera disponible pour les tests ultérieurs.

Dans la figure 3.1 :

- L représente la borne supérieure. On va utiliser l'octet `P2` de la commande APDU pour coder la taille de la boucle : $L = \texttt{P2}^2$. Cette méthode est simple et couvre un nombre assez grand de valeurs de boucle possibles : $0 \leq L \leq 32761 = 181^2$. La taille maximale de la boucle (32761) est ainsi proche de la plus grande valeur possible pour un entier signé codé sur deux octets (type `short`), c'est à dire $2^{15} = 32768$.
- op_0 représente le bytecode qui nous intéresse et que nous essayons d'isoler,
- op_i pour $i \in [1..n]$ sont les bytecodes auxiliaires nécessaires à l'exécution du bytecode op_0.

Pour calculer le temps d'exécution isolé de op_0, il nous faut faire le calcul suivant :

$$\overline{M(op_0)} = \frac{\overline{m_L(op_0)} - \overline{m_L(Bouclevide)}}{L} - \sum_{i=1}^{n} \overline{M(op_i)}$$

Où :

- $\overline{M(op_i)}$ est la moyenne du temps d'exécution isolé du bytecode op_i.
- $\overline{m_L(op_i)}$ est la moyenne du temps d'exécution du bytecode op_i, en incluant les interférences venant d'autres opérations effectuées durant la mesure, à la fois sur la carte et sur le terminal. Ces mesures ont été effectuées avec une taille de boucle L. Ces autres opérations représentent par exemple les bytecodes auxiliaires nécessaires à l'exécution du bytecode qui nous intéresse, ou des opérations spécifiques au système d'exploitation ou encore à la

machine virtuelle. Cette moyenne doit être calculée sur un nombre suffisant de tests. C'est une valeur que l'on peut mesurer expérimentalement.

– *Bouclevide* représente l'exécution d'un cas ou la méthode `run()` ne fait rien.

Cette formule implique qu'avant de calculer $\overline{M(op_0)}$, il nous faut calculer $\overline{M(op_i)}$ pour $i \in [1..n]$.

3.3 Bytecodes d'arithmétiques

La mesure du temps d'exécution d'un bytecode simple, comme une opération arithmétique, sur une carte à puce, est non triviale (il s'agit de la mesure). En effet, le temps d'exécution d'une opération arithmétique est en général assez court (12 cycles d'horloge pour une addition, 48 cycles pour une division [4]), en comparaison avec le bruit généré par l'envoi et la réception d'APDUs. La solution générale proposée ci dessus pourrait ne pas donner des résultats satisfaisants dans leurs cas.

Plus précisément, le fait d'utiliser une large taille de boucle L entraîne un certain degré de confiance dans les résultats, cela implique de faire de longues séances de mesures, ce qui n'est pas très pratique pour quelque chose comme la mesure d'une opération simple. D'un autre côté, si l'on fait des tests avec une taille de boucle relativement faible, on pourrait se retrouver avec des cas où $\overline{m(op_i)} < \overline{m(Emptyloop)}$. En effet, une taille de boucle trop petite associée au faible temps consacré à l'exécution d'opérations arithmétiques pourrait nous induire en erreur en nous faisant calculer un temps de performance isolé négatif (ou juste très éloigné de la valeur véritable). Cela pourrait être le cas par exemple si les mesures pour la boucle à vide sont effectuées pendant une période de charge système soudaine au niveau du terminal.

Et par conséquence, cela pourrait affecter les moyennes des temps d'exécution isolés. Pour minimiser ces situations indésirables, notre solution consiste en la répétition du bytecode mesuré un nombre de fois arbitrairement grand pour chaque appel de la méthode `run()`.

Certaines cartes à puce pourraient effectuer des contre mesures destinées à renforcer la sécurité qui vont dégrader la performance de ce type de test (cf [15]), si l'on exécute plusieurs fois à la suite des bytecodes similaires. Dans ce cas, notre solution générale marcherait encore, mais il nous faudrait juste utiliser une taille de boucle assez large, ce qui rallongerait la durée des mesures.

Cependant, dans le cas des opérations arithmétiques, augmenter le nombre d'exécutions d'un bytecode op_0 (pour k exécutions par exemple) ne nécessite pas forcement de devoir exécuter k fois tous les bytecodes $op_1 ... op_n$. Il s'agit d'opérations de manipulation de la pile (`sspush`, `sconst`, `sload`). Cela est dû au fait que les opérations arithmétiques se terminent en empilant le résultat de l'opération sur la pile. Il est donc possible d'optimiser le temps passé à mesurer. Quand on mesure une opération arithmétique qui nécessite deux opérandes, comme `sadd`, la méthode `run()` contiendra k occurrences de `sadd` précédées de $k+1$ occurrences de `sspush` (au lieu de 2 `sspush` et un `sadd` - cf figure 3.2.

Le calcul du temps d'exécution moyen isolé pour une opération arithmétique binaire op_0, en prenant en compte les k exécutions de op_0 est présentée comme suit :

$$\overline{M(op_0)} = \frac{\overline{m_L(op_0)} - \overline{m_L(Bouclevide)}}{L \times k} - (k+1) \times \overline{M(\texttt{sspush num})}$$

Où :

$$\overline{M(\texttt{sspush num})} = \frac{\overline{m_L(\texttt{sspush num}))} - \overline{m_L(Emptyloop)}}{L \times k}$$

`run` méthode pour isoler op_0	`run` méthode pour isoler `sspush`

`run(){`
`  sspush num`
`  (k)` { `sspush num` / op_0
`}`
	`run(){`
	`  sspush num`
	`  (k) {sspush num`
	`}`

TAB. 3.2 – Méthodes `run` pour les opérations arithmétiques binaires

3.4 Quelques résultats

3.4.1 Performance arithmétique

Nous avons évalué les performances arithmétiques de trois cartes utilisant des plates-formes Java Card 2.1 que l'on a appelé respectivement 3060, 4045, 2046. Conformément aux obligations du projet MESURE, nous ne révélons pas les origines de ces cartes à puce.

Cependant, les cartes 3060 et 2046 ont été conçues respectivement en 2006 et en 2004 par la même compagnie, alors que la carte 4045 a été conçue en 2004 par une autre compagnie. Nous avons utilisé nos benchmarks pour mesurer le temps d'exécution des opérations arithmétiques sur ces trois cartes à puce. Les résultats sont présentés dans la figure 3.5. Ils montrent les temps d'exécution moyens isolés pour quelques opérations arithmétiques sur des entiers signés de 16-bits (de type `short`).

On peut constater que sur chacune des cartes à puce, les temps d'exécution de quelques opérations sont similaires les uns aux autres. Ainsi, le temps d'exécution d'un `sadd` sur une carte à puce donnée est approximativement le même que les temps d'exécution de `ssub`, `sot`, `sand`, `sxor` sur la même carte à puce. Tout ceci peut sembler tout à fait normal pour des opérations assez similaires : ce sont des opérations binaires dont le calcul correspondant (hors opérations sur la pile de la JCVM) devrait prendre un seul cycle machine (soit 12 cycles d'horloge sur une architecture de type Intel 8051). On peut aussi observer que l'opération `sneg` (qui change le signe d'un entier) est plus rapide que les autres opérations quelque soit la carte à puce. On peut comprendre cela en considérant que c'est la seule opération qui ne nécessite qu'un seul opérande sur la pile.

Il est à noter que nous avons effectué des mesures avec différentes valeurs pour chacune de ces opérations sans constater de changement notable. Ainsi, sur chacune de ces cartes à puce diviser 32767 (la plus grande valeur positive pour nos entiers) ou 0 prend exactement le même temps. Autrement dit, on effectue toujours la division sans tester la valeur du dividende.

Avec ce test, nous avons aussi pu nous poser des questions quant aux implémentations des opérations arithmétiques dans les trois cartes à puce Java. Par exemple, on peut observer que pour les cartes à puce 3060 et 4045, le bytecode `sshl` (déplacement de bits vers la gauche) a pratiquement le même temps d'exécution que les bytecodes `smul` (multiplication) et `sshr` (déplacement de bits vers la droite). Pour la carte à puce 2046, le temps d'exécution de `sshl` est plus proche de celui de `sdiv` ou `srem`. De cette observation, on peut déduire que l'implémentation du `sshl` est différent sur 2046.

En conclusion, en comparant les performances de trois plates-formes Java Card, on peut clairement les distinguer. Sur les opérations arithmétiques, la carte 2046 est un peu moins performante que la carte 3060, mais elles sont toutes deux bien plus performantes que la carte 4045. Toutes ces mesures sont faites en "boite noire", c'est à dire qu'on ne connaît pas très exactement le contenu des cartes à puce. C'est à dire qu'on ne connaît pas les microprocesseurs embarqués

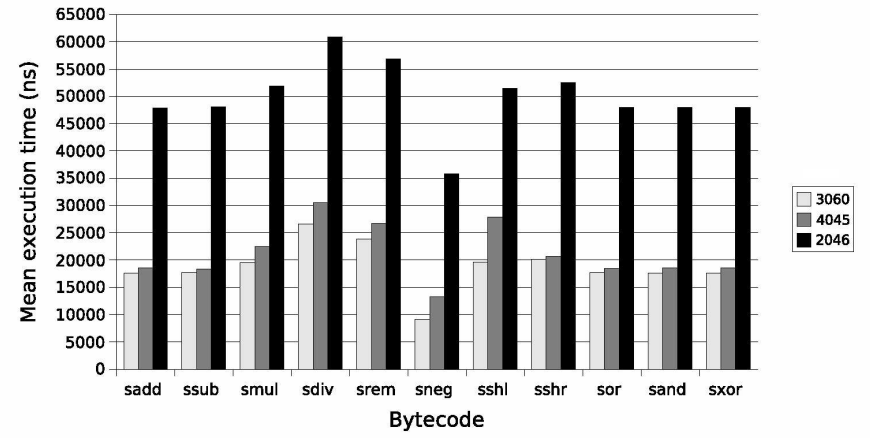


FIG. 3.5 – Performance arithmétiques des trois cartes à puce

dans ces cartes à puce. Mais pour des bytecodes aussi simples que ceux-ci, l'implémentation n'est sans doute pas plus compliquée que la lecture du bytecode et des opérandes, l'exécution du code machine correspondant, et le placement du résultat sur la pile. Les implémentations sont sans doute très similaires d'une carte à l'autre. Donc ce qu'on mesure est influencé par le matériel utilisé ici.

3.4.2 Linéarité des résultats

Avec ce système d'isolation du temps d'exécution d'un bytecode, on s'attend à ce que la moyenne des mesures isolées soit plus où moins la même quelque soit la taille de la boucle. En fait les bruits de mesures devraient être moins perceptibles pour une taille de boucle forte puisque la quantité d'exécution du bytecode qui nous intéresse est plus grande pour les grande tailles de boucles et que le temps de mesure bruitée doit être relativement faible par rapport au temps de mesure non bruité contenant le bytecode.

On s'attend donc à ce que les moyennes soient stables pour des tailles de boucle suffisamment grandes. Nous avons donc vérifié la linéarité des résultats isolés sur deux cartes à puce nommées 3060 et 4045. La figure 3.6 nous montre les temps moyens d'exécution pour un bytecode `sadd` isolé avec 100 mesures pour différentes tailles de boucles. Durant ce test, nous avons utilisé l'octet `P2` des commandes APDUs pour changer la taille de la boucle.

Comme nous pouvons le constater, les mesures tendent à atteindre un certain degré de stabilité au fur et à mesure que la taille de boucle grandit, même si les résultats obtenus pour les deux cartes à puce sont différents en termes de valeurs. Nous pouvons aussi observer que les deux courbes montrent les mêmes signes de stabilisation sur les différentes tailles de boucle. Cela signifie que cette technique d'isolation du temps d'exécution fonctionne quelque soit la taille de la boucle, mais que des bruits dans la mesure doivent perturber les mesures avec une faible taille de boucle de manière plus importante.

De manière générale, la stabilisation du temps d'exécution d'un bytecode isolé semble dépendre de facteurs tels que le CAD employé, le pilote de celui-ci et le système d'exploitation. En conséquence, la taille de boucle nécessaire pour obtenir un résultat précis et exact doit dépendre de l'environnement de test. Cela entraîne un besoin en calibration des mesures en fonction de l'environnement d'exécution avant d'effectuer les mesures.

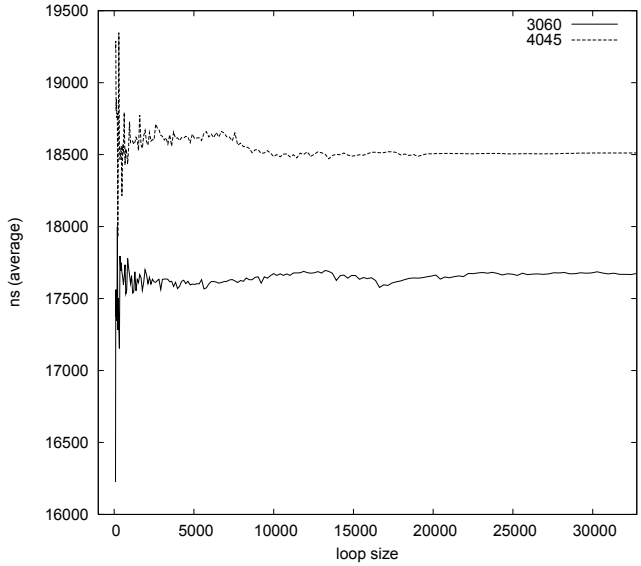


FIG. 3.6 – Mesure de $\overline{M(\texttt{sadd})}$ sur deux cartes à puce

3.5 Extension des mesures à d'autres bytecodes

Puisqu'on peut isoler le temps d'exécution des bytecodes, arithmétiques, on peut aussi utiliser ces méthodes pour d'autres bytecodes.

De la même manière que précédemment, on va utiliser des bytecodes auxiliaires pour exécuter le bytecode qui nous intéresse.

- Certains bytecodes sont élémentaires. Ceux qui manipulent la pile, par exemple, ne dépendent pas nécessairement d'autres bytecodes pour s'exécuter. `sspush` et `scont` permettent de mettre des nombres de type `short` sur la pile. Ils sont donc mesurables directement contre une boucle à vide pour pouvoir isoler leurs performances. D'autres, comme par exemple `pop`, demandent la présence d'un ou plusieurs entiers sur la pile, et leur mesure est donc dépendante des précédents.
- Les bytecodes conditionnels de type `if` ou `slookupswitch` nécessitent d'être mesurés plusieurs fois suivant les valeurs qui leurs sont communiquées. Les `if` se rendent sur une étiquette si la valeur sur la pile remplit une certaine condition mais continuent à la suite si cette condition n'est pas satisfaite. On se propose donc d'exécuter le même code (insignifiant) dans chaque branche du `if` avant de terminer l'appel à `run()`.
- Certains bytecodes échappent totalement à notre technique. En effet, tout, jusqu'ici, est basé sur le fait qu'on peut modifier la pile une fois la méthode `run()` appelée et qu'à la fin de l'exécution de celle-ci, on va pouvoir la rappeler sans se soucier de gérer la pile. La pile peut, par exemple, ne pas être vide après l'appel à `run()`. Les bytecodes de type `invoke` (les appels d'une méthode qui peut prendre plusieurs formes), `return` (retour de la méthode avec ou sans valeur de retour) échappent à ce type d'isolement. On ne pourra pas isoler un bytecode `invoke` sans inclure le temps d'exécution d'un `return` en même temps. Mais mesurer un appel à une méthode sans le retour de celle-ci n'aurait pas grand sens de toutes façons. Nous pouvons donc mesurer les temps d'exécution de ces bytecodes ensemble, ce qui est isolable avec notre méthode.
- Certains bytecodes ne sont tout simplement pas utilisés, parfois pas générés par le compilateur (comme `nop` - un bytecode qui ne fait rien par exemple).

Instruction set					
Opcode	**Mnemonic**	**Reference loop**	**Opcode**	**Mnemonic**	**Reference loop**
0	nop	empty loop	117,118	<t>lookupswitch	1 <t>load | 1 goto
1	aconst_null	empty loop	119-121	<t>return	
2-8	sconst_<n>	empty loop	122	return	
9-15	iconst_<n>	empty loop	123-126	getstatic_<t>	empty loop
16-20	<t1><t2>push	empty loop	127-130	putstatic_<t>	1 <t>load
21	<t>load	empty loop	131-134	getfield_<t>	1 aload
24-35	<t>load_<n>	empty loop	135-138	putfield_<t>	1 aload | 1 <t>load
36-39	<t>aload	1 aload | 1 sload	139	invokevirtual	
40-42	<t>store	1 <t>load	140	invokespecial	
43-54	<t>store_<n>	1 <t>load	141	invokestatic	
55-58	<t>astore	1 aload | 1 sload | 1 <t>load	142	invokeinterface	
59	pop	1 sload	143	new	empty loop
60	pop2	2 sload	144	newarray	1 sload
61	dup	1 sload	145	anewarray	1 sload
62	dup2	2 sload	146	arraylength	1 aload
63	dup_x	1 bspush | n sload	147	athrow	
64	swap_x	1 bspush | n sload	148	checkcast	1 aload
65-88	<t><arithmetic_operation>	2 <t>load	149	instanceof	1 aload
89,90	<t>inc	empty loop	150,151	<t>inc_w	empty loop
91-94	<t1>2<t2>	1 <t1>load	152-159	if<cond>_w	1 sload
95	icmp	2 iload	160-167	if_<t>cmp<cond>_w	2 <t>load
96-103	if<cond>	1 sload	168	goto_w	nop
104-111	if_<t>cmp<cond>	2 <t>load	169-172	getfield_<t>_w	1 aload
112	goto	nop	173-176	getfield_<t>_this	empty loop
113	jsr		177-180	putfield_<t>_w	1 aload | 1 <t>load
114	ret		181-184	putfield_<t>_this	1 <t>load
115,116	<t>tableswitch	1 <t>load | 1 goto			

FIG. 3.7 – Jeu d'instructions

Nous pouvons pour tous ces bytecodes faire une analyse de dépendance. Les résultats de cette analyse sont présentés dans la figure 3.7. Nous utilisons quelques abréviations suivantes pour qualifier les bytecodes :

$$\begin{aligned} t &::= a|b|i|s \\ n &::= m\_1|0|1|2|3|4|5 \\ cond &::= eq|ne|gt|ge|lt|le \end{aligned}$$

Dans ce tableau :
- t représente l'ensemble des types utilisés : les objets (a), les `bytes` (b), les `integers` (i) et les `shorts` (s).
- n représente les constantes entières utilisées.
- *cond* représente les différentes conditions.

Pour la plupart de ces cas, les tests suivront le cas général présenté dans la section 3.2. Les cas grisés dans la figure 3.7 correspondent aux bytecodes qui peuvent être mesurés par paires, mais pas individuellement. Pour ceux ci, notre méthode ne fonctionne pas directement.

3.6 API

De la même manière que l'on mesure les temps d'exécution des bytecodes, on peut mesurer les temps d'exécution des entrées de l'API.

Ces temps doivent être isolés d'une manière proche de celle que l'on a utilisé précédemment. On va donc mesurer le temps pris par une carte à puce pour exécuter un appel à une méthode

```
// Classe OwnerPINTestApplet
public void run(byte[] apduBuffer) {
  dummy.dummyMethodShort(value,offset,length);
}
// Classe DummyClass (qui doit se trouver dans un autre package)
public void dummyMethodShort(byte[] array, short s, byte b) {}
```

FIG. 3.8 – Méthode de référence en Java

```
// Classe OwnerPINTestApplet
public void run(byte[] apduBuffer) {
  pin.check(value,offset,length);
}
```

FIG. 3.9 – Méthode pour appeler `check(...)` en Java

de l'API. Cet appel est exécuté en boucle de taille $L = \texttt{P2}^2$. Nous allons ensuite déduire de cette première série de mesure le temps pris par la même carte à puce pour effectuer, toujours dans une boucle de taille L, un appel à une méthode qui, cette fois ne fait rien. Il faut être assez précis quant aux bytecodes qui vont être exécutés pour être sûr d'exécuter les mêmes bytecodes dans les deux appels. Les appels de méthode de l'API que l'on veut mesurer ressemblent à des appels de méthode que l'utilisateur a programmé lui même, puisqu'on va utiliser `invokevirtual` pour appeler cette méthode. Pour que l'on puisse comparer l'appel d'une des méthodes de l'API a l'appel d'une méthode qui ne va rien faire, il va donc falloir programmer une méthode qui ne fait rien dans un autre package Java, afin de toujours faire appel à `invokevirtual`.

Sans aller jusqu'à la programmation en bytecodes de ces tests, on peut écrire des classes Java qui une fois compilées seront observables sous la forme de fichiers JCA. On peut prendre l'exemple de la vérification d'un code PIN (méthode `OwnerPIN.check(byte[] tableauDOctets, short décalage, byte longueur)`).

Les figures 3.8 et 3.9 présentent le code en Java des méthodes que l'on va appeler pour comparer le temps d'appel à `check(...)` et à une méthode `dummyMethodShort(...)` qui ne fait rien. Ces méthodes utilisent des valeurs `value`, `offset` et `length` qui sont initialisées en utilisant les méthodes `setUp()` de notre applet de test.

Les figures 3.10 et 3.11 représentent le bytecode généré par ces portions de code Java. Regarder le bytecode doit nous aider à retrouver ce qui est utilisé dans chacun de ces appels de méthode pour pouvoir les mettre en parallèle et trouver les différences.

Les différences entre les bytecodes générés dans la partie de référence et la partie contenant l'appel à `check(...)` sont donc les suivantes :

- La partie de référence contient une occurrence de `return` supplémentaire (il correspond au retour de la méthode `dummyMethodShort(...)` qui a remplacé l'appel à `check(...)`. On ne peut pas vraiment éliminer ce `return` de la mesure. Les mesures comprendront donc l'appel (`invokevirtual`) et le retour (`return`) de la méthode, alors qu'on n'a que l'appel pour `check(...)`.
- La partie de mesure de `check(...)` contient un `pop` supplémentaire qui a été généré au retour de l'appel à `check(...)`. On peut éliminer le temps de performance de ce `pop` ou encore le supprimer au niveau bytecode, une fois le fichier JCA généré.

Le temps d'exécution d'un appel à `check(...)` doit donc être de la forme :

```
// Classe OwnerPINTestApplet
.method public run([B)V 6 {
.stack 4;
.locals 0;

L0:getfield_a_this 0;
// this
invokestatic 38;
// dummy
getfield_a_this 0;
invokestatic 33;
// value
getfield_a_this 0;
invokestatic 39;
// offset
getfield_a_this 0;
invokestatic 41;
// length
invokevirtual 42;
// dummyMethodShort([BSB)V
return;
}

// Classe DummyClass
.method public dummyMethodShort([BSB)V 5 {
.stack 0;
.locals 0;

L0:return;
}
```

FIG. 3.10 – Bytecode généré pour la méthode de référence

```
method public run([B)V 6 {
.stack 4;
.locals 0;

L0:getfield_a_this 6;
// this (notre applet de mesure)
invokestatic 49;
// retourne pin
getfield_a_this 6;
invokestatic 33;
// retourne value
getfield_a_this 6;
invokestatic 39;
// retourne offset
getfield_a_this 6;
invokestatic 41;
// retourne length
invokevirtual 32;
// check([BSB)Z
pop;
return;
}
```

FIG. 3.11 – Bytecode généré pour `check(...)`

$$\overline{M(\texttt{check})} = \frac{\overline{m_L(\texttt{check})} - \overline{m_L(BoucleRef)}}{L} - \overline{M(\texttt{pop})} + \overline{M(\texttt{return})}$$

Dans cette équation :
- $\overline{M(\texttt{check})}$ est le temps moyen isolé d'un appel à `check(...)`.
- $\overline{m_L(\texttt{check})}$ est le temps moyen mesuré en utilisant notre applet qui effectue un `check(...)` dans une boucle de taille L.
- $\overline{m_L(BoucleRef)}$ est le temps moyen mesuré en utilisant notre applet qui effectue un appel à `dummyMethodShort(...)` dans une boucle de taille L
- $\overline{M(\texttt{pop})}$ est le temps d'exécution isolé d'un bytecode `pop`. Ce temps peut être mesuré en utilisant notre méthodologie générale.
- $\overline{M(\texttt{return})}$ est le temps isolé moyen pris pour exécuter un bytecode `return`.

Le temps pris par `return` n'étant pas déductible avec nos mesures, la valeur $\overline{M(\texttt{return})}$ restera donc une inconnue. On pourra accepter cela car `return` n'est "qu'un bytecode" en comparaison d'un appel relativement coûteux à une méthode comme `check(...)`. Il est vraisemblable que l'appel à `check(...)` fasse lui-même appel à une fonction native sur la carte à puce qui termine son exécution en faisant elle même appel à des mécanismes similaires au bytecode `return`. Ceci est dépendant de l'implémentation de la méthode `check(...)`, de `return` et plus généralement de la JCVM. Généralement, l'appel à une méthode de l'API est lancé par un bytecode `invoke` (comme le test de référence correspondant) et même si aucun bytecode `return` n'est appelé du côté du test de l'opération de l'API, on peut supposer que la valeur qui est éventuellement retournée l'est faite avec des mécanismes similaires à un appel à `sreturn`, par exemple pour retourner un `short`, ou par un `return` dans le cas général. Le temps correspondant à la lecture du bytecode `return` dans le test de référence est donc un temps compté en excès. On considère donc comme négligeable le temps de lecture de `return` par la machine virtuelle. Il faut rappeler qu'on ne cherche pas à mener des attaques contre les cartes à puce en utilisant des canaux cachés, ici (ce qui demanderait une grande précision et une grande exactitude). On essaie juste d'avoir une idée de la performance des cartes à puce pour donner des notes à ces cartes. On va donc considérer cette approximation comme acceptable.

3.7 Conclusion

Les méthodes présentées dans ce chapitre permettent d'isoler la performance de portions d'application Java Card avec une granularité de l'ordre du bytecode. De plus les résultats isolés semblent relativement constants en fonction de la taille de boucle, même si l'on peut constater un bruit gênant pour les tailles de boucles faibles. Cette méthode peut être également utilisée pour des appels de méthodes de l'API Java Card.

Cependant, il nous reste à proposer un cadre pour récupérer les résultats d'une manière présentable pour un utilisateur de benchmark, et surtout, il faut remettre ces mesures individuelles dans le contexte d'une application. Tous les benchmarks utilisés dans l'industrie présentent des résultats qui sont basés soit sur une donnée considérée représentative d'un calcul typique, comme les opérations flottantes (Mflops) ou un nombre d'instructions par seconde (Mips), mais surtout, les benchmarks récents (en particulier SPEC qui est un benchmark de référence sur PC) font usage de nombreuses applications générales développées dans un cadre précis et utilisées pour noter les performances de la plateforme par rapport à ces applications en particulier.

Les questions qui se posent alors sont : quelles applications sont représentatives des besoins usuels dans les différents domaines d'application des cartes à puce ? Comment retrouver l'usage

qui est fait de ces applications et comment relier cet usage aux données brutes qui viennent de nos mesures ? Comment extraire des notes de synthèse de ces informations ?

Il nous manque également un cadre pour présenter et analyser les résultats. À ce stade nous pouvons facilement faire un calcul d'écart type pour représenter la dispersion des valeurs mesurées, mais nous n'avons pas d'idée de l'exactitude de ces mesures qui sont potentiellement bruitées, comme le montre la figure 3.6. Cette dernière figure suggère aussi un besoin de calibration des mesures. S'il semble que les mesures pour une taille de boucle courte est bruitée, alors il faudra décider de quelle taille de boucle est appropriée pour obtenir des mesures pour lesquelles les effets du bruit se font moins sentir.

4

Les outils de MESURE

Sommaire

4.1 Introduction

Nous décrivons ici les outils développés pour mesurer le temps d'exécution des opérations élémentaires Java Card.

Le framework a été conçu pour réaliser les objectifs du projet MESURE, et est composé de plusieurs modules qui sont décrits ici (voir figure 4.1). L'objectif de la première étape est de trouver les paramètres optimaux à utiliser pour effectuer correctement les tests. Ceux-ci couvrent les opérations de la JCVM et les méthodes de l'API Java Card. Les résultats obtenus sont filtrés en éliminant les mesures non pertinentes.

La mesure des temps d'exécution est réalisée à l'envoi des commandes APDU du PC à la carte via le lecteur. Chaque test (run) est exécuté Y fois pour assurer la fiabilité des temps d'exécution collectés, et au sein de la méthode `run()` une boucle est exécutée L fois. L est codé sur l'octet `P2` des commandes APDUs envoyées à l'application s'exécutant sur la carte. La taille de la boucle sur la carte est $L = \texttt{P2}^2$.

4.2 Les modules

4.2.1 Le module Calibrate

Mesurer le temps d'exécution des différents bytecodes et entrées API peut s'avérer laborieux. D'un autre côté, il est nécessaire d'être suffisamment précis dans la mesure du temps.

Ce module Calibrate calcule des paramètres tels que le nombre de boucles selon une précision donnée. Pour des mesures fiables, nous comparons la valeur de la mesure avec l' écart-type. Pour cela, le module aura en entrée le rapport de la mesure moyenne et de l'écart-type ainsi qu'une valeur minimale, égale à 1 seconde, correspondant au temps des mesures par défaut. Avec ces valeurs d'entrée, des tests sont effectuées avec différentes tailles de boucle pour approcher la valeur idéale.

Les figures 4.2 et 4.3 font état respectivement de la moyenne et de l'écart type obtenus dans les mesures d'un test de référence en faisant varier les tailles de boucles. Ces mesures sont obtenues avec 100 valeurs mesurées pour chaque taille de boucle. On peut observer la nature linéaire de la moyenne. L'écart type, lui, n'est pas stable. Cependant, même s'il est instable, l'écart type atteint un maximum de 9601855 alors que la moyenne croit linéairement. On peut donc mettre en relation l'écart type et la moyenne.

Le coefficient de variation c est défini comme le ratio entre un écart type σ et une moyenne μ ($c = \sigma/\mu$). C'est une valeur qui nous donne une idée de la dispersion dans nos données. Il offre l'avantage de n'avoir pas de dimension. Avec la croissance de la moyenne en fonction de la taille de la boucle, ce nombre devient relativement faible avec une taille de boucle peu élevée.

Les figures 4.4 et 4.5 représentent les évolutions respectives du coefficient de variation et de son logarithme en fonction de la taille de la boucle. Le logarithme est donné d'abord pour montrer les résultats avec plus de clarté. Ensuite comme il est plus facilement observable, c'est lui qui sert éventuellement à l'utilisateur pour indiquer la précision. Les utilisateurs du projet peuvent donc modifier la précision des mesures en notant un indice qui leur convient (par test-erreur) et qui correspond au logarithme du coefficient de variation. Ces figures permettent de constater la décroissance du coefficient de variation en fonction de la taille de la boucle.

Si l'on veut optimiser les mesures de performances de manière à ce que :

1. l'ensemble des tests ne prenne pas trop de temps,
2. les mesures soient suffisamment précises,

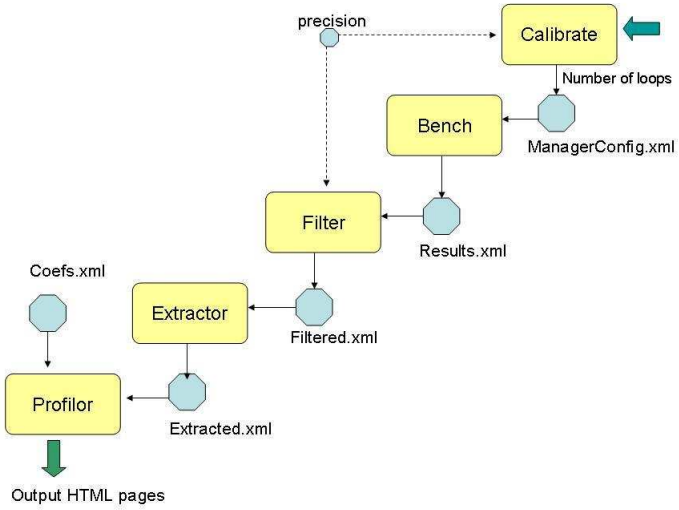


FIG. 4.1 – Les modules du framework de mesure

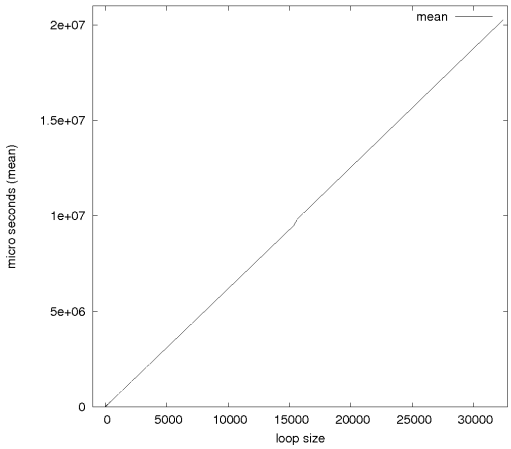


FIG. 4.2 – Moyenne d'une mesure d'un test de référence en fonction de la taille de la boucle

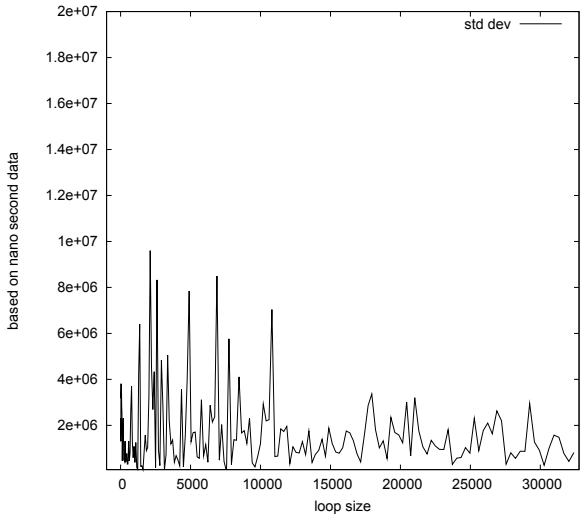


FIG. 4.3 – Écart type d'une mesure d'un test de référence en fonction de la taille de la boucle

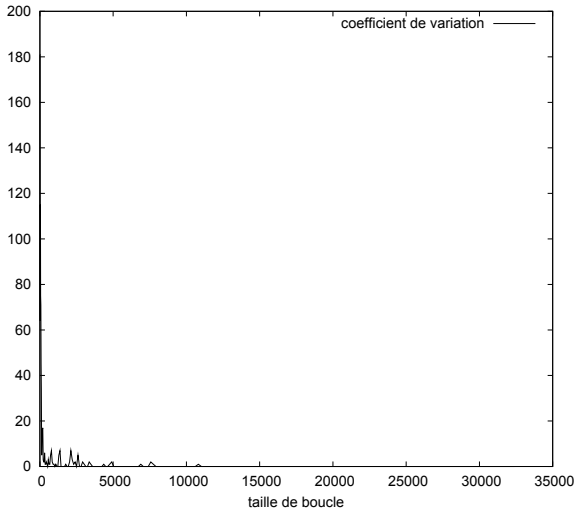


FIG. 4.4 – Évolutions du coefficient de variation en fonction de la taille de boucle

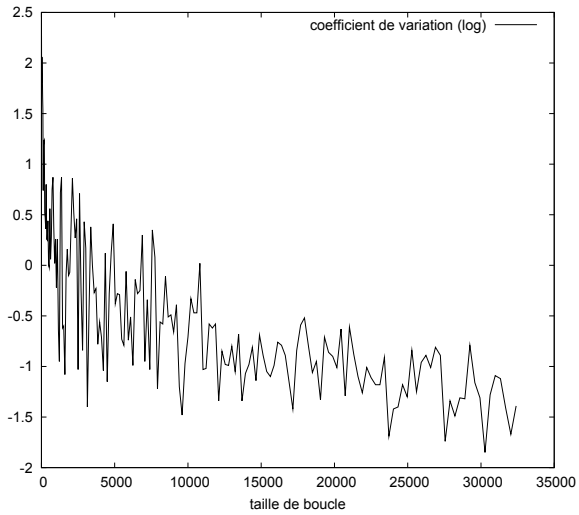


FIG. 4.5 – Évolutions du logarithme du coefficient de variation en fonction de la taille de boucle

3. la différence entre les temps mesurés d'un élément à isoler et les temps mesurés du test de référence correspondant soit suffisant,

on peut tenter de calibrer les tailles de boucles en utilisant le coefficient de variation, et on peut se fixer un temps minimal pour effectuer une mesure.

Pour calibrer les tests, on va utiliser le fait que la moyenne d'une mesure croit linéairement en fonction de la taille de la boucle, alors que le coefficient de variation globalement décroît en fonction de la taille boucle. On va procéder à une recherche dichotomique dans les mesures. Les conditions d'arrêt de la recherche sont :

1. un temps minimal de mesure (par défaut, une seconde),
2. un ratio maximum pour le coefficient de variation.

La calibration n'est pas nécessaire pour tous les tests. On peut regrouper les tests en familles (par exemple toutes les opérations arithmétiques, toutes les signatures par RSA, toutes les vérifications de code PIN). Ces opérations utilisant le même test de référence, seul celui-ci a besoin d'être calibré, avec quelques exceptions : les opérations de chiffrement asymétriques (RSA, ou avec un algorithme cryptographique à courbe elliptique) prennent en général beaucoup plus de temps que leurs tests de référence respectifs. Dans ce dernier cas, de nombreux tours de boucles ne sont donc pas nécessaires pour avoir une différence temporelle entre le temps mesuré pour le test d'API et le test de référence correspondant. On va donc se limiter à garantir que les mesures pour le test de référence ne sont pas trop dispersées (en utilisant le coefficient de variation seul). De cette manière, on se limite à un test à calibrer par famille de tests. Pour les autres tests de la même famille, on utilise le même calibrage que pour le test calibré de la famille.

La figure 4.6 représente un calibrage sur une mesure de référence pour la mesure du bytecode `pop`.

4.2.2 Le module Bench

Le terminal (PC) charge les applets sur la carte, gère le lecteur de cartes et collecte les résultats en envoyant des commandes APDUs à la carte et en recevant les réponses correspondantes. Comme il n'existe pas de timer sur la carte, le code qui effectue la mesure est exécuté sur la carte mais est mesuré sur le PC avec l' horloge de ce dernier.

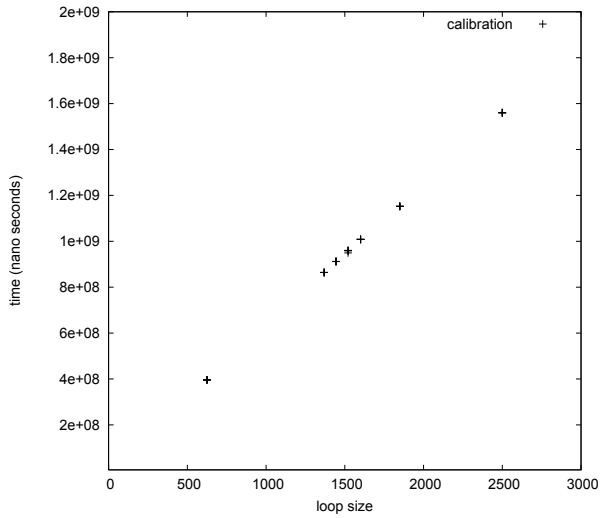


FIG. 4.6 – Exemple de calibration avec un test de référence pour le bytecode `pop` : on fait 30 mesures pour chaque taille de boucle pour obtenir une mesure de plus de 1s.

Pour un nombre de cycles défini par le module Calibrate, le module Bench effectue les mesures en calculant le temps d'exécution moyen pour : les bytecodes de la VM, les méthodes de l'API, les mécanismes JCRE (tels que le mécanisme de transactions).

4.2.3 Le module Filter

Le module Filter permet de détecter éventuellement les problèmes d'incohérence dans les temps mesurés. L'idée est que ces bruits sont dus à des bruits logiciels survenus pendant la mesure et qu'on doit pouvoir les détecter et les éliminer.

Des erreurs expérimentales peuvent générer du bruit dans les mesures brutes. Ce bruit provoque de l'imprécision dans les valeurs mesurées, rendant difficile l'interprétation des résultats. Dans le contexte des cartes à puce, le bruit est généré lors de l'envoi d'une commande de l'application de mesure vers l'applet. L'application de mesure démarre un chronomètre puis lance une commande. La commande passe d'une entité logicielle ou matérielle à une autre et parcourt dans un sens de la machine hôte, le lecteur de cartes et la carte à puce. La réponse APDU parcourt le chemin inverse pour parvenir à l'application de mesure qui peut alors arrêter le chronomètre. La figure 4.7 illustre sommairement les interactions qui doivent survenir entre différentes entités qui interviennent à partir du moment où le programme démarre un chronomètre et jusqu'au moment où il l'arrête. Plusieurs niveaux d'encapsulation sont suggérés par ce schéma, et par le système de communication en piles de protocoles, et pour chaque opération effectuée pendant l'empilage et le dépliage de données du côté émetteur et du côté receveur.

Dans le cas idéal, les mesures obtenues sont normalement distribuées. Dans un tel cas et pour n mesures avec $n \geq 30$ toutes indépendantes les unes des autres, la moyenne $\overline{x}$ des $x_i, i \in [1..n]$ est la meilleure approximation de la valeur réelle [74]. Dans un tel cas, on peut déterminer un intervalle de confiance autour d'une moyenne $\overline{x}$ à l'intérieur duquel on peut garantir une certaine probabilité de retrouver une mesure influencée par un certain paramètre. On pourra parler par exemple d'un intervalle de confiance à 95%. L'outil Filter est utilisé pour éliminer les valeurs qui sont en dehors d'un intervalle de confiance donné.

Un problème majeur est survenu pendant l'élaboration du module Filter. Il s'agit d'un prob-

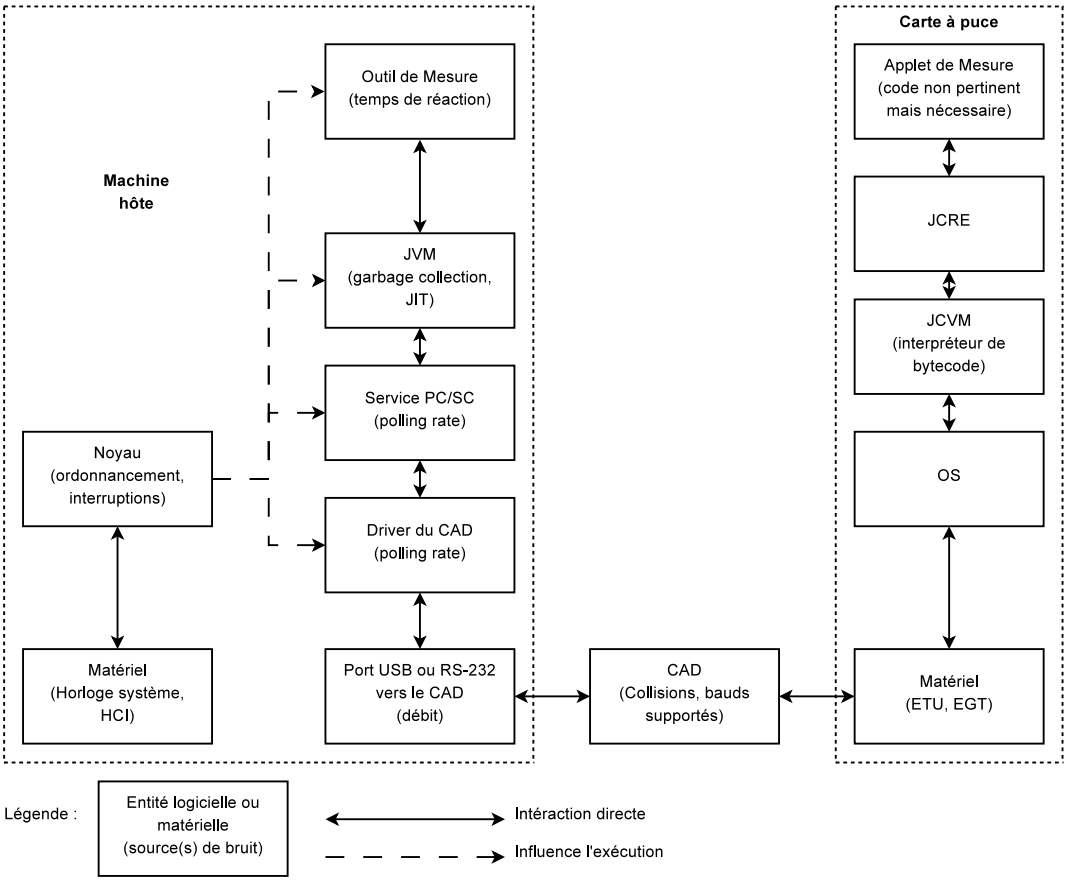


FIG. 4.7 – Les sources de bruit pendant la mesure

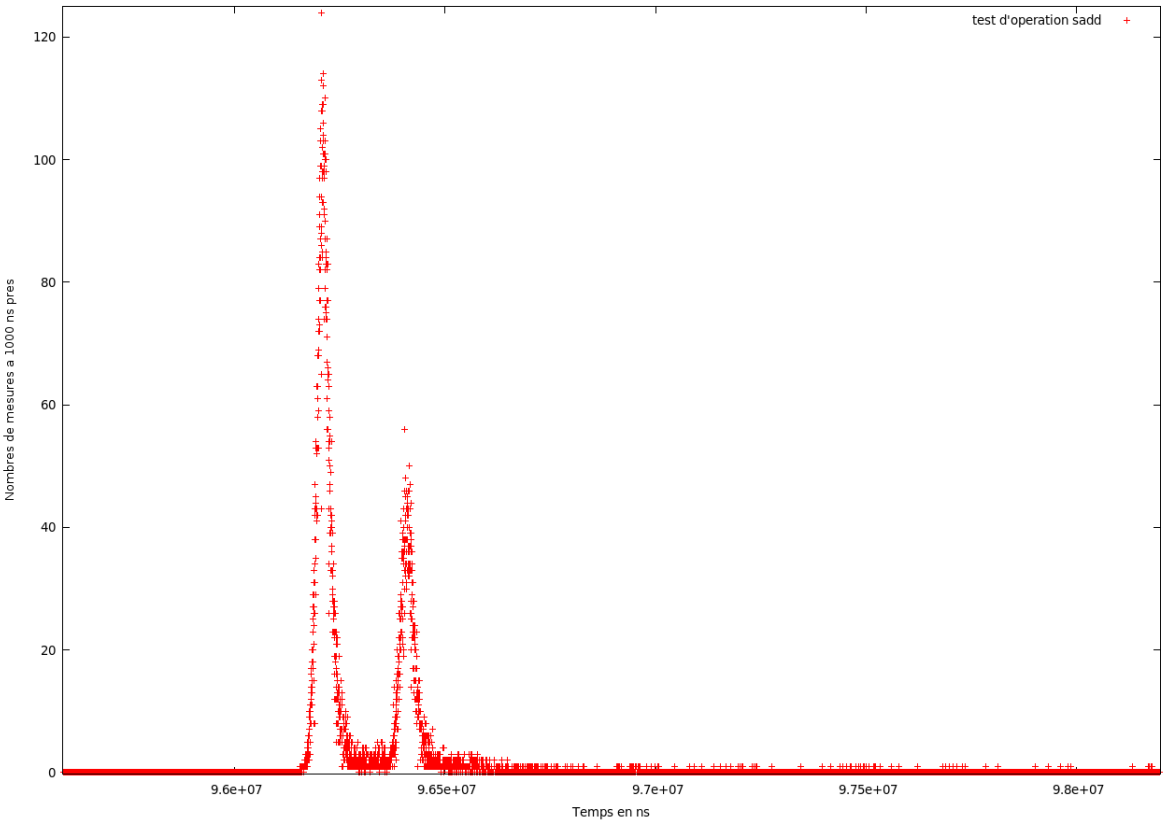


FIG. 4.8 – Distribution d'un exemple de mesures obtenues pour le test d'opération `sadd` sous Windows Vista L=P2^2 = 100

lème de prémisse : les valeurs mesurées ne sont pas nécessairement normalement distribuées. La figure 4.8 illustre un exemple de mesures obtenues et distribuées d'une manière manifestement non normale. L'utilité de l'outil Filter est remis en cause par cette distribution.

Nous reviendrons sur les problèmes liés à la correction des mesures effectuées dans le chapitre 5.

4.2.4 Le module Extractor

Nous avons à ce point des données qui sont mesurées de manière brute sur les cartes à puce. C'est à dire qu'on a d'un côté des mesures de tests d'opération et d'un autre des mesures de tests de référence.

Le module Extractor est utilisé pour isoler le temps d'exécution des opérations visées parmi une masse de mesures brutes. Pour minimiser l'effet de ces interférences, nous avons besoin d'isoler le temps d'exécution de ces opérations tout en s'assurant que le temps d'exécution à mesurer est suffisamment important. Le principe de l'isolement du temps d'exécution est expliqué en détails dans la section 3.2. Le module Extractor reprend ces principes pour isoler les temps d'exécution.

Extractor ne fait que manipuler des équations qui sont indiquées par le développeur du test. Pour développer un test, on doit donc concevoir :

- l'applet (y compris éventuellement la modification du fichier JCA),
- le script qui permet de mesurer un test,
- l'équation qui permet de résoudre les mesures.

En outre, le développeur doit vérifier que les bytecodes dont dépend l'applet sont couverts (déjà mesurés), et il doit associer tous les tests d'opération à un test de référence.

Les équations sont notées par le développeur avec des chaînes de caractères (`String` en Java) de la forme :

$$x \times m_0 + o_1 \times m_1 + ... + o_n \times m_n$$

où :
- o_i pour $i \in [1..n]$ est le temps d'exécution d'un des éléments mesurés (bytecode et méthode de l'API),
- $m_i \in \mathbb{N}$ est le nombre d'occurrences d'un bytecode dans le `run()` du test de l'opération qui n'apparaissent pas dans le test de référence,
- x est le temps d'exécution du bytecode ou de l'appel de méthode de l'API que l'on essaie d'isoler.

Le programme Extractor utilise un *parser* construit à l'aide de JavaCC (Java Compiler Compiler - un générateur de parser pour Java) [57]. La chaîne de caractères est analysée grammaticalement par le programme qui construit un arbre de syntaxe abstraite correspondant à l'équation suivante :

$$t_{mesure} = t_{reference} + (x \times m_0 + o_1 \times m_1 + ... + o_n \times m_n) \times L$$

où :
- t_{mesure} est le temps mesuré pour le test de l'opération,
- $t_{reference}$ est le temps mesuré pour le test de référence,
- L est la taille de la boucle.

Les équations ne sont jamais très complexes (il n'y a jamais d'exposant par exemple). Il est donc facile pour le programme d'isoler x en produisant l'équation :

$$x = \frac{t_{mesure} - t_{reference}}{L \times m_0} - \frac{o_1 \times m_1 + ... + o_n \times m_n}{m_0}$$

Le programme prend en charge la résolution de dépendances avant de résoudre le système d'équations complet issu des mesures.

4.2.5 Le module Profiler

Le but de ce module est la production d'une note de synthèse à partir des mesures isolées brutes extraites avec le module Extractor. Profiler doit pouvoir produire un document HTML donnant une note générale pour une carte donnée, des notes spécifiques pour des domaines d'applications, les performances déterminantes, c'est à dire les performances des bytecodes et des méthodes de l'API qui ont un impact sur les notes pour les différents domaines.

Nous avons défini des domaines d'applications avec plusieurs applets utilisées dans chacun de ces domaines. Certaines de ces applets sont propriétaires et d'autres, comme MuscleCard [104] qui est un projet utilisé dans le domaine de l'identification, est sous licence BSD. Les autres domaines couverts sont les transports et le domaine bancaire. Ces applets sont toutes d'utilisations courantes dans les domaines en question.

Ces applets sont relativement complexes pour des applications Java Card. `CardEdge.cap` qui est la version compressée du code de MuscleCard (ou MCardApplet) qui va être chargé sur la carte à puce est de l'ordre de 10ko, ce qui ne comprend pas les différents champs après l'installation de l'applet, notamment les clés qui vont être utilisées.

Pour produire des notes, il nous faut donner des poids aux différentes mesures effectuées. L'idéal est d'avoir des notes qui soient représentatives de l'utilisation "normale" des applets, de la même manière que **SPECviewperf** [26] donne des notes représentatives d'applications utilisant OpenGL de manière intensive dans leurs usages courants. Pour cela, il nous faut un scénario d'utilisation quotidienne de chaque applet.

Par exemple, l'utilisation de l'applet MuscleCardApplet est basée sur des méthodes d'identification avec plusieurs niveaux de sécurité :

- par code PIN,
- par des défis/réponses utilisant des algorithmes cryptographiques,
- par des moyens laissés de côtés par l'application, mais qui peuvent comprendre des données biométriques.

L'application cliente fera typiquement usage de mécanismes tels que **PKCS#11** [69] (Public Key Cryptography Standards) pour accéder à des logiciels liés avec une authentification unique (Single Sign-on). L'authentification par l'applet est effectuée en plusieurs étapes :

1. par code PIN,
2. par clés (authentification forte).

Chaque authentification donne droit à une authentification subséquente qui donnera d'autres droits en termes d'accès aux applications. À la suite de l'authentification, l'utilisateur va passer par plusieurs phases de chiffrement et de déchiffrement avec les applications utilisées en relation avec l'applet.

Ainsi on peut réaliser un scénario d'utilisation réaliste de l'applet :

1. L'utilisateur se connecte par code PIN.
2. Une étape d'authentification a lieu entre la machine de l'utilisateur et sa carte à puce en commençant par le chiffrement d'un nombre aléatoire par la carte à puce. La valeur chiffrée est envoyée à la machine hôte.
3. La machine hôte déchiffre le nombre aléatoire. Elle renvoie le nombre aléatoire déchiffré à la carte à puce.
4. La carte à puce vérifie que la machine hôte a bien pu déchiffrer le nombre chiffré envoyé.
5. En cas de succès, l'utilisateur est maintenant fortement authentifié. Il peut par la suite utiliser les différents algorithmes cryptographiques auxquels son authentification forte lui permet d'accéder (par exemple avec RSA).
6. Plusieurs séquences de chiffrement/déchiffrement de données s'ensuivent.

Pour les besoins du scénario, toutes les étapes se déroulent avec succès. On suppose par exemple que l'utilisateur présente toujours le bon code PIN. Il faut pouvoir interpréter les scénarios conçus pour en extraire l'usage qui est fait de chaque bytecode et de chaque méthode de l'API.

Une machine virtuelle Java Card est instrumentée pour compter les différentes opérations exécutées au lancement d'un script pour une application donnée. Plus précisément, la machine virtuelle est simulée sur un PC. L'instrumentation est une méthode simple à implémenter en comparaison avec les méthodes basées sur l'analyse statique du code, et peut atteindre un niveau de précision élevé. Le simulateur donne des informations utiles telles que :

- pour les méthodes de l'API : les types et les valeurs des paramètres de méthodes, la taille des tableaux passés en paramètre.
- pour les bytecodes : le type et la durée des vecteurs pour des bytecodes qui manipulent des tableaux, l'état de la transaction lors de l'appel du bytecode. On veut aussi savoir sur quels arguments les opérations conditionnelles (`if then else`, `switch case` ...) ont lieu (*vrai* ou *faux* par exemple).

De cette manière, on peut assembler des tableaux d'utilisation des bytecodes et des méthodes de l'API. On présente en Annexe un exemple de tableau dans le domaine de l'identification.

Ainsi à partir des données de la machine virtuelle instrumentée, nous attribuons pour chaque domaine d'application un nombre représentant la performance d'un nombre représentatif d'applets du domaine sur la carte testée. Chacun de ces nombres sera utilisé pour calculer une note globale et pour pondérer un domaine d'applications. Ces poids sont calculés en fonction du nombre d'occurrences d'une portion de code de Java Card durant une utilisation normale d'applets standard pour un domaine donné. Par exemple, si l'on désire tester une carte du domaine transport, on utilise les valeurs statistiques collectées à partir d'un ensemble d' applets représentatives du domaine pour évaluer l'impact de chaque caractéristique mesurée sur la carte.

Considérons la mesure de la caractéristique f sur la carte à puce c pour un domaine d'applications d.

Pour un ensemble de n_M mesures extraites (fournies par Extractor) $M^1_{c,f}...M^{n_M}_{c,f}$ considérées comme significatives de la caractéristique f, on peut déterminer une moyenne arithmétique $\overline{M^{c,f}}$ modélisant la performance en terme de temps d'exécution de la plateforme c testée pour cette caractéristique f.

$$\overline{M_{c,f}} = \frac{\Sigma_{i=1}^{n_M} M^i_{c,f}}{n_M}$$

Étant donné n_C cartes à puce pour lesquelles f a été mesurée, il est nécessaire de déterminer le temps d'exécution moyen R_f qui servira de base de référence pour tous les tests.

$$R_f = \frac{\Sigma_{i=1}^{n_C} \overline{M_{i,f}}}{n_C}$$

Ainsi la note $N_{c,f}$ d'une carte à puce c pour une caractéristique f est la relation entre R_f et $\overline{M_{c,f}}$:

$$N_{c,f} = \frac{R_f}{\overline{M_{c,f}}}$$

Cependant, cette note n'est pas pondérée. Pour chaque domaine d'application et chaque caractéristique, il nous faut obtenir un coefficient qui est un modèle de l'importance de la caractéristique dans le domaine. Pour chaque paire (caractéristique f , domaine d), on associe un coefficient $\alpha_{f,d}$ qui modélise l'importance de f dans d. Plus une caractéristique est utilisée au sein d'applications typiques du domaine, plus le coefficient est grand :

$$\alpha_{f,d} = \frac{\beta_{f,d}}{\Sigma_{i=1}^{n_F} \beta_{i,d}}$$

Où :

- $\beta f, d$ est le nombre total d'occurrences de la caractéristique f dans les applications représentatives du domaine d. Ces nombres sont ceux obtenus en passant les scénarii d'utilisation "normales" des applets représentatives des domaines d'application dans la machine virtuelle outillée.
- n_F est le nombre total de caractéristiques impliquées dans le test.

Par conséquent le coefficient $\beta_{f,d}$ représente le rapport d'occurrences de f parmi toutes les caractéristiques dans les scénarii des applets du domaine d.

Étant donné une caractéristique f, une carte c et un domaine d, la note pondérée $W_{c,f,d}$ est calculée comme suit :

$$W_{c,f,d} = N_{c,f} \times \alpha_{f,d}$$

La note de référence, calculée sur plusieurs cartes à puce, à laquelle on peut comparer la note pondérée est :

$$\overline{W_{f,d}} = R_f \times \alpha_{f,d}$$

La note globale $P_{c,d}$ pour une carte c d'un domaine d est la suivante :

$$P_{c,d} = \frac{\Sigma_{i=1}^{n_F} \overline{W_{i,d}}}{\Sigma_{i=1}^{n_F} W_{i,c,d}}$$

On obtient par ce calcul une note qui doit s'approcher de 1 pour une carte dont les performances sont moyennes dans un domaine d.

Une note indépendante du domaine général pour une carte est calculée comme la moyenne des notes des différents domaines.

$$\overline{P_c} = \frac{\Sigma_{i=1}^{n_D} P_{c,d}}{n_D}$$

Où n_D est le nombre de domaines considérés. Trois domaines sont considérés : le domaine bancaire, le domaine des transports et le domaine de l'identité. Les notes $\overline{P_c}$, $P_{c,d}$, $W_{c,f,d}$, $N_{c,f}$ sont des notes pour lesquelles une note "meilleure" signifie une note plus "haute" (c'est une note HB - Higher is Better). $\overline{M_{c,f}}$ est une moyenne de mesures brutes. Celle-ci est donc une note qui sera "meilleure" si elle est faible (LW - Lower is Better). Ces métriques HB sont typiques de la mesure de performance [35] [56].

Finalement, une page de présentation des résultats est générée par l'outil. Cette page est constituée de trois niveaux de détails :

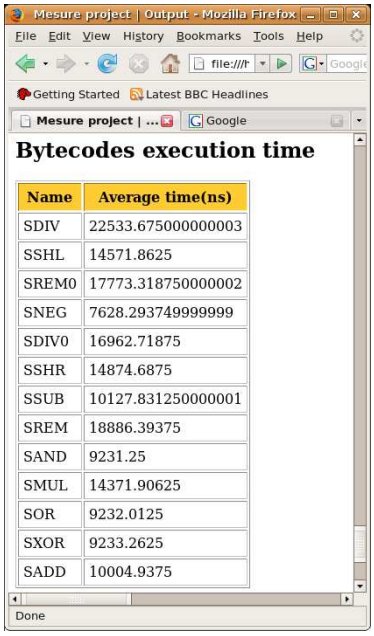
- un niveau brut concerne les temps d'exécution bruts des bytecodes et des entrées de l'API (cf figure 4.9),
- chaque domaine d'application est décrit dans un niveau qui présente les résultats des bytecodes et des méthodes de l'API de poids les plus forts (cf figure 4.10),
- des notes finales pour chaque domaine et la moyenne de la carte à puce mesurée pour tous les domaines sont affichées dans le niveau de détail le plus grossier. Un diagramme en toile d'araignée permet à l'utilisateur d'évaluer ces performances d'un coup d'œil (cf figure 4.11).

4.3 Couverture

Un point important du projet MESURE est de définir un outil de mesure de la performance qui ait un sens pour les applications spécifiques de l'industrie. Il nous faut donc définir ce qui est important et ce qui ne l'est pas.

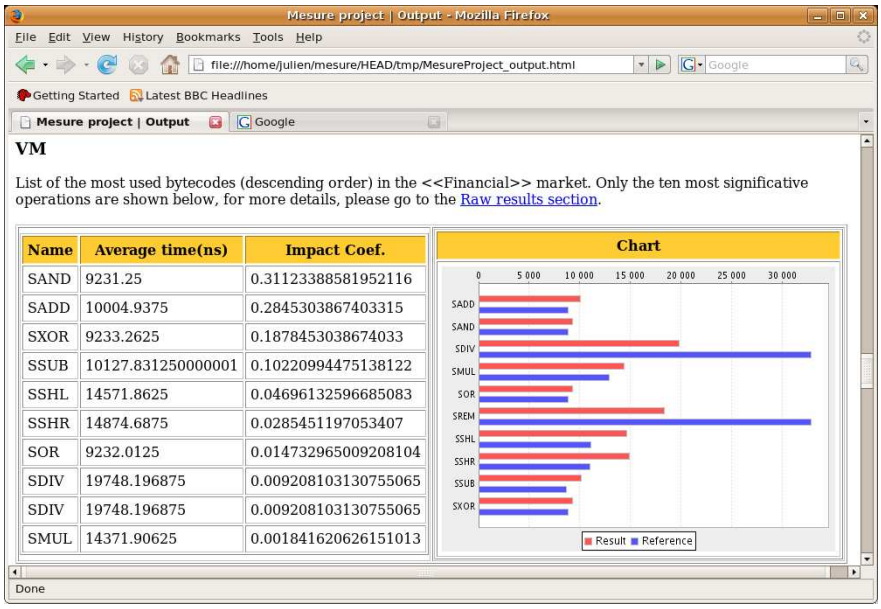
En particulier, nous avons laissé de côté l'API SIM Toolkit. Cette API nécessite un mode de chargement particulier des applications qui augmenterait considérablement la complexité du programme. De plus, il n'y a pas d'applet de référence utilisant l'API SIM Toolkit sur le marché.

Nous nous sommes donc concentrés sur l'API Java Card et l'API Global Platform, ce qui nécessite de pratiquer des mesures concernant la machine virtuelle et l'environnement d'exécution de Java Card. Des contraintes de temps de mesures sont généralement admises dans le cadre



Name	Average time(ns)
SDIV	22533.675000000003
SSHL	14571.8625
SREM0	17773.318750000002
SNEG	7628.293749999999
SDIV0	16962.71875
SSHR	14874.6875
SSUB	10127.831250000001
SREM	18886.39375
SAND	9231.25
SMUL	14371.90625
SOR	9232.0125
SXOR	9233.2625
SADD	10004.9375

FIG. 4.9 – Des mesures brutes



Name	Average time(ns)	Impact Coef.
SAND	9231.25	0.31123388581952116
SADD	10004.9375	0.2845303867403315
SXOR	9233.2625	0.1878453038674033
SSUB	10127.831250000001	0.10220994475138122
SSHL	14571.8625	0.04696132596685083
SSHR	14874.6875	0.0285451197053407
SOR	9232.0125	0.014732965009208104
SDIV	19748.196875	0.009208103130755065
SDIV	19748.196875	0.009208103130755065
SMUL	14371.90625	0.001841620626151013

FIG. 4.10 – Des mesures de bytecode les plus importantes d'un domaine

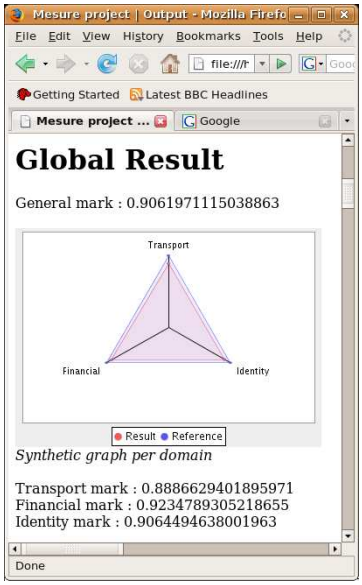


FIG. 4.11 – Les notes globales

d'un benchmark. Ainsi, nous ne devons pas effectuer des mesures exhaustives pour des raisons pratiques. Nous donnerons donc ici, une liste des éléments qu'il n'est pas urgent de mesurer.

Les plates-formes concernées ici sont celles qui sont compatibles Java Card 2.1. En effet, les versions ultérieures à celle-ci sont toutes compatibles vers les versions précédentes jusqu'à la 2.1. La version 3.0 de Java Card est bien différente et peu de cartes à puce compatibles sont véritablement employées dans le monde. La version 2.1 représente tout d'abord l'intersection de plusieurs domaines d'application. Ensuite cette version est aussi la première version véritablement moderne et qui fasse office de standard pour les cartes que nous avons.

4.3.1 Notions de priorités dans les mesures

Pour que les mesures soient représentatives des préoccupations de performance de l'industrie, il faut qu'elles soient faites avec un souci de réalisme dans l'utilisation quotidienne qui est faite des cartes à puce par leurs porteurs.

Dans cette optique, il est inutile de considérer certains aspects du fonctionnement des cartes à puce qui ne font pas partie des actions courantes entreprises par les utilisateurs. On peut donc ignorer :

- L'installation de l'application.
  Toute application Java Card a besoin de définir une méthode d'installation, qui lorsqu'elle est lancée permet d'instancier une classe correspondant à l'applet, de l'initialiser et de l'enregistrer auprès de l'environnement d'exécution. Une fois l'enregistrement effectué, l'applet est prête à recevoir des commandes. L'environnement d'exécution de Java Card peut par la suite sélectionner cette applet et lui envoyer les commandes APDUs subséquentes.
- La personnalisation de l'applet.
  Les applications Java Card peuvent par la suite être personnalisée. On va à ce titre fournir ou générer tous les éléments nécessaires, comme les clés et les données de personnalisation pour permettre à l'applet de fonctionner normalement.
- L'effacement de l'application.
  À tout moment, les applications Java Card peuvent être effacées. Cette opération permet

de désinscrire l'applet auprès de l'environnement d'exécution de la plateforme. Par la suite l'applet ne pourra plus être sélectionnée.

En outre, il est d'usage dans la programmation d'applet de ne pas allouer d'objets ou de tableaux d'octets de manière intempestive. Les ressources comme la mémoire (en particulier la RAM) sont contraintes dans les cartes à puce. En conséquence, toutes les opérations d'allocation et d'instanciation ont lieu pendant les étapes d'installation et de personnalisation de l'applet. Il est donc inutile de mesurer ces allocations pour obtenir une note de performance.

Certains bytecodes et méthodes de l'API ne sont donc pas mesurés, comme les bytecodes `new`, `newarray` et `anewarray`, les constructeurs de l'API et les méthodes suivantes :

- `JCSystem.makeTransientXXXArray` qui sert à allouer des tableaux volatiles,
- `KeyBuilder.buildKey` qui est utilisé pour générer des clés,
- `getInstance` qui permet d'allouer de la mémoire pour un algorithme cryptographique,
- `Applet.register(), Applet.register(byte[],short,byte)` qui ont trait à l'enregistrement des applets vis-à-vis de l'environnement d'exécution Java Card.

Un autre aspect qui n'est pas considéré est l'utilisation erronée de l'application, par exemple les cas où l'utilisateur va présenter un mauvais code PIN. En effet, si on considère la performance, les cas d'échecs sont moins importants, car moins fréquents. Les interventions de l'utilisateur avec la carte à puce sont en général restreintes à quelques cas de figures et souvent encadrées par des applications côté terminal qui vont en principe envoyer des APDUs bien construits à l'applet. On considère également que les applets utilisées sont suffisamment bien conçues pour éviter toute manipulation hasardeuse des données.

Les entrées/sorties (E/S) ne sont pas non plus considérées lors des mesures. Le projet MESURE permet de s'abstraire des conditions de mesures en se concentrant sur les performances des cartes à puce elles mêmes et non pas de la plateforme de mesure. Mesurer avec précision les E/Ss, serait de ce point de vue problématique, car les mesures seront fortement dépendantes de plusieurs éléments extérieurs à la carte à puce :

- L'OS
  Les conditions d'exécution de l'outil de mesure ont nécessairement une grande influence sur le bruit dans les mesures. Par exemple, les résultats risquent d'être différents si on mesure dans un environnement dédié à la mesure ou si on mesure dans un environnement utilisé par d'autres applications.
- Le CAD
  Le CAD joue un rôle important dans les bruits obtenus lors des mesures. Le fait de passer d'un CAD à un autre pourrait en grande partie influer sur les mesures. De la même manière le fait d'utiliser un driver particulier pourrait entraîner une différence de dispersion des résultats et donc une différence de précision.
- L'API d'E/S
  Les outils de MESURE ont été programmés en Java en utilisant un JDK 6.0 qui par son implémentation de la JSR 268 fournit un accès à des fonctionnalités de PC/SC. Le fait de faire des mesures en utilisant directement PC/SC ou CT-API pourrait typiquement engendrer des différences dans les résultats perçus.
- La machine virtuelle utilisée
  La machine virtuelle a elle-même un impact sur les mesures. Utiliser différentes versions d'une machine virtuelle pourrait aussi induire des différences dans les mesures, ne serait-ce que par les différentes utilisations de compilateurs JIT qui peuvent être faites. Le fait de passer par un programme de mesure écrit en C et en assembleur pourrait également causer une différence dans la perception de la mesure.

Par ailleurs mesurer avec du bruit peut être contrebalancé par le fait d'utiliser des boucles et

en vérifiant statistiquement la dispersion des valeurs mesurées si on s'en tient à mesurer des temps d'exécution sur les cartes à puce. Pour mesurer les E/Ss, il faudrait se résoudre à ne plus utiliser de boucles, ce qui rendrait les mesures difficiles à reproduire. Plus généralement, MESURE a été conçu avec à l'esprit, la facilité de mesure et en ne spécifiant pas particulièrement l'environnement de mesure. Ce dernier doit juste être composé d'une machine avec un lecteur PC/SC et avec une machine virtuelle Java 6. Les mesures plus précises peuvent être faites avec l'aide d'outils spécialisés (lecteurs de précision). Mais tout utilisateur de carte à puce détenteur d'un CAD simple et peu coûteux pourrait malgré tout effectuer des mesures.

Des informations contenues dans l'ATR permettent de déterminer les vitesses de transmission acceptées par la carte à puce compatible ISO 7816-3 [53] (de 9600 bits par seconde à 115200 bits par seconde). Il est aussi important de noter que la transmission est sujette à du bruit. Aussi il est conseillé de favoriser une transmission sûre à une transmission à haute vitesse des données [107] [100]. La mesure de performance serait redondante avec les informations issues de l'ATR. On pourrait considérer que la mesure des E/Ss concerne les bruits de la machine hôte et du CAD autant que la mesure de la performance de la carte à puce. Les travaux de Markantonakis [76] couvrent ces aspects.

4.3.2 API

Dans l'API cryptographique de Java Card, les clés et algorithmes rarement supportés par les cartes à puce ou qui posent un problème de sécurité ne sont pas couverts par les mesures. Voici une liste de ces objets :

- Les clés DSA.
- Les clés RSA de taille autre que 512, 768, 1024 ou 2048 bits.
- Les paires de clés (classe `KeyPair`).
- Les empreintes numériques RIPE MD-160.
- Les signatures MAC4 basées sur DES.
- Les signatures MAC8 basées sur DES avec un bourrage PKCS5.
- Les signatures basées sur DSA.
- Les signatures basées sur RSA utilisant RIPE MD-160.
- Les signatures basées sur RSA utilisant un bourrage RFC 2409.
- Le chiffrage DES utilisant un bourrage PKCS5
- Le chiffrage RSA utilisant un bourrage ISO 17888.

Des éléments liés à l'API n'ont pas été utilisés par les applets testées avec la machine virtuelle outillée. En voici une liste :

- Les clés DES de durées `CLEAR_ON_RESET`. Les clés sont soit des clés persistantes soit des clés de session de durée `CLEAR_ON_DESELECT`.
- Les empreintes numériques MD5. Pour des raisons de sécurité (MD5 ne résistant pas aux collisions [120]), SHA-1 est préféré à MD5.
- Les générateurs de nombres pseudo-aléatoires. Pour des raisons de sécurité les applications utilisent des générateurs sécurisés de nombres aléatoires.
- La méthode `RandomData.setSeed`. Cette méthode est utilisée pour la génération de nombres pseudo aléatoires.
- Les méthodes `getAlgorithm`, `getLength`, `Key.getSize`, `Key.getType`, `Key.isInitialized`. Les valeurs retournées par ces méthodes sont généralement implicitement connues des applications.
- Les méthodes `getExponent, getModulus, getDP1, getDQ1, getP, getPQ, getQ`. Ces méthodes retournent une partie des clés en clair. Pour des raisons de sécurité, les applications

ne les invoquent pas.
- Les méthodes `Cipher.update, MessageDigest.update, Signature.update`. Les données sont souvent préparées dans un tableau avant d'être chiffrées, signées ou hachées. Les méthodes `doFinal` sont alors utilisées directement.
- `MessageDigest.reset`. Les applications ne nécessitent en général pas cette méthode, car les objets cryptographiques reviennent à leurs états initiaux après un appel aux méthodes `MessageDigest.doFinal`.
- `Object.equals`. Les applications ont rarement besoin de comparer deux objets.
- `AID.getBytes, AID.equals(Object), AID.equals(byte[],short,byte), AID.partialEquals, AID.RIDEquals, JCSystem.getAID, JCSystem.getPreviousContextAID, JCSystem.lookupAID, JCSystem.getAppletShareableInterfaceObject`. Ces méthodes sont utiles dans un contexte de partage de données avec une autre application, mais pour des raisons de sécurité, ce mécanisme est rarement utilisé dans les applications.
- `APDU.waitExtension`. Les plates-formes Java Card utilisent généralement un mécanisme d'horloge automatisé pour demander plus de temps de calcul au CAD, et dans ces cas, cette méthode peut ne rien faire. Par conséquent, elle est très rarement utilisée.
- `APDU.getInBlockSize, APDU.getOutBlockSize, APDU.getNAD`. Ces méthodes ne sont utiles que pour le protocole T=1 et l'information fournie est rarement nécessaire pour les applications. `getOutBlockSize` retourne 254 pour des cartes de paiement, comme spécifié par l'IFSD (taille du champ d'information pour le terminal) [37], et `getInBlockSize` retourne un IFSC (taille du champ d'information pour la carte à puce) dépendant de la taille de l'APDU [52].
- `APDU.receiveBytes`. La mémoire tampon APDU permet généralement de recevoir les octets en entrée dans les commandes APDU. Cette méthode qui permet de recevoir les octets suivants n'est donc que rarement utilisée.
- `APDU.setOutgoingNoChaining`. Les applications utilisent toutes la méthode `APDU.setOutgoing`.
- `JCSystem.abortTransaction`. Puisqu'on ne considère que les cas où les commandes sont des réussites, il n'y a pas de transaction avortée.
- `JCSystem.getMaxCommitCapacity`, `JCSystem.getUnusedCommitCapacity`. Les valeurs retournées par ces méthodes sont indicatives.
- `JCSystem.getTransactionDepth`. Aucune méthode autre que celles dédiées (`JCSystem.XXXTransaction`) ne peuvent initier ou terminer une transaction. Par conséquent, le niveau d'imbrication de la transaction courante est connue implicitement.
- `JCSystem.getVersion`. La version de Java Card est connue implicitement et est déterminée par la version des fichier d'export lors de la conversion des applets.
- `JCSystem.isTransient`. Le type (volatile ou permanent) des tableaux est connu de manière implicite dans le contexte d'une application.
- `OwnerPIN.getValidatedFlag, OwnerPIN.setValidatedFlag`. Ces méthodes sont protégées et les applications n'utilisent pas de classe héritant de la classe `OwnerPIN`.
- `OwnerPIN.update`. Le code PIN n'est pas mis à jour durant l'utilisation standard d'une applet. Cette méthode peut être appelée pendant la phase de personnification de l'application.

4.3.3 Bytecodes

Lorsque l'on étudie l'utilisation des bytecodes dans l'usage courant des applications standard, il apparaît que les bytecodes utilisant un adressage "large" ne sont pas communément utilisés. Cela est sans doute en partie dû à la programmation en général dans les applets testées qui n'utilisent peu ou pas de méthodes "longues". Ces bytecodes sont donc sans doute très peu utilisés de manière générale dans les applets Java Card. La conséquence de cette sous utilisation est que les mesures de tels bytecodes n'est pas une priorité pour le projet MESURE. Par ailleurs, étant donné la faible occurrence de ces bytecodes dans les statistiques d'utilisation, les bytecodes à large adressage qui apparaissent malgré tout de rares fois, ont un faible impact dans les notes données par les outils.

Ces bytecodes peuvent être catégorisés de la manière suivante :

- Accès aux variables globales : `getfield_a_w`, `getfield_b_w`, `getfield_i_w`, `getfield_s_w`, `putfield_a_w`, `putfield_b_w`, `putfield_i_w`, `putfield_s_w`
- Incrémentations de variables locales : `sinc_w`, `iinc_w`

- Instructions conditionnelles simples : `ifeq_w`, `ifge_w`, `ifgt_w`, `ifle_w`, `iflt_w`, `ifne_w`, `ifnonnull_w`, `ifnull_w`
- Instructions conditionnelles complexes : `if_acmpeq_w`, `if_acmpne_w`, `if_scmpeq_w`, `if_scmpge_w`, `if_scmpgt_w`, `if_scmple_w`, `if_scmplt_w`, `if_scmpne_w`

- Saut : `goto_w`

En outre, les bytecodes suivants ne sont pas utilisés dans les applets testées (et sans doute très rarement utilisés de manière générale) :

`aastore`, `anewarray`, `astore_0`,
`athrow`, `bipush`, `getfield_a_w`,
`getfield_b_w`, `getfield_i`,
`getfield_i_this`, `getfield_i_w`,
`getfield_s_w`, `getstatic_i`, `getstatic_s`,
`i2b`, `i2s`, `iadd`, `iaload`,
`iand`, `iastore`, `icmp`, `iconst_0`,
`iconst_1`, `iconst_2`, `iconst_3`,
`iconst_4`, `iconst_5`, `iconst_m1`,
`idiv`, `if_acmpeq`, `if_acmpeq_w`,
`if_acmpne_w`, `ifgt_w`, `ifle_w`,
`ifnonnull_w`, `ifnull_w`, `if_scmpeq_w`,
`if_scmpge_w`, `if_scmpgt_w`,
`if_scmple_w`, `if_scmplt`, `if_scmplt_w`,
`iinc`, `iinc_w`, `iipush`,
`iload`, `iload_0`, `iload_1`,

`iload_2`, `iload_3`, `ilookupswitch`,
`imul`, `ineg`, `instanceof`, `ior`,
`irem`, `ireturn`, `ishl`, `ishr`,
`istore`, `istore_0`, `istore_1`,
`istore_2`, `istore_3`, `isub`,
`itableswitch`, `iushr`, `ixor`, `new`,
`nop`, `pop2`, `putfield_a_this`,
`putfield_a_w`, `putfield_b_this`,
`putfield_b_w`, `putfield_i`,
`putfield_i_this`, `putfield_i_w`,
`putfield_s_this`, `putfield_s_w`,
`putstatic_b`, `putstatic_i`, `putstatic_s`,
`s2i`, `sinc_w`, `sipush`, `sstore_0`,
`sushr`, `swap_x`.

Sur ces 88 bytecodes, 52 ont trait aux entiers 32 bits. En effet, le support pour les entiers 32 bits est optionnel dans les spécifications de Java Card précédent la version 3.0, ce qui a pour conséquence que :

- la majeure partie des cartes à puce en circulation ne font pas usage des entiers 32 bits (`integer`),
- il n'est pas vraiment réaliste de développer une application utilisable sur de nombreux types de cartes à puce si cette application fait usage d'un type optionnel.

21 de ces 88 bytecodes ont trait à l'adressage large (en fait 3 bytecodes servent à manipuler des entiers 32 bits avec un adressage large).

Les 18 bytecodes restants sont les suivants : `aastore`, `anewarray`, `astore_0`,
`getstatic_s`, `if_acmpeq`, `if_scmplt`,
`instanceof`, `new`, `nop`, `pop2`,
`putfield_a_this`, `putfield_b_this`,
`putfield_s_this`, `putstatic_b`,
`putstatic_s`, `sstore_0`, `swap_x`.

Certains de ces derniers bytecodes ne sont pas utilisés de manière courante durant l'exécution des applications testées parce qu'ils servent à allouer de l'espace mémoire à des objets, ce qui devrait se dérouler pendant les phases de personnalisation des applets plutôt que pendant leurs usages courants (`aastore, anewarray, instanceof, new, putfield_a_this`). Ces mêmes bytecodes sont aussi relativement gourmands en mémoire, ce qui expliquerait leur sous utilisation dans un usage courant. Certains bytecodes sont tout simplement rarement générés à la compilation/conversion des applets (`if_acmpeq, if_scmplt, nop, pop2, swap_x`). Finalement, le bytecode restant est `athrow`, ce qui est clairement utilisable seulement en cas de lancement d'exception, ce qui est un comportement rarement utilisé dans l'usage courant des applets, mais qui peut apparaître dans le cas d'une erreur d'entrée par le terminal. Comme on ne considère que les cas où l'utilisateur ne commet pas d'erreur, ces cas là sont nécessairement mis de côté par les statistiques d'utilisation des bytecodes.

4.4 Conclusion

Les outils qui font parti du projet MESURE ont été mis à la disposition du public en mars 2008. La spécification Java Card 3.0 a été publiée à cette même date. Cela ne remets pas nécessairement en cause les outils présenté ici. En effet, nous avons pu tester la version Classic avec

nos outils au sein d'un simulateur. Cependant, la version Connected de Java Card 3.0 nécessite d'autres développements et une autre méthodologie, à cause de l'introduction du multithreading et de la plus forte dépendance au ramasse miettes.

Nous devons passer par une étape de validation de nos mesures. Nous avons en effet démontré que les mesures isolées restent cohérentes si l'on fait varier les tailles de boucles. L'étape suivante d'après Jain [56] ou Lilja [74] consiste à caractériser les données obtenues en vérifiant que l'on peut bien représenter les valeurs et que celles-ci sont bien significatives.

5

Analyse Statistique de la performance d'une carte

Sommaire

5.1 Introduction

Le principe de notre mesure de performance est basé sur la différence de temps d'exécution d'une commande qui va lancer une boucle avec un bytecode donné et le temps d'exécution d'une commande qui va lancer une boucle similaire à la première mais sans le bytecode qui nous intéresse. Nous appelons *test de référence* le test avec cette dernière boucle et le *test de l'opération*, le test avec la boucle qui inclut le bytecode qui nous intéresse. La différence entre les deux temps doit correspondre à des exécutions du bytecode isolé, celui ci étant exécuté à chaque tour de boucle. Ce type de méthode est utilisé dans d'autres projets ([102], [38], [33] qui en sont les exemples les plus complets), même si ces travaux là ne descendent jamais jusqu'au niveau du bytecode et se contentent d'isoler du code Java, ce qui peut être plus arbitraire.

Les travaux cités ne tentent pas non plus de valider les résultats, même si certains d'entre eux utilisent des techniques supposés plus précises, comme en utilisant un oscilloscope pour effectuer les mesures ([38], [40]).

Cependant, nous n'avons jusqu'ici pas fait la démonstration de la validité de notre méthode. Nous allons chercher ici à valider notre mesure en nous aidant de méthodes statistiques, et en comparant nos résultats obtenus avec des CAD classiques avec des résultats sur des tests similaires, mais cette fois mesurés avec un lecteur de précision du commerce.

5.2 Validation des tests

5.2.1 Correction statistiques des mesures

La distribution attendue d'une mesure est une distribution normale. D'après Lilja [74], la moyenne arithmétique est une valeur représentative acceptable d'un ensemble de valeurs pour une distribution normale, étant donné qu'on effectue un nombre suffisant de mesures (Lilja recommande 30 mesures au minimum). Cependant Rehioui [102] et Erdmann [38] remarquent indépendamment et sur des cartes différentes que des résultats obtenus en faisant de simples mesures sans prendre en compte l'isolation de bytecodes ou de groupes de bytecodes en utilisant un PC et un CAD classique sont distribués de manières non normales. Ces résultats sont obtenus sur des cartes à puce IBM JCOP41. Rehioui présente certaines distributions prenant la forme de "peignes", c'est à dire de plusieurs pics de tailles inégales et répartis de part et d'autre de la moyenne. L'auteur présente aussi une méthode pour se débarrasser de ce qu'elle considère comme des mesures incohérentes en éliminant une portion (basée sur l'écart type des temps mesurés) de mesures qui sont trop éloignées de la moyenne. Cette opération est par la suite répétée plusieurs fois jusqu'à ce que l'on ne garde qu'un pourcentage déterminé par l'utilisateur du nombre de mesures initiales. Le but derrière ces opérations est, d'après l'auteur, de trouver le pic de distribution le plus élevé. On pourrait objecter que dans le cas où le pic le plus élevé est en fait éloigné de la moyenne, ou que dans une distribution avec de nombreux pics de tailles similaires, cette technique semblerait futile.

Erdmann [38] fait des remarques similaires sans vraiment rentrer dans les détails. Il est question de "pas" perceptibles entre plusieurs temps mesurés, et que ces pas sont de tailles similaires. Les cartes à puce concernées sont, cette fois, produites par Infineon, et elles sont de différents modèles. L'auteur ne précise pas la taille de ces pas ni leur nombre, et la moyenne reste la valeur retenue par l'auteur.

Nous avons effectué des mesures à la fois du test de référence et du test de l'opération sur plusieurs cartes à puce provenant de plusieurs fournisseurs, en utilisant plusieurs CADs (Cherry ST-1044U, FSC Smartcard-Reader USB 2A, GemPC Twin, Omnikey Cardman 2020, Omnikey

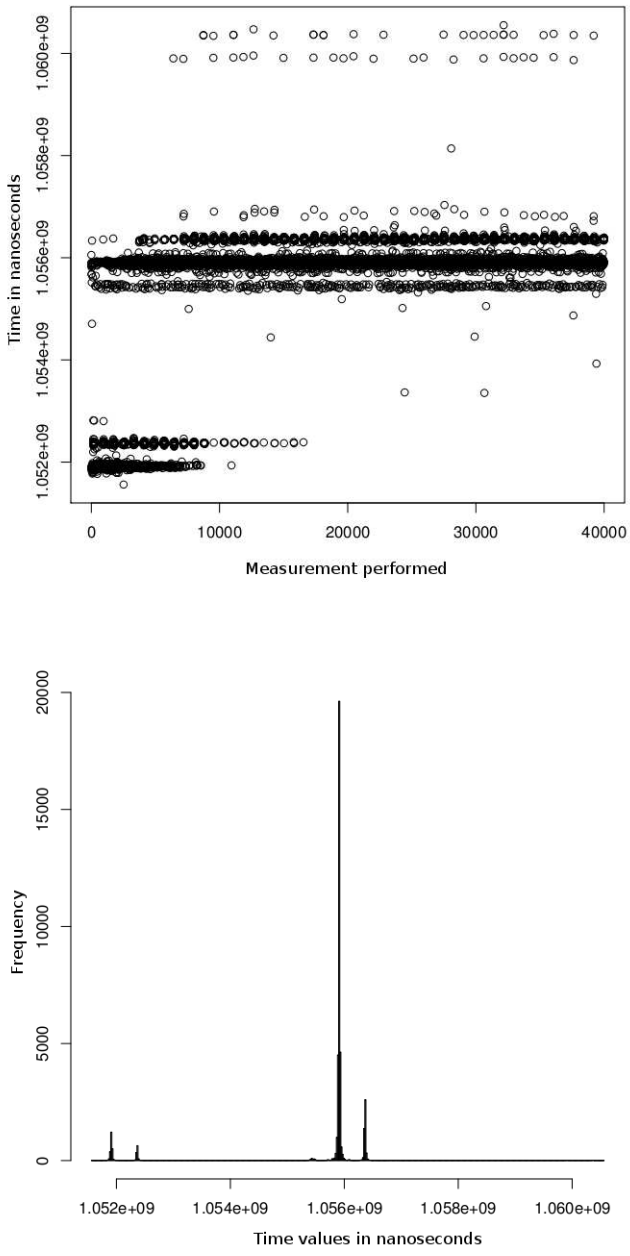


FIG. 5.1 – Valeurs d'une mesure de référence en cours, et la courbe de distribution qui lui correspond $L = 41^2$

```
pcscd.h:#define PCSCLITE_LOCK_POLL_RATE 100000
pcscd.h:#define PCSCLITE_STATUS_POLL_RATE 400000
winscard.c:SYS_USleep(PCSCLITE_LOCK_POLL_RATE);
winscard_clnt.c:SYS_USleep(PCSCLITE_STATUS_POLL_RATE + 10);
ifdwrapper.c:SYS_USleep(100*1000); /* 100 micro s */
```

FIG. 5.2 – Quelques lignes des codes sources extraites de PC/SC Lite et du driver CCID

Cardman 4040, Towitoko Chipdrive Micro, Xiring Teo) et différentes machines hôtes (avec des CPUs AMD Sempron 3100+, AMD X2 3800+, Intel Core2 Quad CPU Q9400), différents OSs (Différentes distributions de Linux basées sur des noyaux 2.6, Windows XP, Windows Vista). Aucune de ces mesures n'a une distribution normale (cf figure 5.1 pour un exemple de mesures d'un test de référence effectuées sur une carte à puce). Ce type de résultat est similaire d'une carte à l'autre en terme de distribution, même pour différentes valeurs temporelles, différentes tailles de boucle. Nous avons aussi changé différentes combinaisons de CAD, de JVM sur la machine hôte, de priorité des processus concernant les services (démons sous Linux) et l'outil logiciel de mesure en éliminant tout processus non vital à la mesure sans que cela n'entraîne de différence sur l'aspect de la courbe de distribution.

Effectuer des mesures sous Linux et sous Windows XP ou Windows Vista, par contre met en évidence des différences dans les courbes de distribution. Le facteur recurent dans les mesures avec un terminal traditionnel est les *pas* entre les pics de distribution. Quand on mesure les performances en utilisant Linux avec PC/SC Lite, un driver CCID, en particulier, on peut observer les distributions obtenues pendant les mesures. Les codes sources de PC/SC Lite et du driver CCID ont l'avantage d'être tous les deux publiés sous licence GPL, ce qui nous permet de relier ces observations aux sources. La figure 5.1 montre les temps obtenus sur un ensemble de mesures. On y voit 40000 mesures effectuées et la distribution de l'ensemble.

Les pics des distributions obtenues dans ces conditions sont de manière consistantes espacées de 4ms pour les pics principaux, des pics secondaires sont à 1ms de ces pics principaux. Ces chiffres correspondent à une partie du code public de PC/SC Lite et du driver CCID (cf figure 5.2). Avec d'autres CADs, les distributions sont similaires. On y voit des pas similaires. Tous les drivers de CAD ne sont pas open source, mais pour ceux qui le sont, on peut utiliser leur code source pour tenter d'analyser les sources d'imprécision. Dans les pics des distributions sous Windows, on trouve des pas de 0.2 ms (cf figure 5.5). Sans avoir accès à l'implémentation de PC/SC sous Windows ni au code source du driver, on peut déduire qu'il y a sans doute des similarités entre les versions propriétaires et open source en termes de temps d'interrogation au niveau logiciel.

Pour vérifier la normalité des distributions des résultats, nous avons utilisé le test de Shapiro-Wilk. Le test de Shapiro-Wilk est un test statistique établi qui est utilisé pour vérifier l'hypothèse nulle qu'un ensemble de données viennent d'une population normalement distribuée [106]. Le résultat d'un tel test se présente sous la forme d'une nombre $W \in [0, 1]$. W est proche de 1 quand les données sont normalement distribuées. Aucun ensemble de valeurs dans les temps mesurés ne nous donne une valeur W proche de 1. Pour les valeurs illustrées dans la figure 5.1, $W = 0.6725$. Sous l'hypothèse nulle (c'est à dire, si la distribution est belle et bien une distribution normale), la *p-value* associée à nos valeurs est $p - value < 2.2E^{-16}$ (ce qui est le minimum affichable par R, notre logiciel de statistiques). Cette dernière valeur est une probabilité (qui est donc négligeable) d'obtenir une valeur au moins aussi extrême avec une distribution normale. Cette valeur représente nous indique donc que si nous avons affaire à une distribution normale, notre

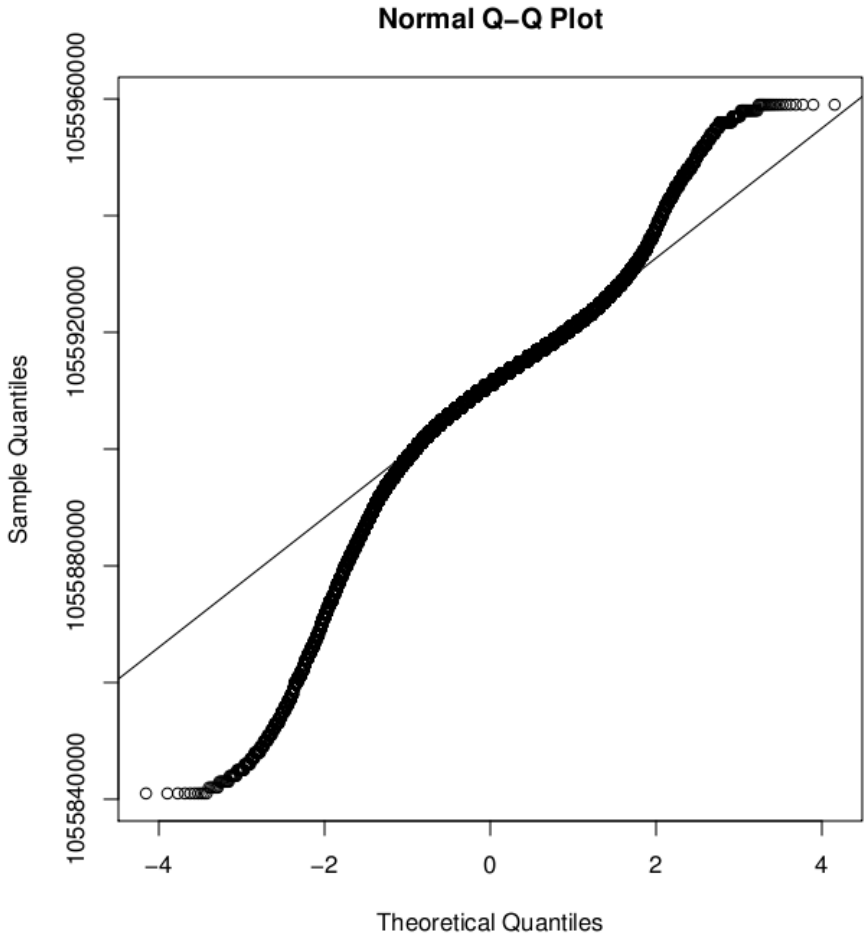


FIG. 5.3 – Droite de Henry représentant les valeurs obtenues dans la figure 5.4

ensemble de mesures est très peu significatif statistiquement. De la valeur de W et de $p-value$, on conclut que notre ensemble de mesures n'est pas normalement distribué.

Non seulement ces mesures ne sont pas normalement distribuées, mais aucun ensemble de mesures faites dans des conditions usuelles n'est normalement distribué. Nous avons également isolé certains pics de forte densité de mesures s'approchant d'une même valeur. Par exemple la figure 5.4 représente un des pics de forte densité de mesures, de valeurs "proches", observées dans la figure 5.1. L'idée est de se restreindre à des valeurs qui sont toutes proches les unes des autres pour vérifier si, sur un sous ensemble de temps mesuré, les valeurs sont normalement distribuées. Nous pouvons déjà observer une courbe en cloche qui pourrait être normale. Nous pouvons aussi observer la droite de Henry qui est obtenue à partir de ce sous ensemble dans la figure 5.3. Cette figure nous permet de visualiser les écarts de nos mesures par rapport à une distribution normale. Pour ce sous ensemble, nous avons obtenu $W = 0.8442$, pour une $p-value$ encore une fois négligeable. Cette valeur de W est la plus élevée que nous ayons rencontrée au cours de nos tests. Certaines autres valeurs sont aussi basses que $W = 0.1384$, ce qui indique que nous n'avons pas de distribution normale. Nous en concluons qu'aucun sous ensemble de valeurs obtenues par mesure standard de nos tests n'est normalement distribué. Cela peut remettre en

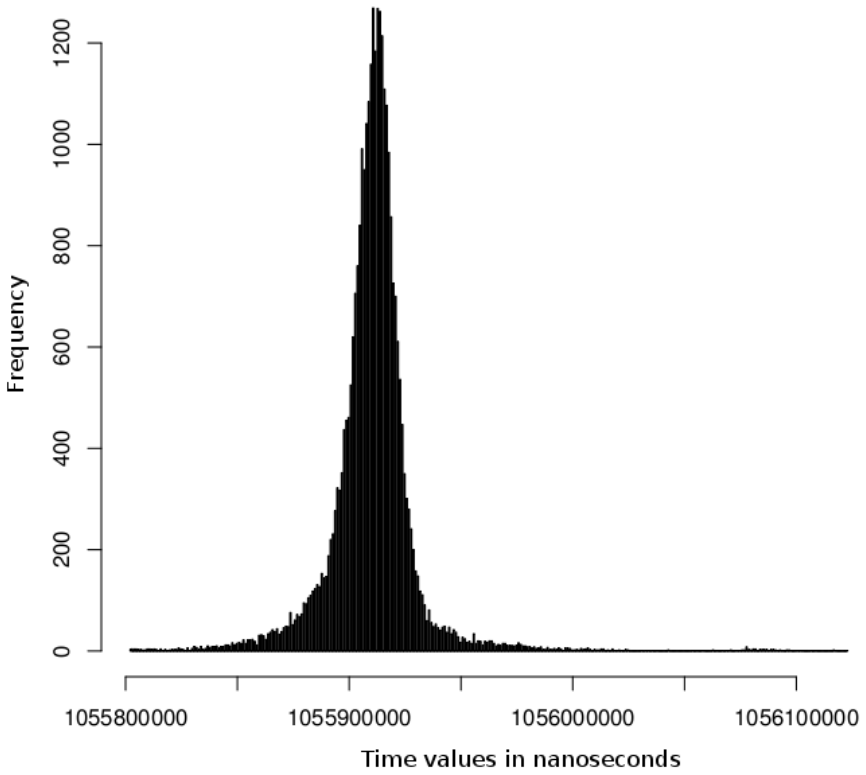


FIG. 5.4 – Distribution des mesures d'un test de référence : recadrage sur un pic de forte densité ($L = 41^2$)

cause l'utilisation de la moyenne arithmétique comme valeur représentative de nos mesures.

Ce qui nous intéresse vraiment n'est pas non plus les mesures brutes des tests, mais les temps isolés qui représentent les temps d'exécution des bytecodes qui nous concernent. Autrement dit, nous sommes plus intéressés par les différences entre les mesures des tests d'opérations et les mesures de référence qui leurs sont associées que par les mesures elles mêmes.

La figure 5.6 montre deux courbes. La courbe supérieure montre les temps mesurés pour des tests d'opérations `sadd`, alors que la courbe inférieure représente les mesures du test de référence correspondant. Chaque courbe est sujette à des variations qui sont dûes à la non normalité des distributions des mesures respectives (c'est à dire aux bruits sur les plates-formes de mesures). Il est difficile pour nous de décider d'une valeur temporelle qui soit représentative de chaque ensemble de mesure. Il y a cependant un écart entre les deux courbes qui est important. En l'occurrence, l'écart est plus important que les variations de chacune des courbes. Cet écart est du à l'exécution d'un `sadd` supplémentaire à chaque itération de la boucle. Pour une taille de boucle suffisamment grande, la différence entre les deux courbes peut être, de plusieurs ordres de grandeur, supérieure aux variations propres à chaque courbe. Le bytecode supplémentaire est alors effectué un nombre suffisamment important de fois pour affecter les performances de la boucle d'opération.

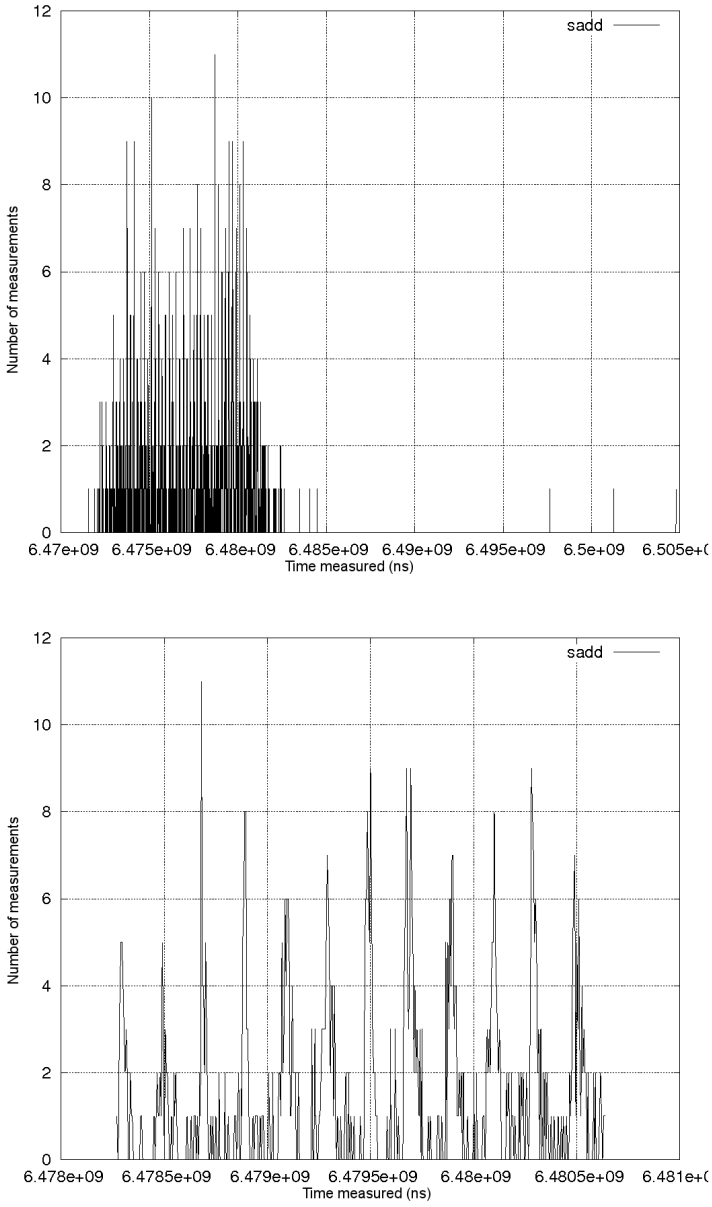


FIG. 5.5 – Distribution d'une mesure de tests d'opération `sadd` sous Windows Vista, et une vue ressérée sur une partie de la courbe ($L = 90^2$)

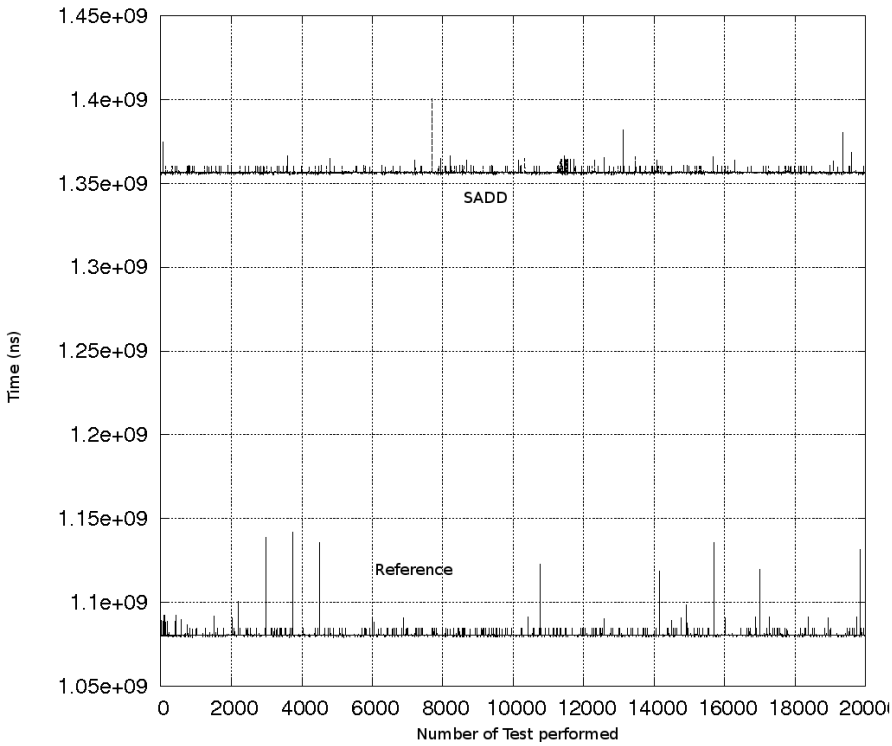


FIG. 5.6 – Comparaison entre les mesures d'opérations `sadd` et les mesures de référence correspondantes ($L = 41^2$)

FIG. 5.7 – Le lecteur de précision MP300 TC1

D'une manière générale, même si nous n'avons pas accès à un ensemble de valeurs normalement distribuées, pour une taille de boucle suffisamment grande, les mesures pourraient être exactes et on pourrait les représenter à l'aide de leur moyenne.

5.2.2 Validation avec un CAD de précision

Nous avons utilisé un lecteur Microprocess MP300 TC1 (cf figure 5.7) pour vérifier la précision et l'exactitude de nos mesures. Cet outil est une plateforme de test de cartes à puce. Elle a été conçue particulièrement pour nous donner des résultats précis, plus particulièrement en termes d'analyse temporelle. Microprocess, le constructeur du MP300 annonce une exactitude de 20 ns. Ce lecteur permet en outre d'espionner les contacts d'une carte. On peut ainsi déterminer les temps nécessaires pour les entrées sorties avec précision, et on peut mesurer les performances de la carte entre les étapes dédiées aux entrées/sorties dans l'application.

Les résultats sur cette plateforme ne sont apparemment pas affectés par les bruits sur la machine hôte. On peut l'utiliser pour mesurer les fronts montants sur les contacts d'une carte à puce, par exemple pour marquer les entrées/sorties. L'avantage le plus évident que l'on a à utiliser cette plateforme est qu'on utilise ainsi un système dédié piloté depuis l'extérieur (nous avons piloté le CAD par réseau).

Nous avons mesuré le temps nécessaire à une carte à puce pour répondre aux mêmes APDUs qui nous avaient déjà servi avec les CAD "ordinaires". Nous avons ensuite testé les valeurs temporelles mesurées avec le test de Shapiro-Wilk. Les valeurs de W que nous avons obtenues de cette manière sont $W \geq 0.96$. Notamment, les courbes de distribution correspondant aux valeurs obtenues avec ce CAD ne présentent pas de multiples pics de distribution (cf figure 5.8). Sur cette dernière figure, $W = 0.9776$ pour une $p - value = 0.08623$. Ces valeurs sont proches de ce que nous attendions de nos mesures. Avec ces valeurs de W, on peut assumer que les valeurs des mesures sont normalement distribuées, et ce, à la fois pour les mesures d'opération et pour les mesures de référence correspondantes.

Nous avons soustrait une valeur de mesure de référence de chaque valeur de mesure d'opération `sadd`, puis divisé par la taille de la boucle pour obtenir des valeurs temporelles qui représentent le temps d'exécution d'un `sadd` isolé. Ces nouvelles valeurs temporelles sont normalement distribuées ($W = 0.9522$, $p - value = 0.02659$). Sur l'ensemble des valeurs obtenues ainsi, la moyenne arithmétique est de 10611.57 ns et l'écart type est de 16.19524. D'après [74], puisqu'on a affaire à une distribution normale, la moyenne arithmétique est une évaluation appropriée du temps nécessaire pour effectuer un bytecode `sadd` sur cette carte à puce.

En utilisant un CAD plus traditionnel (un Cardmann 4040, mais nous avons essayé avec cinq différents CADs), nous avons effectué 1000 mesures de l'opération `sadd` et 1000 mesures du test de référence correspondant. En soustrayant chaque valeur obtenue avec le test de référence de chacune des valeurs obtenues avec le test de l'opération `sadd`, et en divisant par la taille de boucle, on obtient un ensemble de 1000000 de valeurs temporelles. Ce nouvel ensemble a une moyenne arithmétique de 10260.65 ns et un écart type de 52.46025.

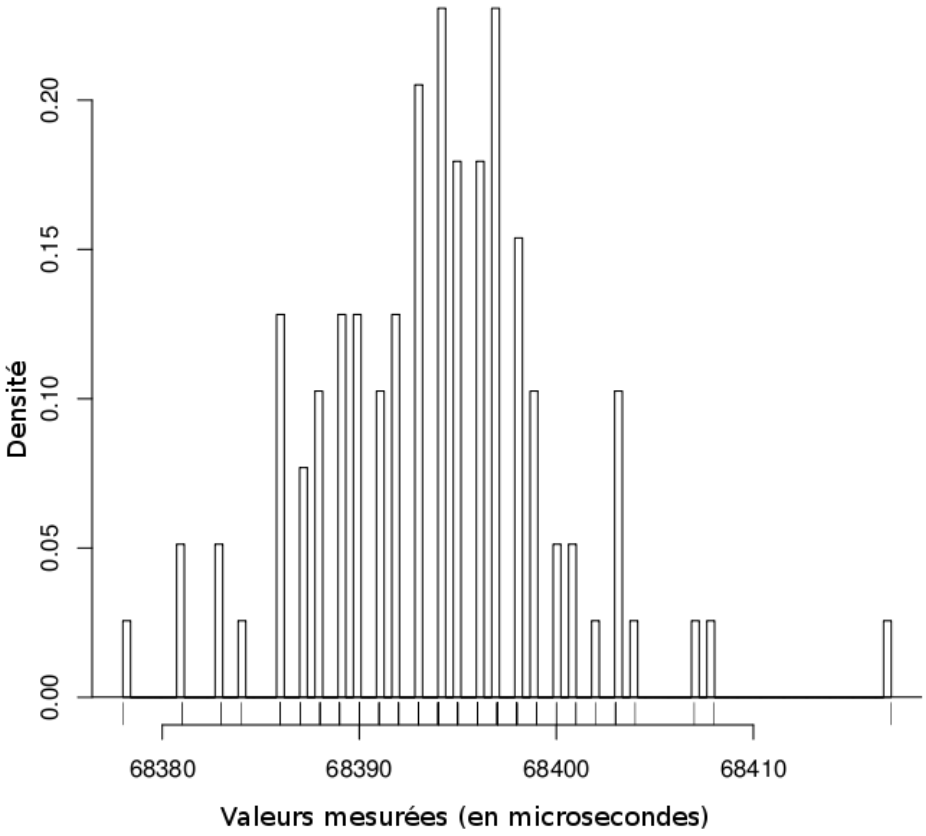


FIG. 5.8 – Distribution des mesures de l'opération `sadd` avec le CAD MP300 ($L = \texttt{P2}^2 = 1024$)

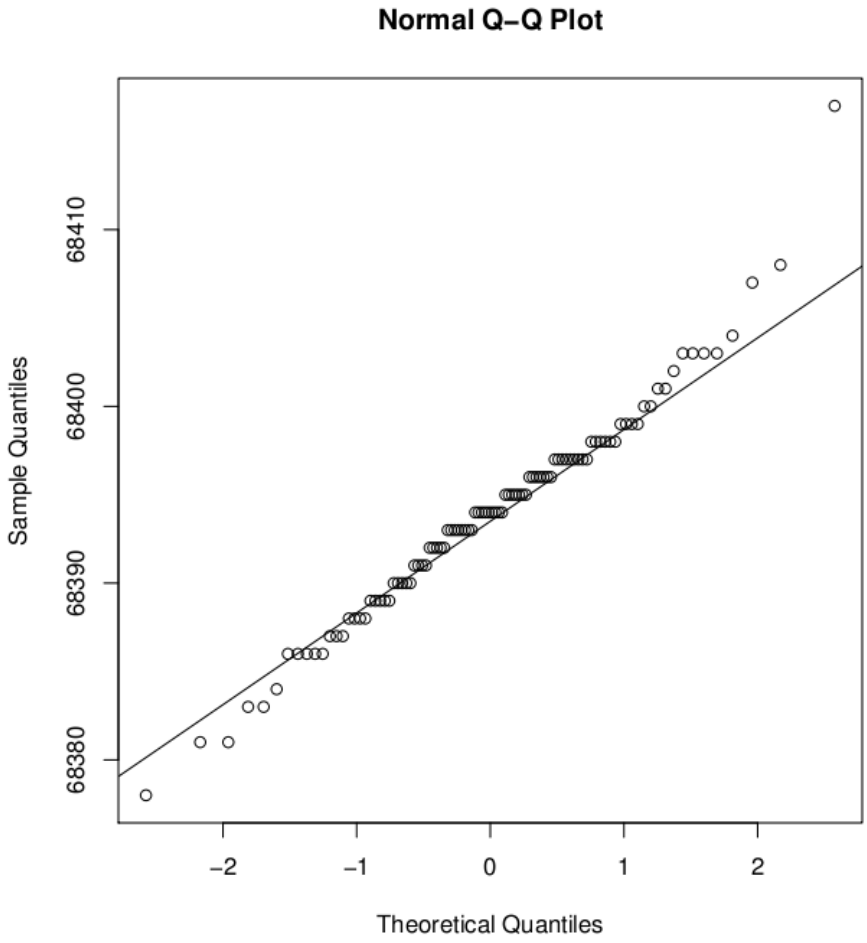


FIG. 5.9 – Droite de Henry des mesures de l'opération `sadd` avec le CAD MP300 ($L = \texttt{P2}^2 = 1024$)

La valeur obtenue en utilisant un CAD ordinaire sous Linux sans modification de priorité dévie de 3.42% de la valeur plus exacte obtenue avec le lecteur de précision. Même si les valeurs ne sont pas normalement distribués ($W = 0.2432$), la moyenne arithmétique de nos mesures expérimentales bruitées semble être une bonne approximation des temps nécessaires à la carte à puce pour exécuter un bytecode `sadd`.

Les mêmes tests sous Windows Vista nous donnent une moyenne de 11380.83 ns avec un écart type 100.7473, ce qui dévie de 7.24 % de la valeur plus exacte obtenue avec le lecteur de précision.

Il est à noter que ce type d'exactitude est celui qui est attendu avec des benchmarks commerciaux dans le cadre de la mesure de performance d'un PC. En particulier LINPACK [28], Dhrystone [121] montrent des signes de dispersions des mesures similaires. Il faut aussi remarquer que le lecteur de précision offre une exactitude à 20 ns près et que les mesures sous Linux dans un environnement bruité dévient d'environ 400 ns par rapport à ces mesures non bruitées.

En conclusion, nos données sont bruitées, mais, malgré un environnement de test potentiellement très bruité, nos mesures gardent une certaine exactitude et une certaine précision.

5.3 Conclusion

L'objectif de MESURE [78] est de permettre aux utilisateurs, notamment aux acteurs de l'industrie de la carte à puce, de mesurer les performances des cartes à puce sans nécessairement disposer d'un environnement de test élaboré coûteux et spécialisé.

Les résultats illustrés dans ce chapitre tendent à montrer que les mesures obtenues sont nécessairement bruitées mais que ces bruits sont contournables par des tailles de boucles et un nombre de mesures important.

La distribution des mesures observées n'est pas normale, mais elle se rapproche d'une distribution normale obtenue sur un lecteur plus précis et plus exact.

Les niveaux de précision et d'exactitude obtenus semblent relativement satisfaisants au regard de résultats obtenus avec des benchmarks existants sur PC. Mais l'exactitude est sans doute insuffisante pour pratiquer des attaques par canaux cachés sur les cartes à puce. Cette inexactitude est un argument plutôt en faveur de l'adoption de ce benchmark comme standard de la mesure de performance sur carte à puce que le contraire. Pour les industriels concernés, le fait d'avoir un outil de mesure disponible et relativement fiable est quelque chose de satisfaisant, mais le fait d'avoir à disposition un outil qui puisse potentiellement être utilisé pour casser la sécurité des cartes à puce est quelque chose de peu enviable. La relative inexactitude des résultats est de ce point de vue un aspect assez positif, tout en s'assurant que les mesures sont éloignées de la réalité, mais d'une manière raisonnable.

Une des conséquence de ces résultats est qu'un grand nombre de mesures pour une grande taille de boucles peut être utilisé pour atteindre une exactitude et surtout une précision potentiellement meilleures, mais que ce n'est pas nécessaire dans le cadre de benchmarking d'une carte à puce dans son sens classique : pour obtenir une note de synthèse sur les capacités techniques d'une carte. Aussi, même si l'étape de calibration de l'environnement de mesure permet de fixer des tailles de boucles pas trop grandes mais assez fiables, elle peut être configurée par l'utilisateur en utilisant le coefficient de variation pour obtenir des résultats plus appropriés à ses besoins.

6

Application

Sommaire

6.1 Introduction

6.1.1 Représentativité des mesures et domaines d'applications

Un des gros problèmes dans la mesure de performance est la pertinence de ce qui est mesuré. C'est un problème qui est ancien (**LINPACK** [28] [32] souffrait déjà de ce problème dans les années 1970). En règle générale, les solutions proposées consistent à rapprocher les outils de mesure de performance des applications "réellement utilisées" par les utilisateurs. Par exemple **SPEC CPU2006** [111] propose d'utiliser de manière intensive des programmes gourmands en un type de calcul. **CINT**, qui couvre la partie calcul sur des entiers, va donc regrouper des programmes de compression, de compilation, d'intelligence artificielle, de recherche de séquence dans des protéines et autres (cf 2.2.3). Alors que **SPECviewperf** va faire un usage intensif de systèmes graphiques OpenGL en reprenant des tâches de rendu provenant d'applications réelles.

Même si l'on parvient à mesurer avec précision des éléments qui font partie de la performance d'une carte à puce, il nous faut relier ces mesures à quelque chose de significatif vis à vis de l'usage qui peut en être fait.

La solution proposée dans le projet MESURE a consisté à utiliser des applications réelles provenant de domaines d'applications traditionnels pour les cartes à puce (bancaire, identité, transport ...). Pour chacune de ces applications, nous avons fourni un profile d'utilisation avec des coefficients pour chaque bytecode et chaque entrée de l'API. Ces coefficients correspondent à l'usage qui est fait de ces éléments dans l'utilisation quotidienne que le porteur a de ces applications.

Nous souhaiterions étudier la performance d'une carte à puce dans un domaine très éloigné de ceux qui sont traditionnels pour celles-ci. D'une part, cela permettrait d'observer la performance d'une carte à puce dans un contexte précis et de créer un profil pour ce domaine. On peut retrouver les performances perçues sur cette carte à puce en concevant un scénario d'exécution réaliste, en retrouvant tous les bytecodes et tous les appels de méthode de l'API effectués, en utilisant MESURE pour retrouver les performances individuelles de toutes ces entités et en recoupant tous ces temps d'exécution individuels pour estimer le temps global de l'exécution. Cela aurait pour première conséquence de montrer l'utilisabilité d'une application dans son contexte.

Ensuite, il serait souhaitable de comparer les résultats individuels obtenus avec MESURE et une mesure de bout en bout de l'application, c'est à dire de mesurer simplement l'exécution d'une commande sans essayer d'isoler le bruit. Cela permettrait de valider les performances obtenues par MESURE.

Les jeux multi-joueurs ubiquitaires (JMUs) pourraient bien être le futur de l'industrie du jeu vidéo. Dans ce type de jeu, les utilisateurs jouent simultanément dans le monde réel et dans le monde virtuel [8]. Pour gérer un système de JMU qui supporte les interactions des joueurs connectés à la fois dans le monde physique et le monde réel, il faut un système qui permette aux joueurs d'embarquer de l'équipement qui doit être déployé dans le monde réel et qui calcule l'état global du monde virtuel.

L'idée générale de ce chapitre est, dans un premier temps d'utiliser les cartes à puce comme support décentralisé pour jouer aux JMUs, puis d'utiliser les connaissances que nous avons sur les performances des cartes à puce pour évaluer l'impact des cartes sur le jeu en terme de jouabilité.

6.1.2 JMUs et plates-formes

Pour augmenter la mobilité et l'ubiquité dans les JMUs, on se concentre ici, sur une approche centrée sur l'utilisateur. Cela peut donner lieu à de nouveaux types d'interaction pour l'utilisateur.

Diverses technologies, telles que les tags RFID, les objets en réseau ou senseurs environnementaux peuvent être utilisés pour aider les utilisateurs à interagir avec l'environnement physique. De plus le joueur lui même peut détenir différents éléments comme un terminal embarqué (par exemple un téléphone), des senseurs biomédicaux ou des lunettes de réalité virtuelle.

Différents types de connexions sont utilisés pour relier tous ces appareils : Wi-Fi [122] , Bluetooth [112] , ZigBee [113] ou le réseau des téléphones cellulaires. Enfin, un serveur de JMU est utilisé pour exécuter la logique du jeu, centraliser les données de jeu et relier les joueurs.

Une manière appropriée de supporter cette hétérogénéïté technologique est d'utiliser un intergiciel comme uGASP [94] [92], qui est un logiciel libre basé sur OSGi [87] dédié aux JMUs.

D'un point de vue jouabilité, les systèmes de JMUs introduisent le concept "interaction du système de Jeu dans le monde Réel" (IJR). Ce concept est basé sur les propriétés suivantes. Tout d'abord, la jouabilité repose sur la mobilité physique du joueur, et, souvent, il nécessite un contexte et une adaptation de l'utilisateur. Deuxièmement, le jeu interagit avec le joueur d'une manière ubiquitaire (à des endroits non dédiés et à travers des objets non dédiés) et proactive (à des moments non contrôlés, par exemple par e-mail ou par téléphone). Finalement, le jeu amène à des interactions sociales qui peuvent être effectives dans le monde réel ou dans le monde virtuel.

Ainsi, les systèmes de JMUs doivent être suffisamment flexibles pour être capables de répondre à ces relations complexes et incertaines entre le joueur et le monde réel. De plus, le joueur doit être capable d'interagir avec le jeu malgré un réseau éventuellement déconnecté, par exemple pour interagir avec un jouet intelligent dans une zone hors réseau.

Au niveau de la conception de niveaux, comme tout jeu, et plus généralement comme toute application de loisir, un système de JMU devrait inclure un modèle d'utilisateur. Un système de JMU peut être vu comme un système d'information nécessitant des données personnelles de l'utilisateur pour intégrer la vie réelle de l'utilisateur dans le jeu, par exemple son numéro de téléphone, ses relations dans la vie sociale. Yan [83] propose un modèle de profil utilisateur pour personnaliser l'expérience de jeu.

Une des manières pour enregistrer le profil du joueur dans un système de JMU est de fournir aux joueurs des appareils embarquant de l'électronique, comme des cartes à puce NFC. Les cartes à puce NFC (Near Field Communication - Communication en champ proche) sont une partie de la famille des cartes à puce. Ce sont donc des appareils cryptographiques assez répandus avec une capacité de stockage avec des propriétés de résistance aux attaques et avec une capacité de communication par NFC. Par nature, les cartes à puce ne possédant pas de source d'alimentation électrique interne, les utiliser nécessite également l'emploi des lecteurs de cartes (CAD) capables de les alimenter et d'interagir avec elles. L'avantage de la technologie NFC, dans ce contexte est de permettre une communication sans contact, essentiellement avec des téléphones portables.

Même si l'utilisation d'un profil utilisateur est courante sur une carte à puce (par exemple dans les cartes de santé de certains pays), il n'existe pas de tentative publique utilisant des cartes à puce pour gérer des profils de joueurs jusque là. D'autre part de nombreux systèmes de jeux tendent à sous-estimer les problèmes de confidentialité et de sécurité qui devraient être considérés dans un environnement en réseau impliquant des données personnelles.

Une partie de ces travaux a été entreprise à l'origine dans le cadre du concours Simagine 2007 [41]. Le concours Simagine est proposé tous les ans par Gemalto, Samsung et Sun. Des projets autour de Java Card et des cartes SIMs sont proposés par les participants de ce concours pour éventuellement gagner des prix. Ce travail a pu être repris au sein du projet PLUG [96].

PLUG est un projet mené par le CNAM-CEDRIC en collaboration avec le Musée des Arts et Métiers, Orange Labs, l'Institut Sud Telecom, L3i lab de l'université de La Rochelle, et des studios de jeu : TetraEdge, Net Innovation et Dune. L'objectif de PLUG est de créer un JMU dans le musée des Arts et Métiers qui prenne en compte les caractéristiques des joueurs. Le JMU

est basé sur l'intergiciel uGASP [94].

Notre approche consiste à utiliser une carte à puce NFC pour stocker le profil du joueur, contribuant ainsi à la mobilité et garantissant l'anonymat de l'utilisateur. L'utilisateur doit détenir des informations ludiques pour interagir avec les objets NFC de son entourage. De plus, la carte à puce permet au joueur de communiquer de manière sécurisée pour manipuler des données confidentielles. Dans ce chapitre, nous présentons un logiciel libre pour gérer les profils de joueurs de JMUs (PJJMU) en utilisant des objets basés sur du Java (aux niveaux des cartes à puce, des lecteurs et des serveurs) : l'API PJJMU (l'API pour gérer notre PJJMU). Dans un premier temps, nous allons décrire le PJJMU. Puis nous présenterons les technologies qui sont utilisées pour gérer les profils sur les cartes à puce. Nous allons ensuite discuter les bénéfices d'un profil de joueur sur une carte à puce NFC pour les JMUs et les types d'interactions que ça peut apporter au joueur et au système de JMU. La section suivante présente l'architecture générale du système et discute des idées liées à la sécurité. Une section suivante décrit les nouveaux types d'interaction impliqués par notre API. Nous parlerons enfin de la performance de cartes à puces que nous avons utilisées et en quoi elles impactent la jouabilité du système.

6.2 Définition du profil utilisateur pour JMUs

L'essence de la jouabilité tient au processus de conception du jeu. Celui-ci doit considérer le point de vue du joueur. Ce point de vue est codé implicitement ou explicitement dans le système de jeu. Tous les jeux et toutes les applications de divertissement incluent un modèle utilisateur. Dans les jeux à joueur unique, il y a au moins une classification générale du joueur cible, et une mémoire limitée des actions du joueur dans le jeu, mais cela peut aller jusqu'à un modèle cognitif complexe. Dans les jeux multi-joueurs, le modèle doit être cognitif, social, et lié au passé et à la situation courante du joueur à la fois dans le monde virtuel et dans le monde réel.

Si on considère que l'espace d'activité du joueur inclut des outils électroniques et que les systèmes d'information sont de plus en plus pervasifs et ubiquitaires, il y a un besoin de considérer l'interaction entre le monde réel et le monde virtuel dans un mode de réalité mixte, et les actions possibles des joueurs dans les deux univers. Ainsi, le modèle utilisateur ne prendra pas simplement en compte l'état et le comportement de l'utilisateur comme dans les situations de jeu en ligne, mais aussi dans les environnements extérieurs de jeux mobiles.

La méthode proposée ici, consiste à utiliser un modèle utilisateur explicite, le profil de joueur de JMU (PJJMU), pour rassembler et classifier les informations sur le joueur. Cette information sera la base de déduction du mécanisme de décision du jeu.

Le PJJMU guide le moteur de décision du jeu pour offrir différentes expériences aux joueurs. Les quêtes du jeu sont adaptées au scénario du contexte personnel du joueur, ce qui mène à une action qui est exécutée à la fois dans le jeu et dans le monde réel. Le but principal de l'utilisation du PJJMU en relation avec la génération automatique de narration est de décider quel type de quête est le plus adapté au profil de l'utilisateur et aux besoins globaux de narration, de manière à promouvoir les relations sociales entre les joueurs. De cette manière, la jouabilité du jeu est augmentée : le jeu est persistant et adaptable. Chaque joueur peut avoir une expérience unique.

Le PJJMU dépend d'un ensemble de paramètres qui peuvent être soit définis statiquement par le développeur de jeu soit ajustés dynamiquement en relation avec les changements en temps réel dans l'état physique du joueur ou dans ses caractéristiques sociales. Cela implique un niveau de paramètres personnalisés dans le modèle utilisateur [83]. Puisque le joueur est représenté à la fois dans le monde réel et dans le système virtuel, nous devons considérer ses connaissances du jeu suivant plusieurs points de vue. Il est très utile de distinguer les informations générales

de l'utilisateur de ses données propres au jeu, comme son profil général pourrait être ré-exploité par différents mécanismes de jeu. Les trois suivants décrivent les types d'information liées à l'utilisateur qui sont recueillies et identifiées.

- Le premier groupe inclut des données sur l'utilisateur "par lui-même", c'est à dire non liées à son utilisation du jeu comme son statut civil, ses préférences ... La plupart de ces données ne peuvent qu'être fournies par l'utilisateur lui-même lors de la création du compte de l'utilisateur dans le jeu. Puisque ces données changent de manière infréquente, elles doivent être accessibles par tout JMU de manière à n'enregistrer ces informations qu'une seule fois.
- Le second groupe rassemble les connaissances sur l'utilisateur définies comme "en tant que joueur". Il contient quelques informations exactes correspondant aux choix de base du joueur : type de compte, durée de jeu à chaque endroit ... Cela comprend aussi des données statistiques et des données temps réelles collectées durant le jeu comme des informations de localisation, ses interactions avec les différents appareils dans l'environnement réel ...
- Le troisième groupe définit le statut de l'avatar du joueur dans le jeu d'un point de vue à la fois statistique et temps réel, avec des données comme les informations standards de l'avatar, son équipement et inventaire, ou ses relations sociales dans le jeu. Ces données "comme avatar" pourraient être utilisées par le serveur du jeu pour proposer des événements de jeu spécialement adaptés au joueur, comme des quêtes spécifiques communes nécessitant un objet particulier dans l'inventaire de deux joueurs.

Ce modèle d'utilisateur a été expérimenté dans le prototype MugNSRC [123]. Le jeu original, NSRC, est basé sur des courses de chaises de bureau dans les bureaux d'une compagnie virtuelle japonaise.

MugNSRC utilise ce contexte et intègre un modèle utilisateur avec le profil de motivation du joueur dans le moteur de jeu pour gérer et développer une communauté à travers des buts coopératifs et compétitifs assignés aux joueurs.

La question est de savoir quel appareil peut être utilisé pour enregistrer le profil utilisateur dans le jeu. En général, les jeux multi-joueurs suivent une architecture client/serveur. Dans un tel cas, le profil utilisateur est géré par le serveur, comme dans MugNSRC. Les valeurs initiales de chaque classe de données dans le PJJMU sont obtenues par un questionnaire. Ces valeurs peuvent être changées grâce à un système de retour d'information des choix et des actions faits par le joueur au cours du jeu. L'utilisateur peut se connecter à son compte pour obtenir son profil. Les jeux multi-joueurs pair à pair, cette fois, gèrent le profil du joueur dans la partie cliente. Les désavantages sont donc que le joueur doit gérer lui-même son compte quand il change de terminal, et qu'il y a un potentiel de triche plus élevé.

6.3 L'utilisation de cartes à puce dans la gestion de profil utilisateur

En raison de potentielles limites de la couverture du réseau et aussi parce que les connexions au réseau sont par essence non fiables, une approche intéressante pour que le jeu puisse se dérouler de manière continue est de permettre au joueur d'embarquer son profil avec lui afin qu'il puisse jouer de manière déconnectée.

On devrait donc construire un système d'information distribué pour les données du jeu, en particulier pour les informations du PJJMU. Pour gérer ces informations, des appareils électroniques portables sont appropriés. Une liste de ces appareils inclut des téléphones portables, des PDA, des cartes à puce, des consoles de jeu portables, des cartes mémoire, des memory spots

[6] ... Dans cette liste, les cartes à puce offrent un bon compromis en termes de portabilité, de mécanismes de sécurité et de coût.

La facilité d'interaction étant un élément essentiel dans le cadre d'un jeu, les cartes à puce sans contact sont sans doute plus adaptées ici. Cela a pour mérite d'améliorer certains aspects de l'interface homme machine du jeu (on n'a plus besoin d'insérer la carte dans un lecteur).

Dans le contexte d'un système ubiquitaire, l'utilisateur peut soit porter une carte à puce NFC qui est lisible à courte portée par un lecteur NFC, soit porter un lecteur qui doit pouvoir interagir avec les objets NFC disséminés dans une zone.

Il n'y a à notre connaissance pas d'autre projet de JMU qui utilise des cartes à puce pour gérer les profils utilisateurs. Cependant, il y a des similarités entre le fait d'utiliser une carte à puce pour un JMU et l'utiliser pour une application commerciale comme dans les domaines bancaires, dans les transports ou dans les télécommunications. À ce titre, de nombreuses villes dans le monde utilisent des systèmes basés sur les cartes à puce sans contact pour gérer les transports publics. Par exemple, les usagers des transports publics parisiens peuvent utiliser leurs cartes à puce sans contact *Navigo* comme moyen pour accéder au réseau ferré ainsi qu'au réseau de bicyclettes publiques (*Velib*). Dans ce dernier exemple, des stations de bicyclettes sont équipées de lecteurs NFC et sont reliées à une autorité centrale. Les cartes à puce sont utilisées pour garder des données liées à l'utilisateur, comme la liste des stations dans lesquelles l'utilisateur est passé.

Le cœur de ce type de système est l'embarquement dans les cartes à puce de données propres à l'utilisateur tout en incorporant des capacités cryptographiques.

Certains travaux ont visé la gestion de profils de santé avec PicoDBMS [98]. PicoDBMS est un système de gestion de base de données dédié aux cartes à puce. PicoDBMS a été utilisé entre autres lors de travaux par Lahlou et Urien [71] pour filtrer des données internet à travers un profil d'utilisateur sur carte à puce. Le profil de l'utilisateur est dynamique (pour que l'utilisateur puisse spécifier et changer ses préférences).

Concernant la sécurité, les auteurs ont utilisé l'approche du groupe de normalisation P3P (Platform for Privacy Preferences) [88]. On y trouve deux niveaux de sécurité, le moins sécurisé étant moins intensif au niveau de la carte à puce. Ce travail n'aborde pas les notions de jeu ou d'ubiquité, et il n'est pas fait mention d'authentification ou de confidentialité des informations. Les systèmes ubiquitaires devraient donc introduire un intergiciel inspiré de celui détaillé dans ces travaux pour supporter un système d'information distribué. Il y aurait trois composants essentiels à ces systèmes :

- les utilisateurs et leurs cartes à puce,
- les lecteurs,
- une autorité centrale.

6.4 Jouer aux JMUs avec des cartes à puce NFC

Les systèmes de jeu de quelques JMUs existants, tels que [63] [109] [79], reposent sur la capacité à contrôler tous les objets physiques qui sont intégrés dans le jeu, leurs impacts sur le joueur et les senseurs disséminés dans le monde réel. Les participants de JMUs ont souvent affaire à un matériel difficilement transportable et à des problèmes de connexions au réseau [17]. Une carte à puce pourrait être utilisée comme interface pour les interactions entre le joueur, le monde réel et le monde virtuel.

Du côté du joueur, le PJJMU peut être spécifié sur la carte à puce, ce qui donne accès au joueur à ses informations liées au jeu. Le joueur peut observer ses parties en cours, gérer les

objets liés aux jeux, et même visualiser ou être informé de la progression du jeu soit en utilisant un des terminaux fixes disséminés sur la zone de jeu, soit en utilisant un terminal mobile. Dans le contexte de l'utilisation d'un profil sur une carte à puce NFC, le joueur pourrait donc utiliser sa carte à puce avec un téléphone mobile intégrant un lecteur NFC.

Les mises à jour du profil sont exécutées automatiquement par le système et manuellement par le joueur. D'abord, le PJJMU doit pouvoir être changé lors des interactions physiques et des déplacements du joueur dans l'environnement réel. Comme l'environnement réel est parsemé d'objets interactifs, la localisation physique du joueur peut être pistée lors de ses déplacements dans les zones de jeu. L'interaction entre la carte à puce et les objets environnants peut être faite à travers des lecteurs NFC et ne nécessitent pas de connexion avec le serveur de jeu. À chaque fois qu'un joueur se rapproche suffisamment d'un lecteur, des informations peuvent être mises à jour et utilisées en utilisant les données "comme joueur".

Deuxièmement, le PJJMU peut être mis à jour à la suite de communications et d'interactions sociales entre joueurs dans le monde réel. Les joueurs doivent être capables de vendre et d'acheter des objets propres au jeu en leur possession à d'autres joueurs alors même qu'ils sont hors ligne.

Le troisième groupe d'informations, c'est à dire les données "comme avatar" peuvent être mises à jour dynamiquement.

La dimension sociale du jeu est entendue dans des dimensions spatiales et temporelles. Ainsi, le système de jeu peut déclencher et contrôler des événements de jeu en temps réel et dans l'espace du monde réel pour un groupe de joueurs dans une même zone de jeu. Le PJJMU doit alors être mis à jour durant les interactions en temps réel dans le jeu et dans l'espace physique.

Jouer à un JMU avec une carte à puce est une expérience nouvelle pour l'utilisateur, ce qui amène de nouvelles formes d'interactions aux joueurs, de nouveaux contenus et de nouvelles problématiques de sécurité. Une interaction automatisée entre la carte à puce NFC et le lecteur NFC peut prendre place simplement en les rapprochant l'un de l'autre. Pour l'utilisateur, le terminal le plus familier accessible et d'un coût raisonnable est le téléphone mobile. Certains téléphones sont par ailleurs capables d'intégrer un lecteur NFC comme le *Nokia 6131 NFC* et le *Sagem My700x*. Ainsi, on va pouvoir développer une application cliente d'un JMU pour téléphone mobile.

Dans l'idéal, un JMU est constitué d'un environnement numérique où des objets intelligents entourent le joueur. On pourrait imaginer embarquer un lecteur NFC dans des objets comme des Nabaztag [82], par exemple. On pourrait aussi créer des jeux ou l'utilisateur se sert de sa carte à puce avec un décodeur de télévision pour accéder à du contenu multimédia.

Sur la carte à puce, nous devons définir et formaliser le PJJMU. Considérant les aspects sécuritaires, la spécification du profil comme étant séparée du serveur doit garantir la confidentialité des informations personnelles de chaque individu. Dans 'World of Warcraft"(Blizzard Entertainment, 2004), l'utilisateur peut enregistrer son compte en banque dans le serveur du jeu, ce qui est potentiellement dangereux malgré des systèmes de protection (nom d'utilisateur/mot de passe), pour obtenir des services spéciaux de l'éditeur de jeu. On pourrait aussi envisager d'inclure des informations biométriques comme moyen d'identification du joueur.

En conséquence, il y a un besoin de développer une infrastructure pour utiliser les cartes à puce NFC dans l'architecture d'un JMU.

6.5 L'architecture pour gérer le PJJMU sur une carte à puce NFC

Les interactions NFC 6.4 dans les JMUs et le profil du joueur sont à la base de l'architecture du PJJMU. Le composant principal de cette architecture est le service qui gère le profil de joueur de JMU sur une carte à puce NFC. Nous avons implémenté une librairie qui permet à des téléphones basés sur *Java 2 Micro Edition* [55] (J2ME) *Mobile Information Device Profile* (MIDP) d'échanger des données avec les cartes à puce dans la logique du serveur de jeu. Le serveur lui-même est implémenté en J2SE et la partie carte à puce de l'application est une applet Java Card. Enfin, nous avons utilisé des mécanismes de sécurité pour garantir le secret des données du joueur.

La figure 6.1 présente un schéma de l'architecture de PJJMU.

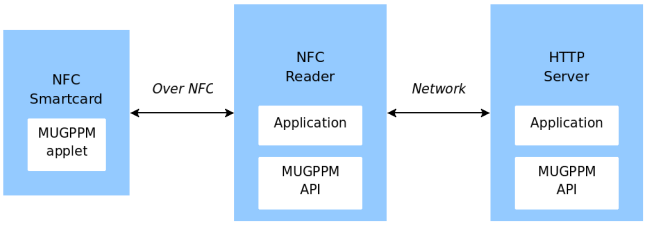


FIG. 6.1 – Vue générale de l'architecture du PJJMU

6.5.1 Le service - partie sur la carte à puce

La partie implémentée sur la carte à puce est une application écrite en Java Card.

L'applet Java est dédiée au PJJMU. Elle est conçue pour recevoir quelques commandes APDUs.

Les instructions APDU

Le jeu d'instructions APDU utilisé dans le PJJMU permet de gérer :

- les entrées présentes quelque soit le jeu, comme le nom du joueur, son âge ou son temps de jeu
- les objets liés au jeu (l'inventaire)
- les clés.

Les objets peuvent être définis comme échangeables entre joueurs. C'est au concepteur du JMU de décider si un objet du jeu peut être échangeable ou non. La table 6.5.1 montre les instructions utilisées par le PJJMU. Les détails des paramètres de chaque instruction et la réponse correspondante de la carte à puce y sont détaillés.

Le modèle de données

La taille des champs doit être bornée pour cause de la quantité de mémoire disponible. Nous avons testé notre implémentation avec une carte à puce qui nous offre 72 ko de mémoire EEPROM.

Le profil lui même n'est pas vraiment encombrant, puisque le fichier CAP contenant notre application a une taille de 6 ko. Nous utilisons 4 ko de mémoire tampon pour les entrées sorties larges (par exemple pour chiffrer des données). Les champs de la classe `GameProfile` incluent un certain nombre de tableaux d'octets (264 octets) et deux objets `OwnerPIN` pour gérer les mots de passe de l'utilisateur. La taille d'un de ces objet dépend de l'implémentation de la machine

Instruction	P1	P2	Données	Retourne
CREATE_PROFILE			login+pwd	status
LOGIN_PROFILE			login+pwd	status
REINIT_PROFILE			login+pwd	status
DELETE_PROFILE			login+pwd	status
LOAD_DEFAULT_ENTRY	key			data
UPDATE_DEFAULT_ENTRY	key		data	status
LOAD_OBJECT_ENTRIES				data
LOAD_OBJECT_ENTRY	key			data
ADD_OBJECT_ENTRY	key	isShareable	data	status
DELETE_OBJECT_ENTRY				status

TAB. 6.1 – APDU instructions utilisées dans le PJJMU

virtuelle Java Card, alors que le mot de passe lui même est limité à 8 octets. De plus on a inclus dans le profil trois clés RSA de 2048 bits (768 octets). L'application est générique et elle peut servir pour plusieurs jeux avec une instance de profil pour chaque jeu. L'application elle-même doit donc occuper 8 ko et chaque instance de profil (un par jeu) doit nécessiter environ 2 ko (en fonction de la taille de `OwnerPIN`). En conséquence, on peut utiliser 30 différents profils de jeu la carte à puce testée.

6.5.2 Le service - partie lecteur NFC

Les fonctionnalités principales de l'API du lecteur NFC sont d'accéder au PJJMU enregistré sur une carte à puce et de communiquer avec les services du serveur de JMU.

Une classe `APDUDataManager` est utilisée pour établir les communications NFC vers la carte à puce et pour formater des commandes APDU. Une classe `GameProfile` est utilisée pour gérer les champs du profil. Enfin, une classe `NetworkCom` se charge des communications vers le serveur en utilisant le protocole MooDS [93].

Nous avons prototypé une version J2ME pour notre service de PJJMU pour qu'un téléphone portable accède au service PJJMU sur la carte à puce. Le choix d'un téléphone portable comme terminal à été relativement facile à faire étant donné la familiarité des téléphones pour les utilisateurs potentiels.

Par ailleurs, des téléphones mobiles incorporant un lecteur NFC sont récemment apparus sur le marché, dont le *Nokia 6131 NFC* ou le *Sagem my700X* (2007). En 2007, une API pour établir la connexion sans contact entre un téléphone mobile J2ME et une carte à puce NFC à été publiés : *JSR257* [65].

Une API spécifique est traditionnellement utilisée pour communiquer à l'aide d'APDUs à partir de téléphones J2ME : *JSR177* [64]. L'usage de cette API n'est cependant pas indispensable dans le cadre d'une carte à puce NFC. Les mécanismes offerts par cette dernière API sont plutôt utilisés pour communiquer avec une carte SIM. Ainsi, on peut n'utiliser que les fonctionnalités de l'API JSR257 pour faire communiquer le téléphone et la carte à puce.

Pour utiliser l'API du gestionnaire de PJJMU, la première étape est la création par le joueur de son PJJMU sur la carte à puce. Il doit créer ses noms d'utilisateurs et mots de passe qui vont être utilisés pour accéder à son profil. Ensuite, le joueur peut utiliser son profil avec le JMU. La figure 6.2 résume l'architecture utilisée pour concevoir un JMU avec une gestion du profil du joueur sur carte à puce.

Les interactions entre le téléphone et la carte à puce dépendent de l'utilisateur puisque ce dernier doit rapprocher la carte du téléphone durant certains moments du jeu, par exemple lors d'une sauvegarde du jeu. Le concepteur de jeu doit donc prendre ce paramètre en compte comme une interaction homme machine spécifique.

FIG. 6.2 – Architecture du système de gestion de PJJMU pour les téléphones J2ME

Le prototype peut en outre communiquer avec le serveur HTTP du JMU.

6.5.3 La gestion de PJJMU et la sécurité

Certaines données du PJJMU ont trait à la vie privée de l'utilisateur. Par ailleurs, le manque de considération pour les problèmes de sécurité et de confidentialité dans le développement d'un JMU rendent le vol d'information et la triche relativement facile [124] [7].

Il y a donc un besoin d'utiliser des contre-mesures.

Les joueurs et le terminal qu'ils utilisent (dans notre cas, un téléphone portable) peuvent être considérés comme n'étant pas sûrs, mais les informations sur la carte à puce peuvent être développées avec un souci de sécurité.

Pour accéder aux données privées sur la carte à puce, le programme nécessite une authentification de la part du lecteur. Le processus d'authentification est basé sur un nom d'utilisateur/mot de passe choisis par le joueur durant la création de son compte. La classe `OwnerPIN` permet de garder le mot de passe de manière sécurisée. La procédure de connexion doit être effectuée pour autoriser l'accès aux fonctionnalités cryptographiques de la carte à puce. Quand l'utilisateur ne joue plus, le joueur est délogé de la carte à puce. Un autre code PIN permet au développeur de bloquer ou de débloquer l'accès en écriture du joueur à certains champs.

Nous avons choisi une infrastructure à clé publique pour aider les concepteurs de système de JMU à garantir la sécurité de leurs applications.

Il y a besoin d'une phase de personnalisation au niveau de la carte à puce pour créer des paires de clés et les garder dans la carte à puce. Le côté du serveur nécessite aussi une paire de clés et une infrastructure X.509 pour certifier les clés publiques.

Quand l'application a besoin d'interagir avec le serveur, le serveur envoie sa clé publique ainsi qu'un certificat. La carte à puce vérifie la validité de la clé. Si la clé est valide, la carte à puce peut garder la clé publique. La clé publique de la carte à puce peut aussi être envoyée au serveur. Toutes les interactions suivantes entre la carte à puce et le serveur peuvent être chiffrées en utilisant la clé publique de l'un et la clé privée de l'autre. Plus généralement, le mécanisme vise à proposer aux développeurs de JMU une identification plus forte qu'un simple login/mot de passe pour aider à lutter contre des phénomènes de tricherie en ligne. Un avantage de cette méthode est de n'envoyer aucune donnée sensible en clair sur le réseau.

Une tricherie courante consiste à remplacer du code ou des données dans le jeu. Le simple

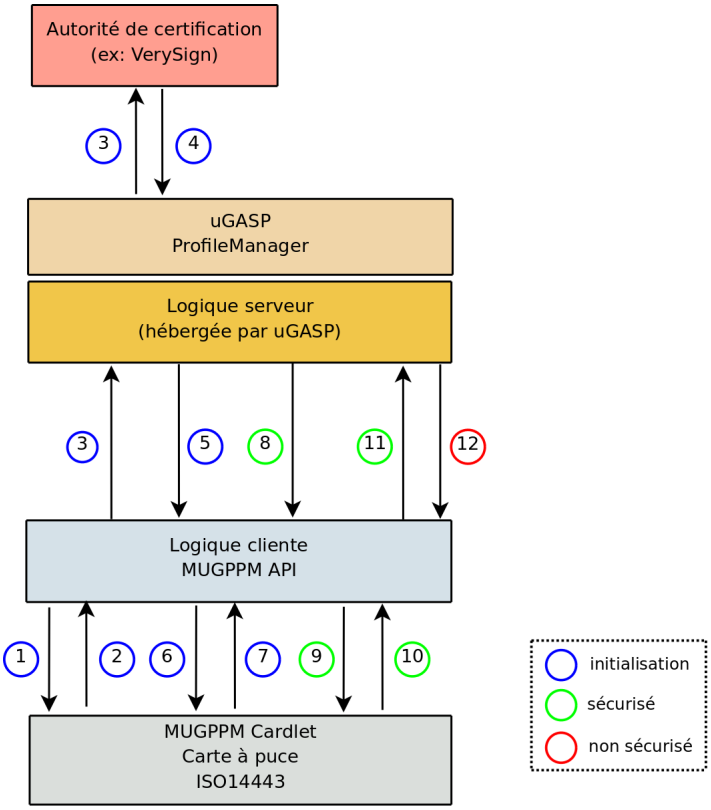


FIG. 6.3 – Exemple de mise en place de transaction sécurisée dans un JMU avec carte à puce

fait d'utiliser une carte à puce pour gérer le PJJMU rend considérablement plus difficile le fait de pratiquer à de telles modifications, le tricheur devant pénétrer les mécanismes de sécurité d'une carte à puce pour accéder à son profil. Le concepteur de jeu pourrait vouloir vérifier une signature pour toute opération modifiant certains éléments du profil.

Une autre tricherie consiste à abuser des procédures de jeu. Par exemple, un joueur peut se déconnecter avant de perdre une partie. Rendre obligatoire la signature de certaines procédures par le serveur peuvent aider à contrer une telle tricherie.

L'aspect mobile des JMU peut impliquer des interactions entre deux joueurs sans être connectés avec le serveur de jeu. Par exemple dans un jeu de rôle, les joueurs peuvent vouloir échanger des objets du jeu. Cette opération pourrait avoir lieu sans serveur en minimisant les risques de triche.

Le mécanisme à clé publique utilisé ici permet d'éliminer les attaques d'écoute des transactions "man in the middle", ce qui pourrait arriver avec des cartes à puce NFC [67]. Il est à noter que toutes les cartes à puce n'implémentent pas forcément ces mécanismes.

Côté serveur, le service peut reposer sur la librairie `java.security.*` et `javax.crypto.*` implantant les mécanismes de génération de clés, d'encodage et de décodage des données à partir de clés valides ... Pour être totalement sécurisé, le service peut intégrer une gestion de certificats conforme à l'architecture X.509 [116] d'échanger les clés publiques certifiées. Ceci nécessite la création d'un certificat auprès d'une autorité de certification telle que VerySign [118] par exemple. La figure 6.3 présente les échanges de données requis lors de ce type de transaction sécurisée pour l'obtention des données de la carte par la logique serveur de jeu dans le cadre d'un service reposant sur le profil de joueur :

- étape 1 : le dispositif NFC demande la clé publique de la carte à puce
- étape 2 : la carte à puce fournit sa clé publique C
- étape 3 : le dispositif NFC envoie C à la logique serveur de jeu
- étape 4 : la logique serveur de jeu valide la clé publique C auprès de l'autorité de certification qui lui retourne un certificat électronique *cert*, le joueur est alors authentifié
- étape 5 : la logique serveur de jeu envoie sa clé publique S accompagnée de *cert* à la logique cliente de jeu
- étape 6 : la logique cliente de jeu transmet S et *cert* à la carte à puce
- étape 7 : la carte à puce valide le certificat (ou pas, dans ce cas là, elle remonte une erreur à la logique cliente de jeu) et peut mettre à jour la valeur de S si celle-ci diffère de la précédente clé stockée
- étape 8 : la logique serveur de jeu requête les clés de données de profil requises par un service spécifique les clés contenues par le message sont chiffrées au sein du message
- étape 9 : la logique cliente envoie une requête de données à la carte à puce (elle ne peut pas identifier les clés requises chiffrées)
- étape 10 : la carte à puce retourne les données chiffrées, à l'aide de sa clé privée c et de S, à la logique cliente de jeu
- étape 11 : la logique cliente génère un message de réponse contenant les données chiffrées
- étape 12 : la logique serveur de jeu reçoit les données, les décode à l'aide de sa clé privée s et de C, calcule le service basé sur le profil et envoie un message associé

Les étapes 1 à 7 correspondent à l'initialisation nécessaire aux mécanismes de sécurité, les clés publiques sont échangées et vérifiées. Les étapes 8 à 12 illustrent les mécanismes de chiffrement et de déchiffrement des données au niveau de la carte à puce et de la logique serveur de jeu, requises pour le calcul d'un service basé sur le profil du joueur. Les transactions entre la carte à puce et la logique serveur de jeu sont ainsi sécurisées. Il est important de noter que la logique cliente elle- même ne peut identifier les données de profil échangées au cours d'une transaction, elle a un rôle de médiateur au sein de la transaction. En ce qui concerne la modification des données stockées sur la carte à puce par la logique serveur de jeu, les étapes 1 à 7 sont conservées et seules les étapes suivantes diffèrent :

- étape 8 : la logique serveur de jeu envoie un message contenant les clés et les valeurs mises à jour correspondantes chiffrées
- étape 9 : la logique cliente transmet le message
- étape 10 : la carte à puce déchiffre les clés et met à jour les données associées
- étape 11 : la logique cliente génère un message d'acquitement, valide ou invalide, qu'elle transmet au serveur
- étape 12 : en cas d'acquitement invalide, la logique serveur de jeu peut réitérer ou non la transaction (retour à l'étape 8)

Ce type de transaction permet la mise à jour du profil en fonction des actions effectuées par le joueur au sein du jeu. Par exemple, si le joueur obtient un objet virtuel de jeu de type clé, celui-ci peut être stocké sur sa carte dans l'optique de débloquer l'accès à une partie de la zone de jeu requérant la clé en question.

6.6 Performances de l'application sur la carte à puce

Pour évaluer la performance de la carte à puce utilisée, on peut se tourner vers le projet MESURE. Nous n'avons pas de profil d'utilisation d'une applet comme celle utilisée dans le PJJMU. L'utilisation d'un profil sur carte à puce pour les JMU reste expérimental et ne concerne

```
// Objet Applet
public void loginProfile(APDU apdu) {
  // [...]
  // La commande APDU a ete copiee dans un tableau d'octets "tmp".
  // indexCurrentUser pointe sur l'utilisateur en cours d'identification
  if (!gameProfiles[indexCurrentUser].
        login(tmp, (byte)(tmp[0]+1), (byte) (lc - (tmp[0]+1)))) {
    ISOException.throwIt(SW_VERIFY_FAILED);
  }
  }

// Objet GameProfile
public boolean login (byte [] pin, byte offset, byte length) {
  boolean checked = password.check(pin, offset, length);
  if (checked) {
    isVerified = true;
  }
  return (checked);
}
```

FIG. 6.4 – Extrait de l'application

pas encore directement les industriels et les universitaires que ce soit dans le domaine de la carte à puce ou dans le domaine des jeux.

Nous avons analysé la performance pour chaque bytecode et pour chaque appel de méthode. Le principe utilisé est le même que lorsque l'on analyse une application pour en faire un profil.

Les figures 6.4 et 6.2 illustrent un extrait de l'application sur la carte à puce. La figure 6.4 correspond à une identification par code PIN d'un utilisateur. Cette procédure d'identification doit nécessairement démarrer toute séance de jeu. La figure 6.2 reprend une partie de ce code source sous forme de bytecode en illustrant les performances moyennes constatées sur une carte à puce utilisée.

On peut observer la performance de l'intégralité du code en suivant un scénario d'utilisation courante de l'application. Sur une carte à puce donnée, on peut donc retrouver le temps total nécessaire pour l'exécution d'une commande. Ce type de test peut nous être utile pour éventuellement prouver que notre concept de JMU avec des cartes à puce embarquant les profils des joueurs est réalisable en pratique.

Ainsi, sur la carte à puce testée, nous pouvons évaluer le temps d'exécution d'une connexion à environ 18,859 ms en comptabilisant tous les temps moyens de performance des bytecodes et des méthodes de l'API utilisés. On peut mesurer l'ensemble des opérations "bout à bout". C'est à dire que l'on démarre un chronomètre, puis on envoie la commande APDU déclenchant la connexion à l'applet qui gère le PJJMU. À la réception de la réponse APDU, on arrête le chronomètre. Le temps ainsi mesuré comprend l'exécution de l'ensemble de la commande, et inclut du bruit (dû au contexte de la mesure), qui cette fois n'est pas isolé. Ces mesures peuvent être effectuées un nombre arbitraire de fois.

Sur la même carte à puce, nous avons mesuré un temps "bout à bout" moyen de 20,573 ms, ce qui représente une déviation d'environ 9,08 de la valeur calculée en additionnant les temps isolés calculés avec MESURE. Il est à noter cependant, que le calcul additionnant les temps isolés

Bytecode	**Temps d'exécution (ns)**	**Notes**
`getfield_a_this 0;// password`	81401	
`aload_1; // pin`	9693	
`sload_2; // offset`	10393	
`sload_3; // length`	9546	
`invokevirtual 30; // check`	9473934	Résultat *true*
`sstore 5; // checked`	28320	
`sload 5; // checked`	17981	
`ifeq L2;`	56732	*false*
`aload_0; // this`	10225	
`sconst_1; // true`	9072	
`putfield_b 1; // isVerified`	76721	
`L2 :sload 5; // checked`	17981	
`sreturn;`	247420	Temps compté avec `invokevirtual`

TAB. 6.2 – Analyse de performance d'un extrait de l'application (l'appel a `login()`)

ne prend pas en compte le temps de communication avec la carte à puce ou encore les bruits potentiels que la mesure "bout à bout" peut engendrer.

Par ailleurs, on peut appliquer la même technique pour toutes les commandes nécessaires à notre application. Dans un but de vérifier l'utilisation de cartes à puce embarquant les profils des joueurs de JMU, il nous faut avant toutes choses vérifier que le chiffrement choisi, RSA avec une clé de 1024 bits, est réaliste. Le temps d'exécution d'un chiffrement RSA sur la carte à puce testée est en moyenne de 389,313 ms. C'est de loin l'opération la plus coûteuse en terme de temps d'exécution parmi toutes celles testées. Il est à noter aussi que la carte à puce testée n'est pas particulièrement récente (2004) et n'est pas la plus performante des cartes à puce testées en règle générale. Elle reste la seule carte à puce a notre disposition utilisant une interface NFC et proposant du RSA. Malgré ses contre-performances, cette carte à puce reste acceptable dans le cadre de la gestion de PJJMU.

Il est à noter que ces mesures sont faites à partir d'une machine hôte qui est un PC et non pas un téléphone portable. Dans le cadre de l'application avec un téléphone portable NFC, il faut prendre en compte la performance de l'interface entre le téléphone et la carte à puce et la performance du téléphone lui même. Cela complique donc l'évaluation des mesures. Par ailleurs, les mesures ont été effectuées avec une carte dual interface (qui est accessible d'une manière sans contact ou par un lecteur compatible ISO 7816), ce qui a permis de mesurer la performance de la carte avec un CAD classique sans prendre en compte la partie communication NFC. Cependant, nous sommes avant tout concernés ici par la réalisabilité de la gestion de PJJMU sur carte à puce et par la correction des mesures individuelles dans le cadre d'une application "réelle". La communication NFC est sans doute un point négatif pour la performance de la carte à puce. Mais le fait est que la carte à puce en question est utilisable dans le cadre de l'application proposée et MESURE participe à la démonstration de ce fait.

6.7 Conclusion et Perspectives

Dans ce chapitre, nous avons vu une utilisation d'une carte à puce NFC pour gérer un profil utilisateur dans le cadre de JMUs. L'approche centrée sur la carte à puce NFC permet de nouvelles

formes d'interactions de manières centralisées et de manière décentralisée. Un des avantages principaux de notre méthode est de permettre aux joueurs de jouer à tout moment et partout, d'où l'aspect ubiquitaire du jeu.

Cela permet aux développeurs de JMUs d'implémenter une architecture basée sur les cartes à puce pour fournir des services basés sur les profils. Le but est avant tout d'offrir aux joueurs des expériences personnalisées. L'API permet au joueur d'accéder à un certain niveau de confidentialité.

Sur la base de notre travail et avec uGASP, il est possible de spécialiser et de réaliser un outil pour développer un JMUs. L'utilisation de cartes à puce dans un domaine comme le jeu apporte un outil intéressant pour les développeurs et les concepteurs de jeu pour mettre en œuvre de nouvelles formes d'interaction et de narration basées sur des technologies de mobilité et d'ubiquité.

La question de "qui personnalise la carte à puce" reste ouverte. Dans des secteurs traditionnels de la carte à puce, comme dans le domaine bancaire, la personnalisation est entreprise par l'émetteur de la carte. Mais la qualité multi-applicative des cartes à puce et le caractère personnalisable des jeux vidéos classiques rend la réponse à cette question non triviale. À l'heure actuelle, nous laissons le joueur remplir un formulaire ce qui est questionnable en terme de sécurité. Le fournisseur de l'application a un certain contrôle sur certains champs comme le code PIN.

Certains projets, comme T2TIT [117] (Things to Things in the Internet of things) pourraient avoir des retombées positives sur notre gestion de profil de joueur sur carte à puce. Ce projet propose une infrastructure pour des interactions avec des objets sans contact pour leur donner une identité sur le réseau, en gardant des garanties fortes concernant la sécurité. Un objet RFID peut être bon marcher, mais il manque de ressources pour être connecté nativement via IP. Ce projet se propose donc d'implémenter une pile de communication standard entre un objet RFID et d'autres entités IP. En particulier, T2TIT peut être combiné à une technologie comme HIP (Host Identity Protocol) qui se base sur une infrastructure à clé publique pour permettre de dissocier l'adresse IP de l'objet et son "identité" (dans le but de favoriser la mobilité) devrait aboutir à des objets mobiles identifiés de manière unique et communicant de manière sécurisée. La conclusion de ce projet devrait nous être utile, par exemple pour utiliser des canaux chiffrés.

La nature des jeux pousse à diversifier les applications sur une carte donnée pour fournir un maximum d'expérience de jeu. Des travaux concernant la vérification de code sur la carte, par exemple en faisant de l'analyse de flux de données (cf [43]) pourrait être aussi utilisée pour fournir une forte protection entre les différentes applications installées sur une carte. Ainsi nous pourrions partager des données d'une application à une autre et restreindre l'accès à certaines données.

Du point de vue des performances, ce chapitre nous permet de conclure à une relative pertinence des données résultant de MESURE, et de la mesure de fractions de code isolées pour mesurer simplement la performance d'une application dans son contexte d'utilisation. Pour le développeur, il est question de savoir si l'application développée est réaliste ou non. Tews et al. [114] ont fait les frais d'un tel développement sans savoir si l'application était réaliste avant de la tester sur différentes cartes à puce. Il se trouve que leur usage des cartes est pour l'heure irréaliste. Mais de tels travaux sont très utiles pour des projets comme PLUG pour découvrir la réalisabilité à priori d'une idée de développement sans pour autant devoir tester l'application dans un contexte d'utilisation complet.

7

Conclusion

MESURE

Le projet MESURE s'est achevé officiellement en Mars 2008. À sa conclusion, le projet avait achevé certains de ces buts principaux. Trois domaines d'applications ont été retenus pour concevoir des notes, et plusieurs applications de chacun de ces domaines ont été utilisées pour concevoir les notes. Les tests créés pour le projet couvrent tous les cas d'utilisation considérés.

Les résultats présentés dans cette thèse concernent plus particulièrement les résultats scientifiques obtenus lors du projet. Une technique d'isolation au niveau des bytecodes a été présentée et elle permet d'avoir une relative confiance dans les résultats. Ceux-ci sont capables de supporter différentes tailles de boucle sans souffrir de bruits supplémentaires. Au contraire, les bruits peuvent être atténués par une taille de boucle suffisamment grande.

Nous avons présenté divers outils utilisables à l'heure actuelle. Tous sont disponibles au téléchargement sur le site de MESURE [78]. Un apport de nouveaux outils pourrait aussi compléter les travaux entrepris ici. Le caractère modulaire de tous les outils développés rend possible le développement d'applications supplémentaires qui compléteraient les outils déjà développés.

Aucun environnement de mesure dédié n'est nécessaire pour utiliser les outils développés (en dehors d'un environnement avec Java 6 et un CAD). Les outils développés permettent de récupérer des résultats considérés comme consistants avec des mesures dans un environnement potentiellement bruité. L'analyse a permis de montrer que malgré une distribution des résultats relativement chaotique, l'essentiel, c'est à dire le temps d'exécution isolé reste proche d'une mesure idéale.

Il est très possible de considérer une extension de MESURE à d'autres domaines d'application. Les efforts nécessaires pour franchir un tel pas consistent essentiellement à utiliser une ou des applications représentatives d'un domaine d'application, à produire de scénarii d'utilisation de ces applications, et à en extraire les bytecodes et les méthodes de l'API qui sont utilisés. Un domaine comme la télévision mobile, qui nécessite des capacités importantes de la carte en terme de performance pourrait par exemple bénéficier de cet outil. Pour diverses raisons nous n'avons pas fourni de profil de carte pour ce type d'application, et ce travail pourrait être complété spécifiquement par les entreprises concernées.

Pour des raisons de licence, l'accès à la machine outillée de Java Card qui permet d'obtenir des profils d'utilisation des applications nous a été refusé. Il a donc toujours fallu passer par un de nos partenaires pour obtenir les profils des applications "réelles". Pour la même raison, il nous est impossible de fournir un tel outil avec la distribution de MESURE. Un tel outil aurait pour objectif la conception à la volée de profil de cartes pour des applications originales.

Les développeurs pourraient ainsi développer des applications et connaître les performances de plusieurs cartes à puce pour ces applications données. Les problèmes rencontrés par Tews [114] par exemple seraient ainsi directement contournables.

Un travail de maintenance est à envisager sur le code développé. Ce code est hébergé grâce à l'outil GForge de l'INRIA.

Il est à noter qu'il reste tout à fait possible de modifier fortement un PC ordinaire pour en faire une machine assez précise de mesure de performance. L'utilité d'une telle modification serait d'avoir une plateforme de mesure de la performance des cartes à puce à moindre coût mais qui garde une grande exactitude. Pour cela il serait indispensable d'avoir accès au code source du noyau de l'OS, de la version de PC/SC utilisée et du driver du CAD utilisé. Certaines manipulations sont à envisager comme la désactivation de certains signaux d'interruption. Dans l'idéal, une partie cliente devrait alors être développée en C et en assembleur en utilisant des appels systèmes à l'horloge par exemple.

Certains des buts originaux de MESURE ont vite été abandonnés comme les mesures d'entrées/sorties. Ces abandons ont été abondamment justifiés dans les documents publiés [20].

La qualité de logiciel libre du projet a aussi probablement une importance pour l'avenir de la mesure de performance sur la carte à puce. En effet, si d'autres travaux ont tenté de mesurer les performances des plates-formes Java Card, une grande partie d'entre eux restent restreints dans leurs objectifs. Vraisemblablement, la plupart des acteurs de l'industrie ont déjà des benchmarks, au moins pour juger sommairement la performance des cartes à puce. Aucun de ces projets n'est disponible au télé-chargement. Le futur des travaux sur la performance dans les différentes entreprise du secteur pourraient donc bien s'inspirer de MESURE, voir d'utiliser l'intégralité des outils présentés ici.

Le souci qui nous a animé au cours de ce projet a été de ne pas condamner une carte pour des performances insuffisantes sur un point précis ou pour un domaine d'application, mais de fixer des notes qui représentent la performance des cartes et qui puissent être utilisées en relation avec des données sur le niveau de sécurité dans la carte et le prix de celle-ci. La sécurité, en particulier, pourrait se faire au détriment de la performance. Les tests ont révélé qu'une différence frappante se situe entre deux générations de cartes d'un même fabricant. Les performances peuvent être ralenties par les besoins de sécurité, mais elles seront inexorablement tirées vers le haut par l'évolution matérielle. Ainsi, des applications qui semblent inaccessibles et trop coûteuses aujourd'hui seront peut être la norme demain.

MESURE a été récompensé par le prix Isabelle Attali décerné par l'INRIA tous les ans lors de la conférence Smart Event. Ce prix récompense la meilleure innovation technologique.

Certains travaux entrepris ici, comme l'intégration de profils de joueurs pour des JMUs restent des travaux originaux qui ont encore un potentiel de développement important. En effet il reste à montrer certaines utilisations pratiques de l'application développée dans un cadre complet. Les interactions de jeu avec des cartes à puce et des téléphones ou des bornes forment un cadre nouveau pour expérimenter avec les capacités des cartes. La personnalisation des profils de joueurs avec l'aide de cartes à puce représente une nouvelle approche de ce type de jeu qui peut être combinée à d'autres techniques comme la géolocalisation pour fournir de nouvelles expériences ludiques.

Ce développement a finalement permis d'utiliser MESURE dans un cadre nouveau tout en validant une partie du travail accompli avec l'isolation des mesures. Si le recoupement des temps d'exécution individuels avait été très éloigné du temps d'exécution global, les résultats de MESURE auraient été remis en cause. Le fait que les deux résultats soient relativement voisins nous permet de conclure que l'isolement des mesures reste une méthode valable, mais qu'il faut cependant questionner toutes les mesures utilisées pour savoir de quelle marge est ce qu'il est

question lorsqu'on parle d'exactitude dans le cadre d'une application réelle.

Java Card 3.0

La publication de la version 3.0 de Java Card en Mars 2008 a marqué l'évolution du domaine vers des plates-formes plus imposantes, plus souples et plus gourmandes. L'évolution notable est l'apport d'une approche "serveur web" en plus de la conservation de la compatibilité ISO 7816.

Si des démonstrations ont été faites et que des prototypes ont été réalisés par des entreprises, aucune carte n'est encore largement commercialisée avec Java Card 3.0 aujourd'hui, et le standard continue d'évoluer.

Le standard 3.0 est divisé en deux éditions : une édition connectée et une édition classique. Si l'édition classique est une évolution de la version 2.2.2, l'édition connectée, elle propose une approche relativement nouvelle : la programmation peut ressembler à celle d'un serveur HTTP, le ramasse miette est plus pro-actif, les Threads font leur apparition ... La version connectée se rapproche sensiblement de J2ME CLDC. Un émulateur est fourni avec l'implémentation de référence (RI) [59]. Cela a permis de tester une applet telle que celles conçues avec MESURE pour tester son exécution dans l'émulateur. Les résultats n'ont bien sûr pas de sens, mais cela prouve la portabilité du code de MESURE vers Java Card 3.0.

Cependant tout un pan de cette nouvelle plateforme échappe à MESURE à l'heure actuelle. En effet, tous les éléments propres à la version connectée comme le multithreading, et l'utilisation d'applets sous la forme de serveurs HTTP ne sont pas testés par MESURE. Le développement de ces tests est envisageable, mais la véritable question devrait être "quelle est l'utilisation standard d'applications typiques en Java Card 3.0 ?" Il faut donc reprendre une phase de sélection de domaine, de scénarii d'utilisation typique de Java Card 3.0. Étant donné que les applications en question n'existent que sur des émulateurs ou sur des prototypes développés par les acteurs principaux derrière cette évolution, il est encore prématuré de vouloir en fournir des profils d'utilisation.

Encore une fois, c'est une question qui est liée à la maturité de cette technologie. Il semblerait logique que les outils développés pour MESURE soient étendus pour tester les plates-formes Java Card 3.0.

A

Publications

Publications en rapport avec le sujet - 11

Revues internationales - 2

- S. Bouzefrane, J. Cordry et Pierre Paradinas. *MESURE Tool to benchmark Java Card platforms*. International Journal of Computer Science Issues, - à paraître 2009.
- R. Pellerin, J. Cordry, E. Gressier et C. Yan. *Player profile management on NFC Smart Card for Multiplayer Ubiquitous Games* International Journal of Computer Games Technology à paraître 2009.

Conférences - 6

Conférences internationales - 4

- P. Paradinas, J. Cordry et S. Bouzefrane. *Measurement Analysis when Benchmarking Java Card Platforms*. Third IFIP WG 11.2 International Workshop in Information Security Theory and Practices (WISTP 09), Springer LNCS, pp. 84-94, Bruxelles Belgique, 1-4 Septembre 2009.
- S. Bouzefrane, J. Cordry, H. Meunier et P. Paradinas. *Evaluation of Java Card Performance*. Smart Card Research and Advanced Applications, 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008 pp. 228-240, Londres, Royaume Uni, 8-11 Septembre 2008.
- R. Pellerin, C. Yan, J. Cordry et E. Gressier-Soudan. *Player profile management on NFC Smart Card for Multiplayer Ubiquitous Games*. CyberGames'08 International Conferance on Games Research and Development, ACM, pp. 16-23 Beijin, Chine Octobre 2008.
- Pierre Paradinas, Julien Cordry, Samia Bouzefrane. *Performance Evaluation of Java Card Bytecodes*. Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop, WISTP 2007, pp 127-137, Héraklion, Crète, Grèce, 9-11 Mai 2007.

Conférences nationales - 2

- S. Bouzefrane, J. Cordry et P. Paradinas. *A methodology for testing Java Card performance*. In CFSE'08 Conférence Française en Systèmes d'Exploitation, Suisse, 2008.
- J. Cordry. *La performance des plates-formes Java Card*. 9ème Atelier en Évaluation de Performance, Aussois, France, Juin 2008.

Communications orales sans actes dans un congrès international - 2

- S. Bouzefrane, J. Cordry, G. Grimaud et P. Paradinas. *An Open-Source Tool to Benchmark Java Card Platforms.* In e-Smart Conference, Sophia Antipolis, Septembre 2008
- P. Paradinas, J. Cordry et S. Bouzefrane. *How to Measure the Performance of Java Card Platforms ?* e-Smart Conference, Sophia Antipolis, Septembre 2007

Rapports du projet MESURE - 5

- C. Boë, S. Bouzefrane, J. Cordry, G. Grimaud,, H. Meunier, P. Paradinas, C. Pascal et E. Vétillard. *MESURE - State of the Art*, Mars 2008
- C. Boë, S. Bouzefrane, J. Cordry, G. Grimaud, H. Meunier, P. Paradinas et C. Pascal et E. Vétillard. *MESURE - Requirements*, Mars 2008
- C. Boë, S. Bouzefrane, J. Cordry, G. Grimaud, H. Meunier, P. Paradinas, C. Pascal et E. Vétillard. *MESURE - Functionalities*, Mars 2008
- C. Boë, S. Bouzefrane, J. Cordry, G. Grimaud, H. Meunier, P. Paradinas, C. Pascal et E. Vétillard. *MESURE - Methodology*, Mars 2008
- C. Boë, S. Bouzefrane, J. Cordry, G. Grimaud, H. Meunier, P. Paradinas, C. Pascal, E. Tsassong et E. Vétillard. *MESURE - User Guide*, Mars 2008

Article de vulgarisation - 1

- S Bouzefrane, J. Cordry et G. Grimaud. *La programmation Java Card.* Multi-System & Internet Security Cookbook (MISC) Hors Série - Carte à Puce. Diamond Edition Novembre 2008.

Distinction

Prix Isabelle Attali pour *How to measure the performance of the Java Card Platforms* avec Samia Bouzefrane, à Sophia Antipolis en 2007. Ce prix en hommage à Isabelle Attali distingue la communication scientifique la plus innovante présentée lors de la conférence annuelle Smart Event.

Autres publications - 3

Conférences internationales - 2

- J.-P. Etienne, J. Cordry et S. Bouzefrane. *Applying the CBSE Paradigm in the Real Time Specification for Java.* In JTRES 06 4th international workshop on Java technologies for real-time and embedded systems. Paris France, pp. 218-226, ACM, 2006.
- J. Cordry, N. Bouillot, S. Bouzefrane. *Performing Real-Time Scheduling in an Interactive Audio-Streaming Application.* ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, May 25-28, 2005 ISBN 972-8865-19-8 5, pp. 140-147, Chin-Sheng Chen, Joaquim Filipe, Isabel Seruca, José Cordeiro (Eds.), 2005.

Conférences nationales - 1

- J. Cordry, N. Bouillot et S. Bouzefrane. *BOSSA et le Concert Virtuel Réparti, intégration et paramétrage souple d'une politique d'ordonnancement spécifique pour une application*

multimédia distribuée. In RTS'05 13th International Conference on Real-Time Systems Paris. Avril 2005, pp. 18-39, CNRS-Loria.

B

Encadrements

Voici une liste de stage encadrés :

Sébastien Ronsse — Mai - Août 2006

Stage de 4ème année de l'ESIEE - section systèmes embarqués
Module de chargement Globlal Platform.

Ernest Tsassong — Mai - Août 2007

Stage de 4ème année de l'ESIEE - section systèmes embarqués
Module de présentation des résultats.
Ce travail a abouti sur le développement d'une partie du module "Profiler" dans le projet MESURE et sur l'écriture d'une partie des rapports : "MESURE - User Guide".

Henri Pied — Mai - Août 2008

Stage de 4ème année de l'ESIEE - section systèmes embarqués
Jeux de tests et Java Card RMI.

C

CAD de précision

Ces captures illustrent les mesures effectuées au lecteur de précision MP300.

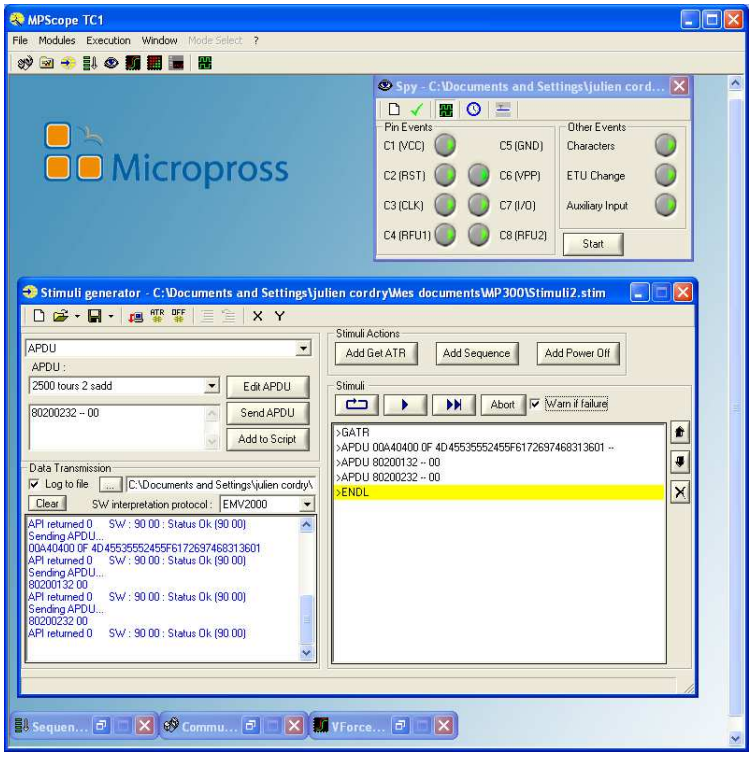


FIG. C.1 – Début de capture sur un ensemble de commandes

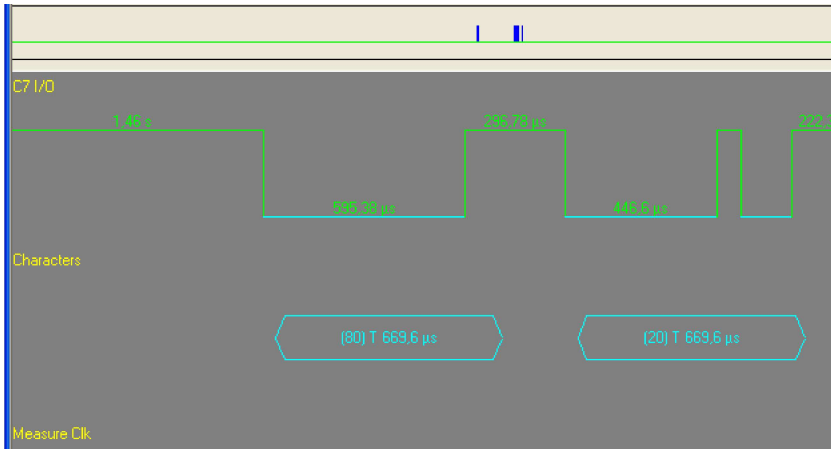


FIG. C.2 – Partie d'une trace mesurée sur une carte. On y distingue l' début d'un APDU envoyé.

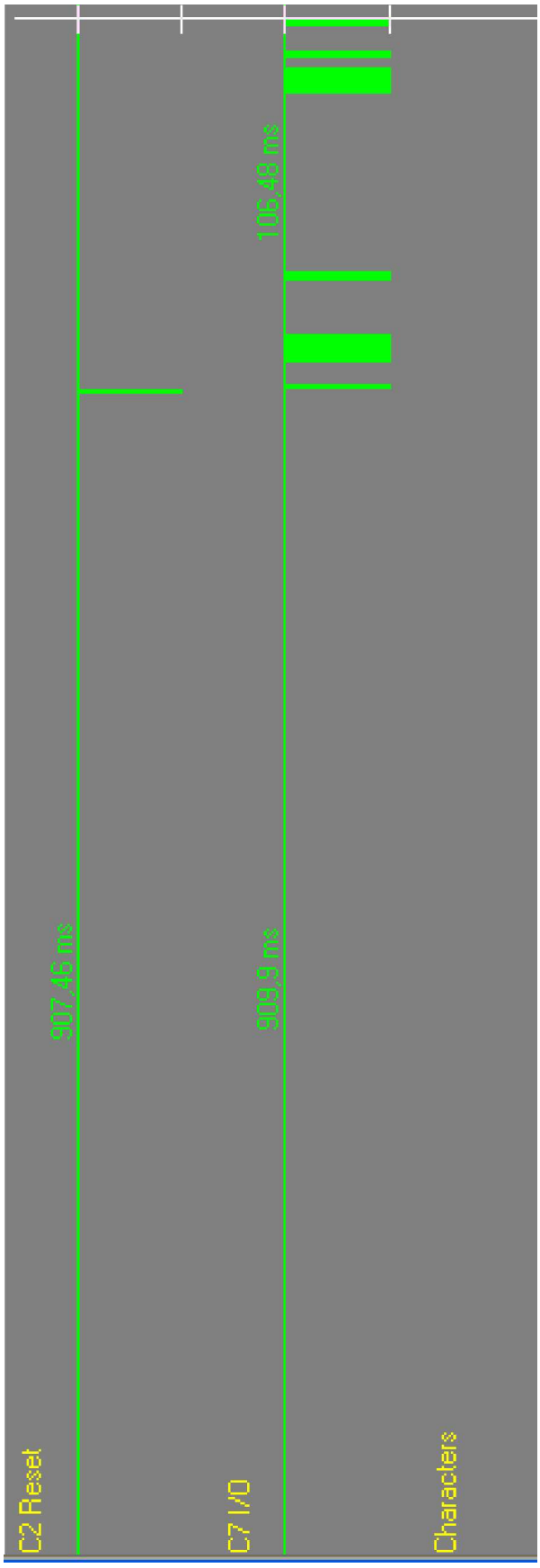


FIG. C.3 – Partie d'une trace mesurée sur une carte. On y distingue des échanges entre une carte et le lecteur.

D

Profile de l'application MCardApplet

Pour chaque type primitif, la valeur passée est indiquée. Pour chaque type de classe, la référence est indiquée. Pour chaque tableau, la longueur, la durée de vie et la référence sont indiquées.

Les durées de vie possibles correspondent au type de mémoire concérné :

- `STD` : un tableau standard de durée persistante.
- `COD` : un tableau de durée CLEAR ON DESELECT (il est effacé une fois que l'application n'est plus utilisée).
- `COR` : un tableau de durée CLEAR ON RESET (il est effacé une fois que la carte est retirée).
- `GLOB` : un tableau global (comme le buffer de l'APDU).

Méthode (arguments)	**N**	**Transact.**
`javacard/security.RSAPublicKeyImpl :setExponent` (1 invocation)		
`(byte[0x2710 STD ]=0xa00f, short=0x009a, short=0x0003)`	1	0
`javacard/security.RSAPublicKeyImpl :setModulus` (1 invocation)		
`(byte[0x2710 STD ]=0xa00f, short=0x0018, short=0x0080)`	1	0
`javacardx/crypto.CipherImpl :doFinal` (1 invocation)		
`(byte[0x0106 STD GLOB]=0x9000, short=0x08, short=0x80,`		
`byte[0x2710 STD]=0xa00f,short=0xb1)`	1	0
`javacard/framework.Util :arrayCopy` (1 invocation)		
`(byte[0x0006 STD ]=0xa20f, short=0x00, byte[0x2710`		
`STD ]=0xa00f, short=0xa5,short=0x06)`	1	0
`javacard/framework.AID :getBytes` (1 invocation)		
`(byte[0x0106 STD GLOB]=0x9000, short=0x09)`	1	0
`javacard/framework.OwnerPIN :check` (1 invocation)		
`(byte[0x0008 STD ]=0xa90b, short=0x00, byte=0x08)`	1	0
`javacard/framework.Util :arrayFillNonAtomic` (2 invocations)		
`(byte[0x2710 STD ]=0xa00f, short=0x9f, short=0x25a9, byte=0x00)`	1	0
`(byte[0x2710 STD ]=0xa00f, short=0x9f, short=0x82, byte=0x00)`	1	0
`javacard/framework.Util :setShort` (22 invocations)		
`(byte[0x0006 STD ]=0xa20f, short=0x00, short=0x01)`	1	0
`(byte[0x0006 STD ]=0xa20f, short=0x02, short=0x01)`	1	0
`(byte[0x0006 STD ]=0xa20f, short=0x04, short=0x01)`	1	0
`(byte[0x0106 STD GLOB]=0x9000, short=0x00, short=0xfffe)`	1	0
`(byte[0x0106 STD GLOB]=0x9000, short=0x02, short=0x00)`	1	0
`(byte[0x0106 STD GLOB]=0x9000, short=0x04, short=0x00)`	1	0

(byte[0x0106 STD GLOB]=0x9000, short=0x06, short=0x8b)	1	0
(byte[0x0106 STD GLOB]=0x9000, short=0x0e, short=0x00)	1	0
(byte[0x0106 STD GLOB]=0x9000, short=0x1a, short=0x400)	1	0
(byte[0x2710 STD]=0xa00f, short=0x131, short=0x2527)	1	0
(byte[0x2710 STD]=0xa00f, short=0x133, short=0x2673)	1	0
(byte[0x2710 STD]=0xa00f, short=0x9d, short=0x25bb)	2	0
(byte[0x2710 STD]=0xa00f, short=0x9d, short=0x94)	1	0
(byte[0x2710 STD]=0xa00f, short=0x9f, short=0x02)	1	0
(byte[0x2710 STD]=0xa00f, short=0x9f, short=0x2673)	1	0
(byte[0x2710 STD]=0xa00f, short=0xa1, short=0xffff)	1	0
(byte[0x2710 STD]=0xa00f, short=0xa3, short=0xffff)	1	0
(byte[0x2710 STD]=0xa00f, short=0xab, short=0x01)	1	0
(byte[0x2710 STD]=0xa00f, short=0xad, short=0x25a9)	1	0
(byte[0x2710 STD]=0xa00f, short=0xad, short=0x82)	1	0
(byte[0x2710 STD]=0xa00f, short=0xaf, short=0x80)	1	0
javacard/framework.Util :getShort (72 invocations)		
(byte[0x0106 STD GLOB]=0x9000, short=0x00)	1	0
(byte[0x0106 STD GLOB]=0x9000, short=0x06)	1	0
(byte[0x0106 STD GLOB]=0x9000, short=0x08)	1	0
(byte[0x2710 STD]=0xa00f, short=0x02)	6	0
(byte[0x2710 STD]=0xa00f, short=0x04)	14	0
(byte[0x2710 STD]=0xa00f, short=0x06)	9	0
(byte[0x2710 STD]=0xa00f, short=0x0c)	2	0
(byte[0x2710 STD]=0xa00f, short=0x0e)	1	0
(byte[0x2710 STD]=0xa00f, short=0x10)	2	0
(byte[0x2710 STD]=0xa00f, short=0x131)	2	0
(byte[0x2710 STD]=0xa00f, short=0x133)	1	0
(byte[0x2710 STD]=0xa00f, short=0x14)	2	0
(byte[0x2710 STD]=0xa00f, short=0x16)	1	0
(byte[0x2710 STD]=0xa00f, short=0x265a)	4	0
(byte[0x2710 STD]=0xa00f, short=0x265c)	5	0
(byte[0x2710 STD]=0xa00f, short=0x265e)	1	0
(byte[0x2710 STD]=0xa00f, short=0x2673)	1	0
(byte[0x2710 STD]=0xa00f, short=0x2675)	2	0
(byte[0x2710 STD]=0xa00f, short=0x98)	1	0
(byte[0x2710 STD]=0xa00f, short=0x9d)	4	0
(byte[0x2710 STD]=0xa00f, short=0x9f)	3	0
(byte[0x2710 STD]=0xa00f, short=0xa1)	3	0
(byte[0x2710 STD]=0xa00f, short=0xa3)	3	0
(byte[0x2710 STD]=0xa00f, short=0xad)	2	0
javacard/framework.Util :arrayCopyNonAtomic (5 invocations)		
(byte[0x0106 STD GLOB]=0x9000, short=0x05, byte[0x0008 STD] =0xa90b, short=0x00, short=0x08)	1	0
(byte[0x0106 STD GLOB]=0x9000, short=0x08, byte[0x0106 STD GLOB]=0x9000, short=0x11, short=0x08)	1	0
(byte[0x2710 STD]=0xa00f, short=0x08, byte[0x0106 STD GLOB]=0x9000, short=0x08, short=0x06)	1	0

(byte[0x2710 STD]=0xa00f, short=0x266a, byte[0x0106 STD GLOB]=0x9000, short=0x1c,short=0x09)	1	0
(byte[0x2710 STD]=0xa00f, short=0xaf, byte[0x0106 STD GLOB]=0x9000, short=0x00,short=0x82)	1	0

TAB. D.2: Méthode le l'API sur des tableaux

Méthode de l'API	**N**	**Transaction**
javacard/framework.Util :getShort	72	0
javacard/framework.Util :setShort	22	0
javacard/framework.OwnerPIN :reset	8	0
javacard/framework.Util :arrayCopyNonAtomic	5	0
javacard/framework.APDU :getBuffer	5	0
javacard/framework.APDU :setIncomingAndReceive	4	0
javacard/framework.Util :makeShort	3	0
javacard/framework.APDU :setOutgoingAndSend	3	0
javacard/security.KeyImpl :getSize	2	0
javacard/framework.Util :arrayFillNonAtomic	2	0
javacardx/crypto.CipherImpl :init(Key,byte)	1	0
javacardx/crypto.CipherImpl :doFinal	1	0
javacard/security.RSAPublicKeyImpl :setModulus	1	0
javacard/security.RSAPublicKeyImpl :setExponent	1	0
javacard/security.KeyImpl :getType	1	0
javacard/security.KeyImpl :clearKey	1	0
javacard/framework.Util :arrayCopy	1	0
javacard/framework.OwnerPIN :getTriesRemaining	1	0
javacard/framework.OwnerPIN :check	1	0
javacard/framework.JCSystem :getAID	1	0
javacard/framework.Applet :selectingApplet	1	0
javacard/framework.APDU :getOutBlockSize	1	0
javacard/framework.AID :getBytes	1	0

TAB. D.1 – Les appels de méthode de l'API dans MCardApplet

Bytecodes	N	Tr	Bytecodes	N	Tr
GETFIELD_A_THIS	217	0	GETFIELD_B_THIS	13	0
SLOAD_1	192	0	SCONST_3	12	0
SLOAD_3	169	0	IFNULL	12	0
SLOAD	145	0	SINC	11	0
INVOKEVIRTUAL	137	0	PUTFIELD_S	11	0
SADD	128	0	STABLESWITCH	10	0
INVOKESTATIC	127	0	BASTORE [COR]	8	0
SLOAD_2	120	0	SSTORE_2	7	0
SRETURN	99	0	BASTORE [STD GLOB]	7	0
IF_SCMPNE	89	0	ALOAD_3	7	0
BSPUSH	82	0	INVOKEINTERFACE	6	0
SSTORE	80	0	SLOOKUPSWITCH	5	0
SCONST_M1	71	0	SCONST_5	5	0
SCONST_2	69	0	GOTO_W	5	0
ALOAD_2	68	0	ASTORE_2	5	0
SCONST_0	67	0	ASTORE	5	0
SSTORE_3	55	0	SSHL	4	0
RETURN	53	0	BALOAD [STD]	4	0
ALOAD_0	51	0	ASTORE_3	4	0
IFNE	45	0	SSPUSH	3	0
BALOAD [STD GLOB]	43	0	SAND	3	0
GOTO	41	0	IF_SCMPLE	3	0
GETFIELD_S_THIS	38	0	IFNONNULL	3	0
SCONST_1	37	0	GETSTATIC_B	3	0
POP	32	0	DUP	3	0
SCONST_4	27	0	CHECKCAST	3	0
GETSTATIC_A	27	0	BALOAD [COR]	3	0
ALOAD_1	27	0	SDIV	2	0
ALOAD	26	0	PUTFIELD_B	2	0
IFEQ	23	0	PUTFIELD_A	2	0
INVOKESPECIAL	22	0	IFLE	2	0
SSUB	19	0	ACONST_NULL	2	0
SSTORE_1	19	0	SREM	1	0
SLOAD_0	18	0	SOR	1	0
IF_SCMPGE	18	0	SMUL	1	0
IF_SCMPEQ	18	0	PUTSTATIC_A	1	0
AALOAD [STD]	18	0	NEWARRAY	1	0
IF_SCMPLT	17	0	IFLT	1	0

TAB. D.3 – Les bytecodes du profile de MCardApplet

E

Autres Distributions

Il est important de montrer que quelque soit la carte à puce, quelque soit le CAD, quelque soit l'OS les distributions ne sont pas normales. Ainsi nous comparons quelques distributions sur plusieurs cartes à puce, plusieurs CAD, plusieurs OS, plusieurs tailles de boucle pour un test donné.

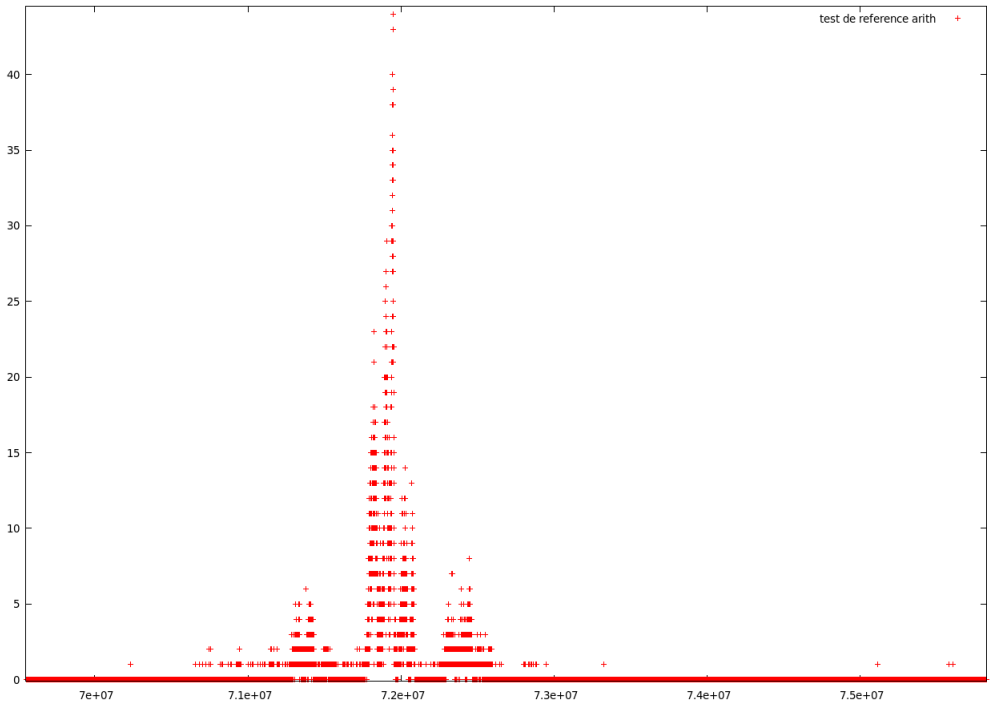


FIG. E.1 – Carte A, CAD 1, Linux, $L = \texttt{P2}^2 = 100$

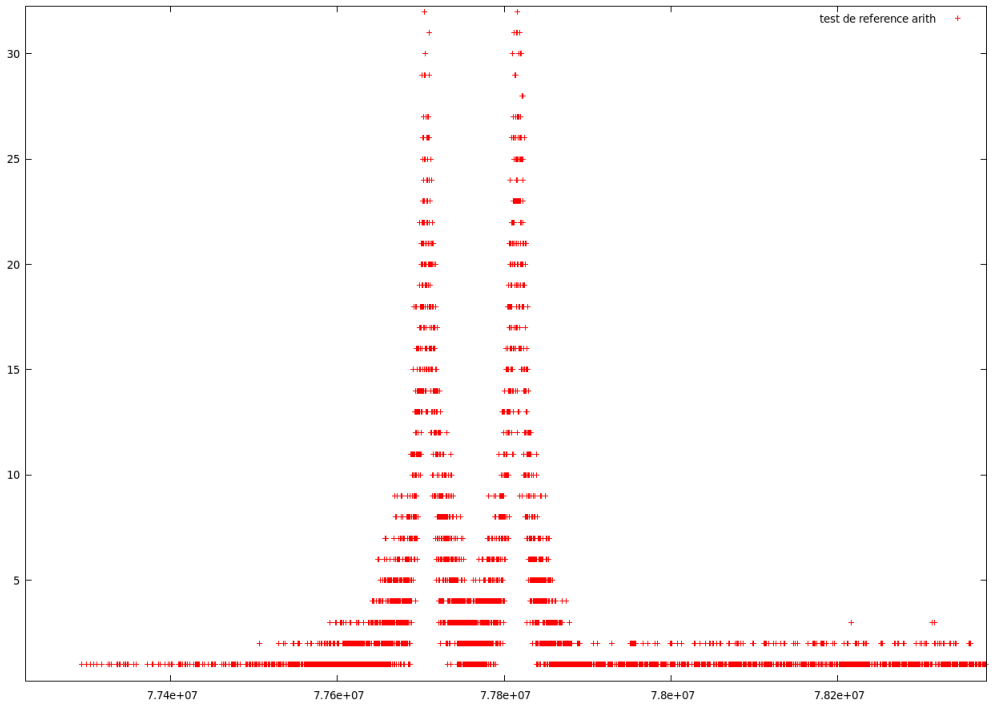


FIG. E.2 – Carte A, CAD 1, Vista, $L = \texttt{P2}^2 = 100$

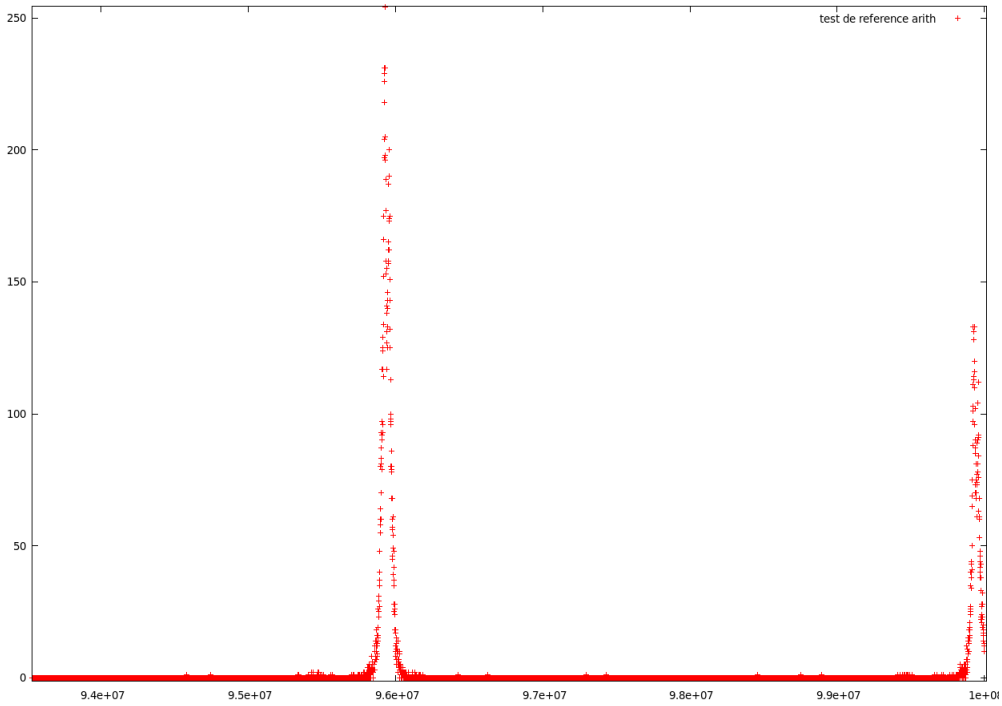


FIG. E.3 – Carte A, CAD 2, Linux, $L = \texttt{P2}^2 = 100$

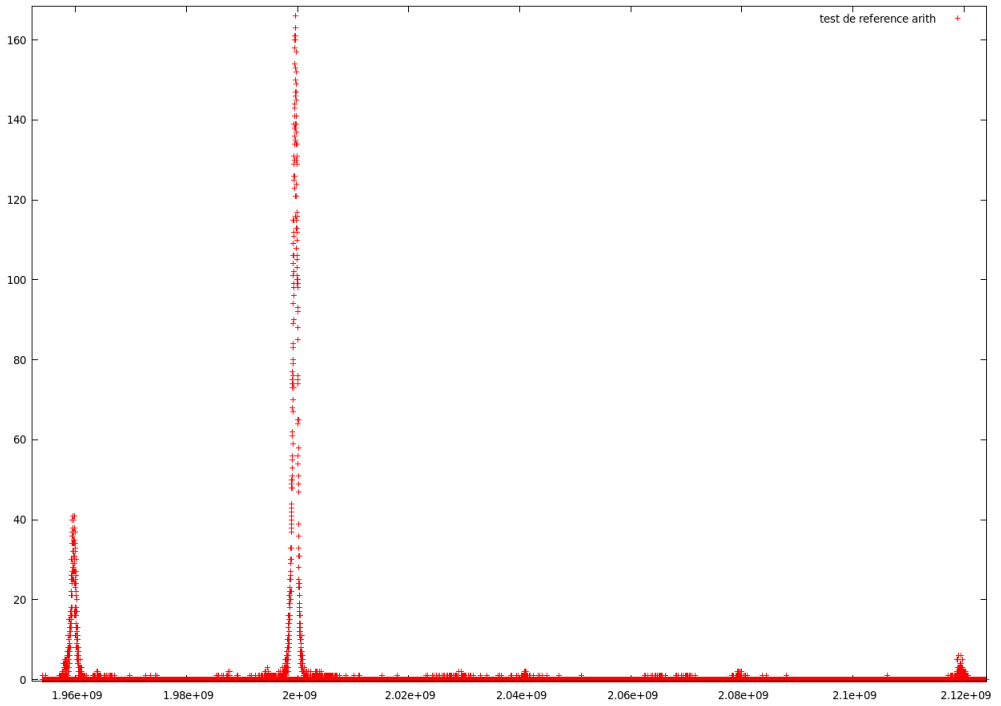


FIG. E.4 – Carte B, CAD 2, Linux, $L = \texttt{P2}^2 = 100$

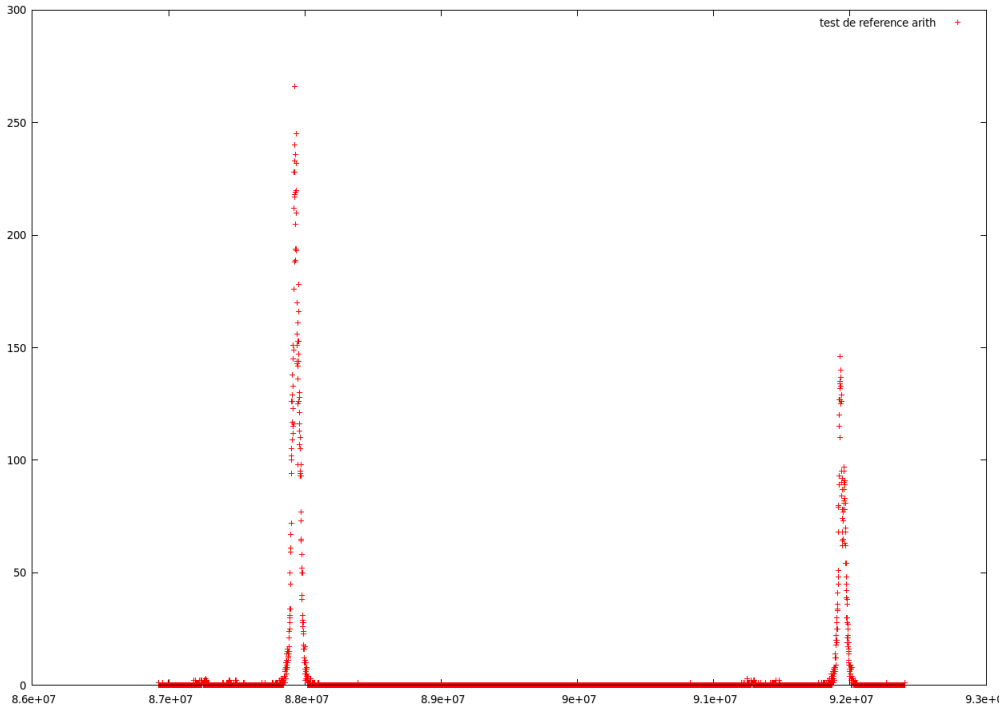


FIG. E.5 – Carte C, CAD 2, Linux, $L = \texttt{P2}^2 = 100$

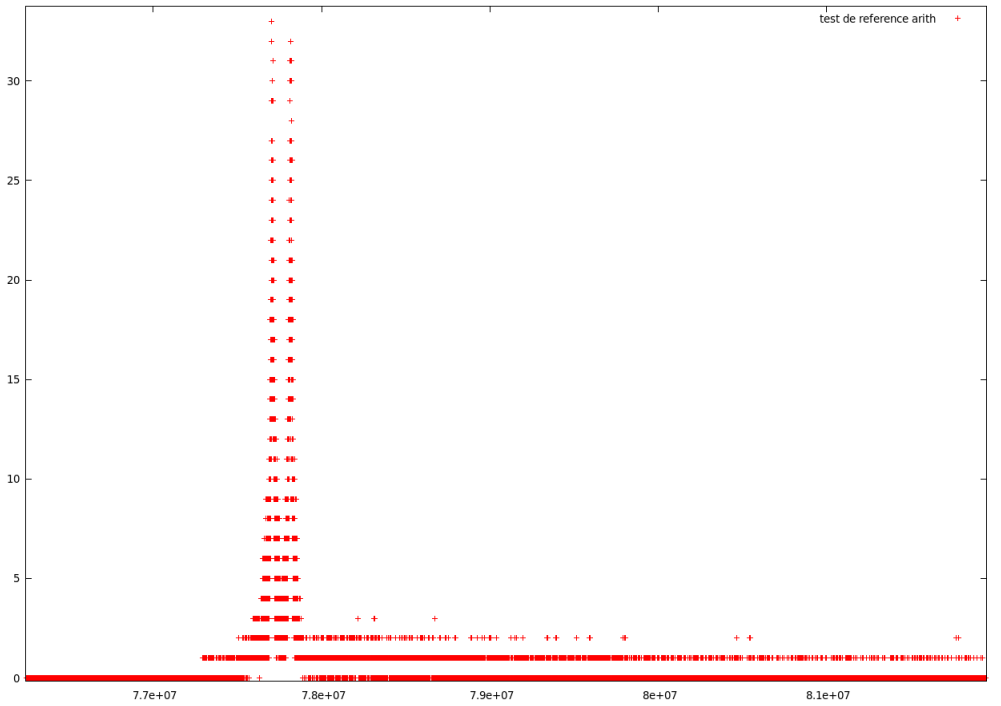


FIG. E.6 – Carte A, CAD 2, XP, $L = \texttt{P2}^2 = 100$

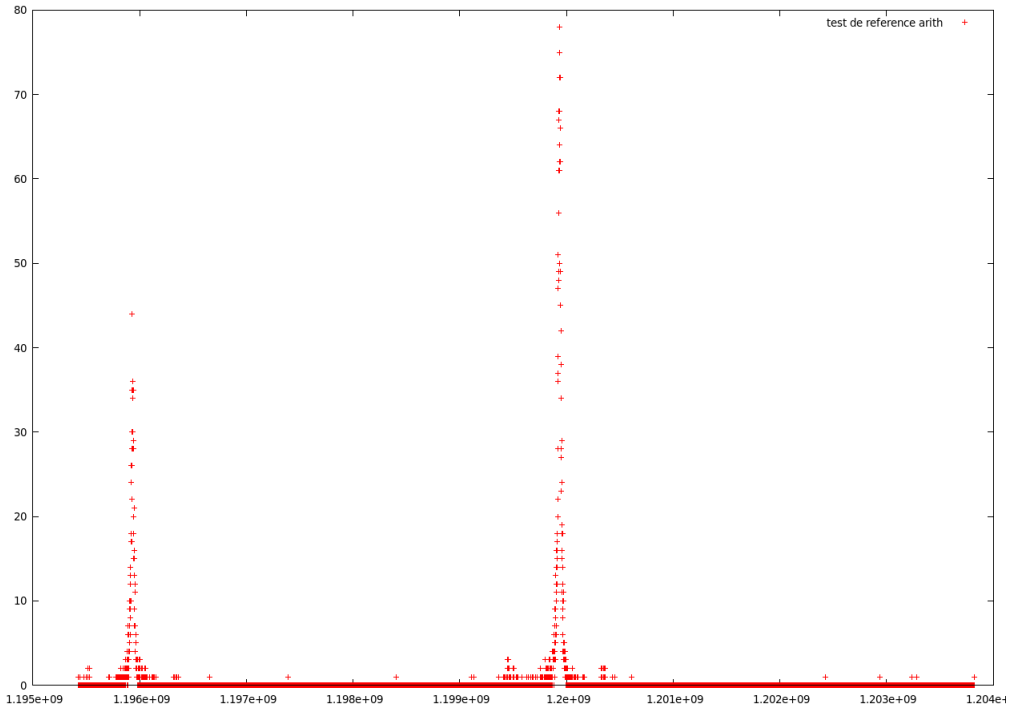


FIG. E.7 – Carte A, CAD 2, Linux, $L = \texttt{P2}^2 = 4225$

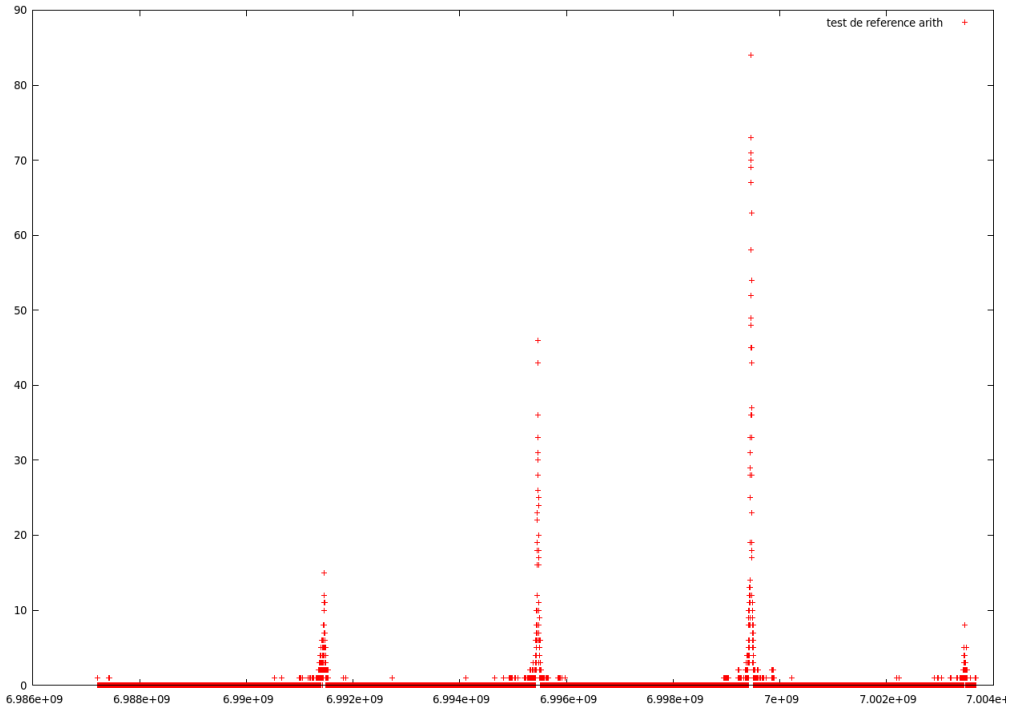


FIG. E.8 – Carte A, CAD 2, Linux, $L = \texttt{P2}^2 = 10000$

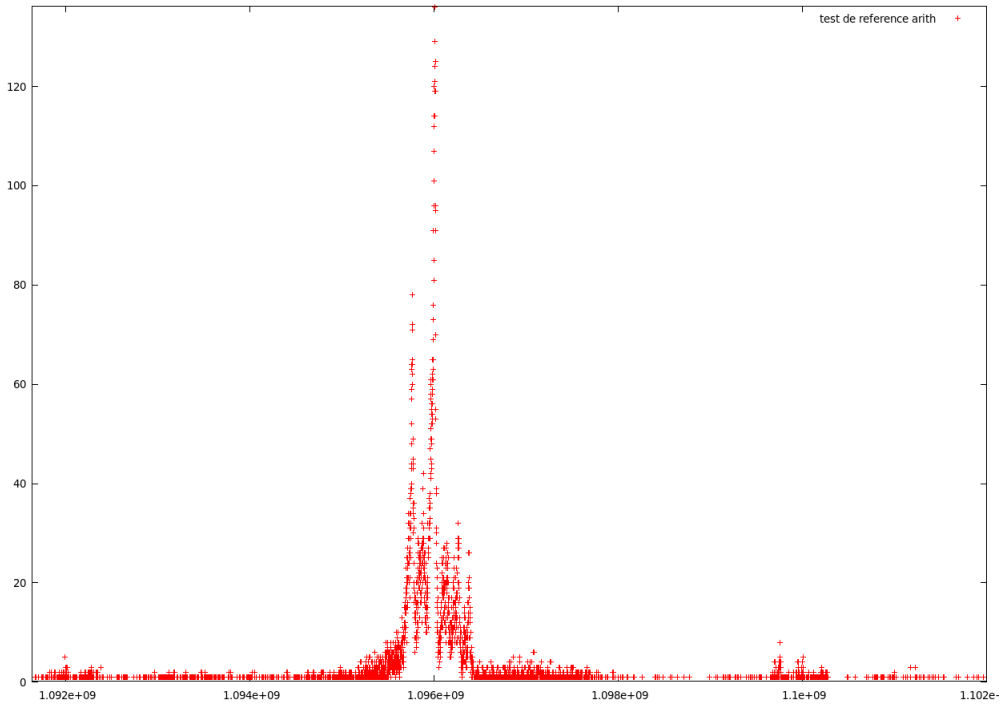


FIG. E.9 – Carte C, CAD 3, Linux, $L = \texttt{P2}^2 = 4225$

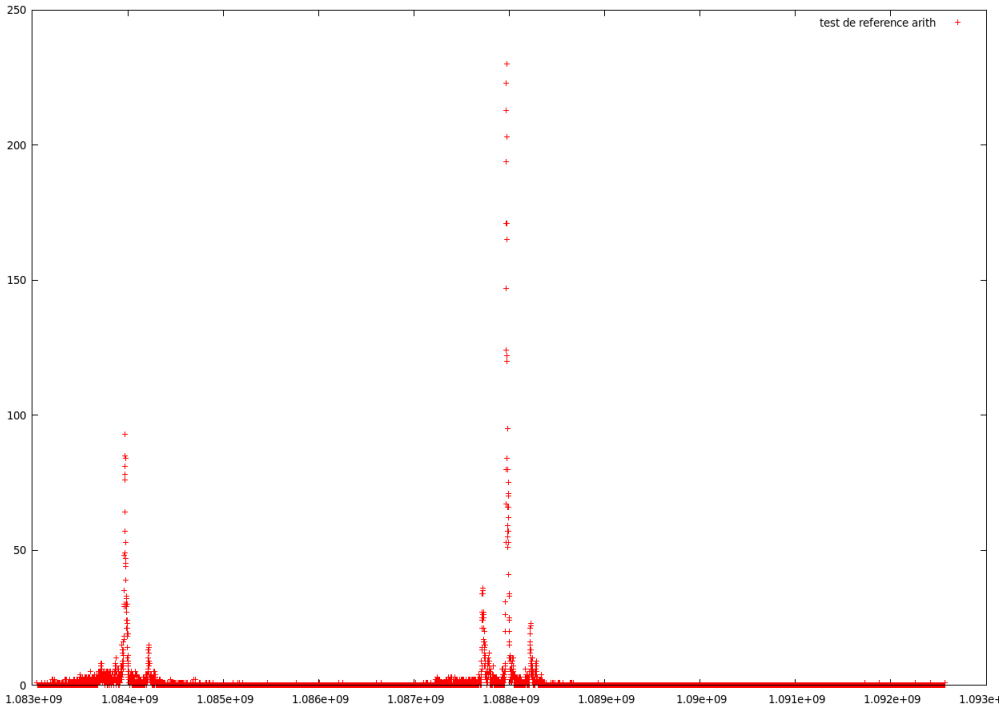


FIG. E.10 – Carte C, CAD 4, Linux, $L = \texttt{P2}^2 = 4225$

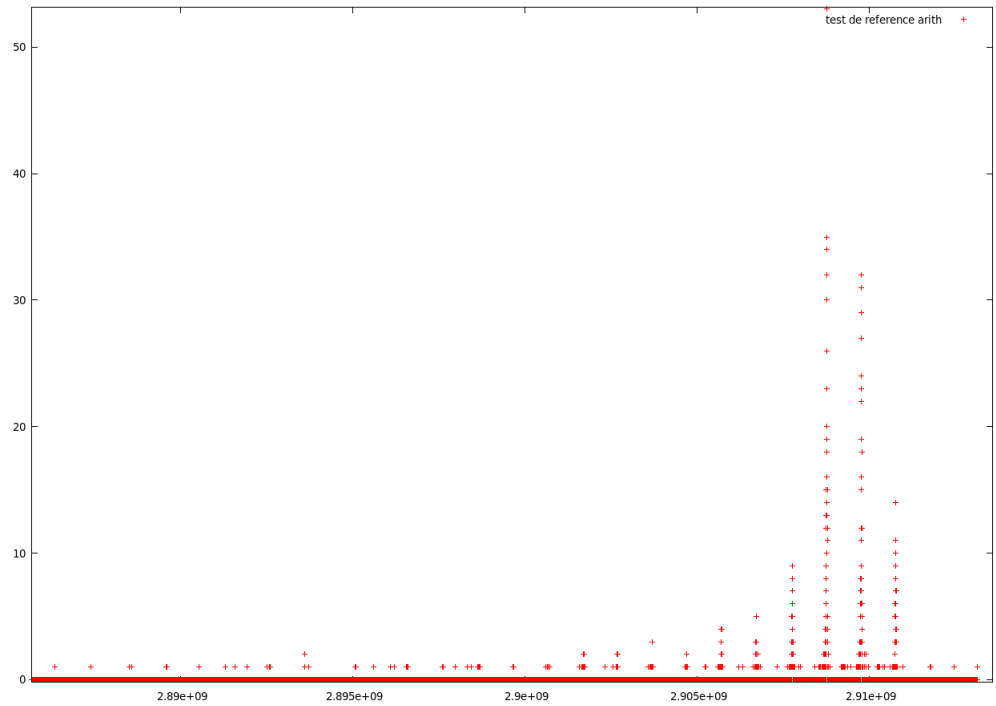


FIG. E.11 – Carte B, CAD 4, XP, $L = \texttt{P2}^2 = 4225$

Bibliographie

[1] ETSI TS 131 111 : Digital cellular telecommunications system (phase 2+) ; universal mobile telecommunications system (umts) ; universal subscriber identity module (usim) application toolkit (usat) (3gpp ts 31.111 version 4.16.0 release 4), 2007.

[2] ETSI TS 102 221 : *Smart Cards UICC Terminal Interface : Physical and logical characteristics (Release 7)*. ETSI, 2005.

[3] Pierrick Arlot : Le marché de la carte à puce ne connait pas la crise. Rapport technique, Electronique international, 2008.

[4] Ken Arnold : *Embedded Controler Hardware Design*. LLH Technology Publishing, Lewis Lewis & Helms LLC, 3578 Old Rail Road, Eagle Rock, VA, 24085, 2000.

[5] Eve Atallah, Franck Darrigade, Serge Chaumette, Achraf Karray et Damien Sauveron : A grid of Java Cards to deal with security demanding application domains. *In 6th edition e-Smart conference & demos*, Septembre 2005. Sophia Antipolis, French Riviera.

[6] H. Balinski, E. MacDonnell, L. Chen et K. Harrison : Anti-counterfeiting using memory spots. *In* Springer, éditeur : *Information Security Theory and Practice*, pages 52–67. Third IFIP WG 11.2 International Workshop WISTP, 2009.

[7] E. Baughman, M. Liberatore et B.N. Levine : Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions On Networking, Vol. 15, no. 1*, 2007.

[8] Björk et al. : Designing ubiquitous computing games - a report from a workshop exploring ubiquitous computing entertainment. *Personal and Ubiquitous Computing, Vol. 6, Issuepp. 443-458*, pages 5–6, 2002.

[9] Samia Bouzefrane, Julien Cordry, Hervé Meunier et Pierre Paradinas : Evaluation of Java Card Performance. *In Eighth Smart Card Research and Advanced Application Conference CARDIS*, Egham, United Kingdom, Septembre 2008.

[10] S. Brands : *Rethinking Public Key Infrastructures and Digital Certificates : Building in Privacy*. Cambridge MIT Press, 2000.

[11] Werner Buchholz : A synthetic job for measuring system performance. *IBM Systems Journal 8(4)*, pages 309–318, 1969.

[12] Clemens H. Cap, Nico Maibaum et Lars Heyden : Extending the data storage capabilities of a Java-based smart card. *In Sixth IEEE Symposium on Computers and Communications (ISCC'01)*. IEEE, 2001.

[13] Jordy Castellà-Roca, Josep Domingo-Ferrer, Jordi Herrera-Joancomati et Jordi Planes : A performance comparison of Java Cards for micropayment implementation. *In CARDIS*, pages 19–38, 2000.

[14] Serge Chaumette, Pascal Grange, Achraf Karray, Damien Sauveron et Pierre Vignéras : Secure distributed computing on a Java Card Grid. Rapport technique 1331-04, LaBRI, Université Bordeaux 1, 2004.

[15] Serge Chaumette et Damien Sauveron : Some security problems raised by open multi-application smart cards. *In 10th Nordic Workshop on Secure IT-systems : NordSec 2005*, Octobre 2005.

[16] Zhiqun Chen : *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison Wesley, 2000.

[17] D. Cheok et al : Human pacman : a mobile, wide-area entertainment system based on physical, social, and ubiquitous computing computing entertainment. *Personal and Ubiquitous Computing, Vol. 8, Number 2*, pages 71–81, 2004.

[18] Cnam-Cédric, Trusted Labs et INRIA POPS : Mesure project functionalities. Rapport technique, ANR, 2007.

[19] Cnam-Cédric, Trusted Labs et INRIA POPS : Mesure project methodology. Rapport technique, ANR, 2007.

[20] Cnam-Cédric, Trusted Labs et INRIA POPS : Mesure project requirements. Rapport technique, ANR, 2007.

[21] Cnam-Cédric, Trusted Labs et INRIA POPS : Mesure project state of the art. Rapport technique, ANR, 2007.

[22] Cnam-Cédric, Trusted Labs et INRIA POPS : Mesure project user guide. Rapport technique, ANR, 2007.

[23] USB Device Working Group Committee : Specification for integrated circuit(s) cards interface devices v1.1, 2005. http ://www.usb.org/developers/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf.

[24] The Standard Performance Evaluation Corp. : Spec. `http ://www.spec.org/`.

[25] The Standard Performance Evaluation Corp. : Spec cpu2006. `http ://www.spec.org/cpu2006`.

[26] The Standard Performance Evaluation Corp. : Specviewperf 10. `http ://www.spec.org/gwpg/gpc.static/vp10info.html`.

[27] The Standard Performance Evaluation Corp. : Spejvm2008. `http ://www.spec.org/jvm2008`.

[28] Wayne R. Cowell, éditeur. *Sources and Development of Mathematical Software*. Prentice-Hall Series in Computational Mathematics, Cleve Moler, Advisor. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1984.

[29] H.J. Curnow et Wichman : "a synthetic benchmark". *Computer Journal, Volume 19, Issue 1*, pages 43–49, 1976.

[30] H.J. Curnow et B.A. Wichman : A synthetic benchmark. *Computer Journal, Volume 19, Issue 1*, pages 43–49, 1976.

[31] J. Dongarra, H. W. Meuer, et E. Strohmaier : Top500 supercomputer sites (14th edition). Rapport technique, University of Tennessee Computer Science, Novembre 1999.

[32] J. J. Dongarra, C. B. Moler, J. R. Bunch et G.W. Stewart : *LINPACK Users' Guide*. SIAM Press, Philadelphia, PA, USA, 1979.

[33] Jean-Michel Douin, Pierre Paradinas et Cédric Pradel : Open Benchmark for Java Card Technology. *In e-Smart Conference*, Septembre 2004.

[34] EEMBC : Grinderbench for the java platform micro edition. Rapport technique, EEMBC, 2006. `http ://www.eembc.com/techlit/datasheets/java2_wp.pdf`.

[35] R. Eigenmann : *Performance Evaluation and Benchmarking with Realistic Applications.* The MIT Press, 2001.

[36] EMVCo : *Integrated Circuit Card Specifications for Payment Systems - Book 1 - Application Independent ICC to Terminal Interface Requirements.* `http ://www.emvco.com/`, Juin 2008.

[37] EMVCo : *Integrated Circuit Card Specifications for Payment Systems - Book 2 - Application specification.* `http ://www.emvco.com/`, Juin 2008.

[38] Monika Erdmann : Benchmarking von Java Cards. Mémoire de D.E.A., Institut für Informatik der Ludwig-Maximilians-Universität München, 2004.

[39] ETSI : Digital cellular telecommunications system (phase 2+) ; specification of the sim application toolkit for the subscriber identity module - mobile equipment (sim - me) interface (gsm 11.14) v5.2.0, 1996.

[40] Mario Fischer : Vergleich von Java und native-chipkarten toolchains, benchmarking, messumgebung. Mémoire de D.E.A., Institut für Informatik der Ludwig-Maximilians-Universität München, 2006.

[41] Gemalto : Simagine. `http ://www.gemalto.com/simagine/`.

[42] Gemplus. *GemXpresso Reference Manual*, 1998.

[43] D. Ghindici, G. Grimaud et I. Simplot-Ryl : An information information flow verifier for small embedded systems. *WISTP 07 LNCS 4462*, pages 189–201, 2007.

[44] Giesecke & Devrient. *Sm@rtCafe Reference Manual*, 1999.

[45] Gilles Grimaud, Pierre Paradinas et Eric Vétillard : Measuring the performance of the Java Card Platform. Java One, Mai 2006.

[46] V. Guyot : *La Carte à Puce, Vecteur de Mobilité.* Thèse de doctorat, Université Paris VI Pierre et Marie Curie, 2005.

[47] Vincent Guyot, Nadia Boukhatem et Guy Pujolle : Smart card performances to handle session mobility. *In ICI.* IFIP/IEEE, Septembre 2005.

[48] G. Heumer et al. : Paranoia syndrome : a pervasive multiplayer game using pdas, rfid, and tangible objects. *In 3rd International Symposium on Pervasive Gaming Applications PerGames 2006*, pages 157–158, 2007.

[49] G. Heumer et al. : Via mineralia : a pervasive museum exploration game. pages 157–158, 2007.

[50] ISO : *ISO/IEC 7816-4 :2003 : Identification cards — Physical characteritics.* International Organization for Standardization, Geneva, Switzerland, 2003.

[51] ISO : *ISO/IEC 7816-12 :2005 : Identification cards — Integrated circuit(s) cards with contacts – USB electrical interface and operating procedures.* International Organization for Standardization, Geneva, Switzerland, 2005.

[52] ISO : *ISO/IEC 7816-4 :2005 : Identification cards — Integrated circuit(s) cards with contacts – Organization, security and commands for interchange.* International Organization for Standardization, Geneva, Switzerland, 2005.

[53] ISO : *ISO/IEC 7816-3 :2006 : Identification cards — Integrated circuit(s) cards with contacts – Electrical interface and transmission protocols.* International Organization for Standardization, Geneva, Switzerland, 2006.

[54] ISO : *ISO/IEC 14443 :2008 Identification cards – Contactless integrated circuit cards – Proximity cards.* International Organization for Standardization, Geneva, Switzerland, 2008.

[55] J2ME MIDP. `http ://java.sun.com/j2me/`.

[56] R. K. Jain : *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1991.

[57] JavaCC - Java Compiler Compiler - The Java Parser Generator, 2007. `https ://javacc.dev.java.net/`.

[58] Java Card 2.2.2 Specification, Avril 2006.

[59] Java Card 3.0 Specification, Mars 2008.

[60] Java Grande Forum Benchmark Suite. `http ://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html`.

[61] Java Grande Forump Benchmark DHPC. `http ://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/`.

[62] L. K. John et Lieven Eeckhout : *Performance Evaluation and Benchmarking.* CRC press, 2006.

[63] S. Jonsson, A. Waern, M. Montola et J. Stenros : Game mastering a pervasive larp. experiences from momentum. *In* Carsten et al., éditeurs : *Proceedings of the 4th International Symposium on Pervasive Gaming Applications, PerGames'07*, pages 31–39, Salzburg, Austria, 2007.

[64] JSR 177 Security and Trust Services API for J2ME (SATSA). `http ://jcp.org/jsr/detail/177.jsp`.

[65] JSR 257 Contactless Communication API. `http ://jcp.org/jsr/detail/257.jsp`.

[66] JSR 268 : Java Smart Card I/O API, Décembre 2006. `http ://jcp.org/en/jsr/detail?id=268`.

[67] Timo Kasper, Dario Carluccio et Christof Paar : An embedded system for practical security analysis of contactless smartcards. *In WISTP 07*, Heraklion, Greece, mai 2007. Springer Verlag.

[68] B. A. Kowal : Java card platform market, roadmap, sun's deliverables. Sun Microsystems Java Card Academic Exchange Session Sophia Antipolis, Septembre 2009.

[69] RSA Laboratories : *PKCS 11 v2.20 : Cryptographic Token Interface Standard.* `http ://www.rsa.com/rsalabs/node.asp?id=2133`, RSA, The Security Division of EMC. 174 Middlesex Turnpike Bedford, MA 01730, 2004.

[70] RSA Laboratories : *PKCS 11 Base Functionality v2.30 : Cryptoki - Draft 4*, Juillet 2009.

[71] A. Lahlou et P. Urien : Sim-filter : User profile based smart information filtering and personalization in smartcard. *UMICS2003 Ubiquitous Mobile Information and Collaboration Systems*, 2003.

[72] Frédéric Levy : Calypso functional specification card application, 2005.

[73] S. Levy : *Crypto : How the Code Rebels Beat the Government Saving Privacy in the Digital Age*. Penguin, Janvier 2002.

[74] David J. Lilja : *Measuring Computer Performance : A Practitioner's Guide*. Cambridge University Press, 2000.

[75] Tim France-Massey MAOSCO : Multos - the high security smart card os, 2005.

[76] Constantinos Markantonakis : Is the performance of smart card cryptographic functions the real bottleneck ? *In 16th international conference on Information security : Trusted information : the new decade challenge*, volume 193, pages 77 – 91. Kluwer, 2001.

[77] F. H. McMahon : Livermore fortran kernels : A computer test of numerical performance range. Rapport technique, Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[78] The MESURE project website. `http ://mesure.gforge.inria.fr`.

[79] Mogi mogi. `http ://www.mogimogi.com`.

[80] Morpheme : Morphmark. `http ://morpheme.co.uk/index.jsp`.

[81] Wojciech Mostowski, Jing Pan, Srikanth Akkiraju, Erik de Vink, Erik Poll et Jerry den Hartog : A comparison of java cards : State-of-affairs 2006. Rapport technique, Radboud University, Nijmegen, Technische Universiteit Eindhoven, Twente University Pays-Bas, 2006.

[82] Nabaztag. `http ://www.nabaztag.com/fr/index.html`.

[83] S. Natkin et C. Yan : User model in multiplayer mixed reality entertainment applications. *In In International Conference on Advances in Computer Entertainment Technology ACE'06, ACM SIGCHI*, pages 14–16, Hollywood, USA, 2006.

[84] S. Natkin, C. Yan, S. Jumpertz et B. Marquet : Creating multiplayer ubiquitous games using an adaptive narration model based on a user's model. *In Situated Play, Conference of Digital Games Research Association, DiGRA edition, Tokyo*, pages 24–28, Japan, 2007.

[85] NFC Forum. `http ://www.nfc-forum.org/home`.

[86] NFC Forum. `http ://www.nfc-forum.org`.

[87] OSGi alliance. `http ://www.osgi.org`.

[88] Platform for Privacy Preferences (P3P) Project. `http ://wwww3.org/P3P/`.

[89] Konstantinos Papapanagiotou, Constantinos Markantonakis, Qing Zhang, William G. Sirett et Keith Mayes : On the performance of certificate revocation protocols based on a Java Card certificate client implementation. *In 20th IFIP International Information Security Conference (Sec 2005) - Small Systems Security and Smart cards*, Mai 2005.

[90] Pierre Paradinas, Samia Bouzefrane et Julien Cordry : Performance evaluation of java card bytecodes. *In* Springer, éditeur : *Workshop in Information Security Theory and Practices (WISTP)*, Heraklion, Grèce, 2007.

[91] PC/SC Workgroup : *Interoperability Specification for ICCs and Personal Computer Systems Part 1. Introduction and Architecture Overview*, volume 1-9. PCSC Workgroup, 2005.

[92] R. Pellerin : GASP/uGASP project. `http ://gasp.objectweb.org`.

[93] R. Pellerin : The moods protocol : a j2me object-oriented communication protocol. *In Mobility Conference 2007*, pages 10–12, Singapore, 2007.

[94] R. Pellerin : *Contribution à l'ingénierie des jeux multijoueur ubiquitaires.* Thèse de doctorat, CNAM-Cedric, 2009.

[95] Phoronix : Phoronix test suite v2.0. Rapport technique, Phoronix Media, 2008.

[96] The PLUG project. `http ://www.capdigital.com/plug`.

[97] Roldan Pozo et Bruce Miller : Scimark 2.0. `http ://math.nist.gov/scimark2`.

[98] P. Pucheral, L. Bouganim, P. Valduriez et C. Bobineau : Picodbms : Scaling down database techniques for the smartcard. *Very Large Data Bases Journal, VLDBJ, 10(), (13 pages) 2001. Extended version of the Best Paper Award of VLDB2000*, pages 2–3, 2001.

[99] Wolfgang Rankl : *Smart Card Applications : Design models for using and programming smart cards.* Wiley, 2007.

[100] Wolfgang Rankl et Wolfgang Effing : *Smart Card Handbook.* Wiley, 2000.

[101] O. Rashid et al. : PAC-LAN : Mixed-Reality Gaming with RFID-Enabled Mobile Phones. *ACM Computers in Entertainment Vol. 4, No. 4*, Octobre 2006.

[102] Karima Rehioui : Java Card Performance Test Framework, Septembre 2005. Université de Nice, Sophia-Antipolis, IBM Research internship.

[103] Ludovic Rousseau et David Corcoran : Muscle : Mouvement for the use of smart card in a linux environment. `http ://linuxnet.com`.

[104] Ludovic Rousseau et David Corcoran : Musclecard cryptographic token framework. `http ://linuxnet.com/musclecard/index.html`.

[105] Damien Sauveron : *Étude et réalisation d'un environnement d'expérimentation et de modélisation pour la technologie Java Card. Application à la sécurité.* Thèse de doctorat, Bordeaux I, 2004.

[106] S. S. Shapiro et M. B. Wilk : An analysis of variance test for normality (complete samples). *Biometrika, 52, 3 and 4*, pages 591–611, 1965.

[107] Nicole Shillington, Travers Waker et Andrew Hutchison : The design of a smart card interface device. Rapport technique, DNA group University of Cape Town, South Africa, 2002.

[108] W. G. Sirrett : *Analysis, Implementation, and Deployment of Behaviour-based Temporally aware Security in Smart Card.* Thèse de doctorat, Royal Holloway, University of London, 2006.

[109] O. Sotamaa : All the world's a botfighter stage : Notes on location-based multi-user gaming. *In* Frans Mäyrä, éditeur : *In Proceedings of Computer Games and Digital Cultures Conference (CDGC'02)*, pages 6–8, Tampere, Finland, 2002.

[110] SPEC : SPEC benchmark release 1.0. *SPEC newsletter 2*, pages 3,4, 1990.

[111] SPEC : *SPEC CPU2006*, volume 34. ACM SIGARCH newsletter, Computer Architecture News No. 4, 2006.

[112] Bluetooth technology : `http ://www.bluetooth.com`.

[113] ZigBee technology : `http ://www.zigbee.org`.

[114] H. Tews et B. Jacobs : Performance issues of selective dsiclosure and blinded issuing protocols on java card. *In* Springer-Verlag, éditeur : *Workshop on Information Security and Practices (WISTP)*, pages 95–111, Septembre 2009.

[115] Top 500, Juin 2009. `http ://www.top500.org`.

[116] Iternational Telecommunication Union : *Recommendation X.509 (08/05)*. ITU-T Publications, Place des Nations 1211 Geneva 20 Switzerland, 2005.

[117] P. Urien et al. : The t2tit research project. introducing hip rfids for the iot. *International Workshop on System Support for the Internet of Things WoSSIoT'07*, 2007.

[118] Verisign, 2008 : `http ://www.verisign.com/`.

[119] D. Vermoen, M. Witteman et G. N. Gaydadjiev : Reverse engineering java card applets using power analysis. *In* Springer, éditeur : *Workshop in Information Security Theory and Practices 2007*, 2007.

[120] Xiaoyun Wang et Hongbo Yu : How to break MD5 and other hash functions. *Infosec*, 2008.

[121] Reinhold Weicker : Dhrystone : A synthetic systems programming benchmark. *Communications of the ACM (CACM), Volume 27, Number 10*, pages 1013–1030, 1984.

[122] WiFi alliance : `http ://www.wi-fi.org`.

[123] C. Yan : *Adaptive Multiplayer Ubiquitous Games : design principles and an implementation framework.* Thèse de doctorat, PhD thesis in a cotutelle research program with Orange Labs and CNAM, Paris, 2007.

[124] J. Yan et B. Randell : A systematic classification of cheating in online games. *In NetGames '05 : Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, New York, USA, 2005.