IronHide : plate-forme d'attaques par entrées-sorties

Fernand Lone Sang[1,2], Vincent Nicomette[1,2], and Yves Deswarte[1,3]
{fernand.lone-sang,vincent.nicomette,yves.deswarte}(@)laas.fr

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Université de Toulouse, INSA, LAAS, F-31400 Toulouse, France
[3] Université de Toulouse, LAAS, F-31400 Toulouse, France

Résumé Les attaquants continuent à faire preuve d'ingéniosité pour contourner les nombreux mécanismes de protection mis en œuvre dans les systèmes informatiques. Aujourd'hui, les scénarios d'attaques qui ciblent les composants logiciels ne reposent plus exclusivement sur l'utilisation d'autres composants logiciels, et peuvent impliquer des composants matériels. Ainsi, nous avons pu constater, ces dix dernières années, des attaques utilisant des contrôleurs d'entrées-sorties (ex. un contrôleur Ethernet, un contrôleur de clavier) ou des périphériques (ex. un iPod FireWire, une souris USB), à des fins, par exemple, d'escalade de privilèges. Malheureusement, cette classe d'attaques hybrides reste encore méconnue à ce jour. Du fait de leur caractère récent, nous n'avons qu'une vision limitée des actions malveillantes qu'un attaquant peut effectuer s'il a le contrôle sur un composant matériel. Dans l'optique d'élargir cette vision, et de proposer une classification de ces attaques, nous avons mis au point un contrôleur d'entrées-sorties capable de générer des requêtes quelconques (valides mais surtout invalides) sur les bus d'entrées-sorties. Cet article présente ses caractéristiques principales et détaille quelques expérimentations qu'il est difficile, voire impossible, d'effectuer avec des contrôleurs d'entrées-sorties sur étagère, et que nous avons pu mener avec succès avec notre contrôleur. Finalement, nous discutons d'autres cas d'utilisation que nous avons envisagés pour ce contrôleur, à la fois d'un point de vue offensif et d'un point de vue défensif.

Mots-clés: Attaques par entrées-sorties, *chipset*, contrôleur d'entrées-sorties, *bus-mastering*, PCI-Express, FPGA

1 Introduction

De nos jours, les systèmes informatiques font partie intégrante de notre quotidien. Nous les utilisons, par exemple, pour travailler, pour échanger des informations, ou pour effectuer des achats. Malheureusement, ces systèmes sont encore régulièrement victimes d'attaques réussies, en dépit des nombreux mécanismes de protection mis en place.

La majorité des malveillances visant ces systèmes s'appuient sur des vecteurs d'attaques liés au logiciel s'exécutant sur le processeur principal : un attaquant peut, par exemple, utiliser des fonctionnalités trop permissives du système (ex. le chargeur de modules noyau, les périphériques virtuels tels que `/dev/mem` ou `/dev/kmem` dans les systèmes UNIX) ou exploiter des erreurs d'implémentations logicielles (ex. les débordements de tampons ou d'entiers, les chaînes de format). Cette classe d'attaques est relativement bien connue à ce jour et de nombreux mécanismes (ex. les piles non-exécutables, le mécanisme d'ASLR, ou les *stack-cookies*) permettent de s'en protéger [21,31]. Depuis quelques années, on constate que les scénarios d'attaques ciblant ces systèmes évoluent et deviennent, chaque jour, plus complexes. En effet, ces attaques ne reposent plus exclusivement sur l'utilisation de composants logiciels, et on observe de plus en plus d'attaques impliquant des composants matériels, tels que le *chipset*, les contrôleurs d'entrées-sorties (ex. un contrôleur Ethernet, un contrôleur de clavier) ou les périphériques (ex. un iPod FireWire, une souris USB). Ainsi, des publications récentes ont présenté des attaques qui reprogramment un contrôleur de clavier [16] afin d'exploiter une vulnérabilité dans la routine de traitement des *System Management Interrupts* (SMI) exécutée par le processeur lorsqu'il est dans le mode *System Management* (SMM) ou qui modifient à distance le comportement d'une carte réseau [15] afin de mener des attaques de type *Direct Memory Access* (DMA).

Nous classons ces attaques hybrides selon deux catégories. La première catégorie d'attaques concerne les attaques qui détournent des fonctionnalités légitimes du matériel, tels que les mécanismes d'accès direct à la mémoire ou d'interruption, à des fins malveillantes. On recense dans la littérature un nombre important de preuves de concept qui utilisent des contrôleurs d'entrées-sorties variés : des contrôleurs PCI programmables [4,8,11,27], des contrôleurs Ethernet [15,37], des contrôleurs ou des périphériques FireWire [3,5,7,12,24], des contrôleurs ou des périphériques USB [13,25], etc. Ces attaques ciblent principalement des composants logiciels chargés en mémoire principale, le noyau du système d'exploitation en particulier. Dans une moindre mesure, ces attaques ciblent également d'autres composants matériels [23,34,35]. Notons qu'un attaquant peut délibérément utiliser de manière inappropriée ces fonctionnalités afin de déclencher des fautes d'interaction. Ces fautes peuvent alors impacter la sécurité du système lorsqu'elles ne sont pas correctement traitées. R. Wojtczuk *et al.* [37] ont démontré, par exemple, qu'en configurant volontairement le registre d'interruption d'un contrôleur Ethernet avec une valeur inappropriée mais bien choisie, ou en initiant des accès de type DMA vers

des adresses précises [4], il est possible de générer des requêtes d'interruption similaires à celles utilisées pour la communication inter-processeurs. Précisons que seuls les processeurs sont supposés pouvoir générer ce type d'interruptions. Comme le *chipset* ne filtre pas de manière adéquate ces requêtes d'interruption illégitimes, un attaquant peut utiliser ce moyen pour déclencher l'exécution d'un code malveillant qui aurait été placé dans la routine de traitement associée à l'une de ces interruptions.

La seconde catégorie d'attaques regroupe les attaques qui modifient le support de contrôle dans le but d'impacter indirectement le comportement de composants logiciels. Nous rappelons que le support de contrôle regroupe tous les éléments essentiels pour le fonctionnement du noyau système d'exploitation. Le *Basic Input/Output System* (BIOS), les registres et les mémoires internes des processeurs, du *chipset* ou des contrôleurs, par exemple, font partie du support de contrôle. Ce support de contrôle stocke des informations (des données et parfois du code) sur lesquelles s'appuient directement des composants logiciels privilégiés, tels que le noyau de système d'exploitation, pour fonctionner. Un attaquant peut alors altérer ces informations de façon à exploiter une faiblesse dans un composant logiciel privilégié au moment de leur utilisation ou leur interprétation. De nombreuses attaques s'appuient aujourd'hui sur le BIOS, au travers de la re-programmation de son *firmware* [2], de l'altération des ROM d'extensions [17,19], des tables ACPI [14,18] ou de méta-données qu'il utilise [39]. Citons à titre d'exemple le maliciel Mebromi [17] qui ajoute une ROM d'extension malveillante au BIOS pour assurer sa persistance. La littérature fait également état d'attaques reposant sur la modification de registres du *chipset* [33,38] ou de contrôleurs d'entrées-sorties [9,16,36].

En dépit des études nombreuses sur le sujet, nous constatons que ces attaques hybrides restent encore méconnues à ce jour. Quelles actions malveillantes peut espérer effectuer un attaquant dans un système informatique s'il parvient, par exemple, à implanter une porte dérobée dans un contrôleur d'entrées-sorties ? Nous distinguons ici deux scénarios d'attaques. Dans le premier scénario, l'attaquant parvient à modifier le masque de la puce afin d'implanter une porte dérobée au moment de sa fabrication. Il lui est alors possible de générer tout type de requêtes sur les bus d'entrées-sorties. Dans le second scénario, l'attaquant parvient à modifier le comportement du contrôleur en vie opérationnelle, en altérant

4. Depuis la révision 2.2 de la norme PCI [28], il est possible de générer des interruptions de type *Message Signaled Interrupt* (MSI) depuis un contrôleur en écrivant dans les registres de l'I/O APIC (contrôleur d'interruption dans le *southbridge*) projetés dans l'espace d'adressage principal de la mémoire (adresses de `0xfee0000` à `00xfee00fff`).

par exemple son *firmware* [10,15,34,35]. Il est alors limité aux requêtes d'entrées-sorties prévues par le fabriquant. Jusqu'à quel point les mécanismes de protection existants, tels que les *Input/Output Memory Management Units* (I/O MMU) [1,20] ou les *Access Control Services* (ACS) [29], sont-ils efficaces ? C'est à ces questions que tentent de répondre nos travaux précédents [22,23], ainsi que ceux que nous présentons dans cet article.

Nous nous intéressons plus précisément ici aux attaques hybrides initiées depuis les contrôleurs d'entrées-sorties. Dans l'optique d'identifier exhaustivement, puis de classifier les actions malveillantes qu'un attaquant peut effectuer avec un contrôleur, nous avons mis au point notre propre contrôleur d'entrées-sorties en utilisant les technologies de logique programmable. La littérature fait état de plusieurs prototypes de contrôleurs d'entrées-sorties similaires [4,8,11,27]. Le nôtre se différencie cependant de ceux-ci par le fait qu'il s'interface directement avec les bus d'entrées-sorties constituant l'architecture PC actuelle, c'est-à-dire les bus PCI Express, et par le fait qu'il soit capable de générer des requêtes quelconques (valides, mais surtout invalides) sur ces bus. En effet, les contrôleurs précités ont été conçus pour s'interfacer avec les bus PCI, au travers d'un connecteur PCI ou PC Card, et répondent à des besoins spécifiques (ex. investigation informatique légale, détection de maliciels).

Les travaux que nous décrivons dans cet article ont été menés sur une architecture matérielle PC. Nous rappelons, en section 2, quelques éléments de cette architecture. Nous décrivons brièvement son organisation, puis nous introduisons les différents mécanismes d'entrées-sorties et nous détaillons finalement comment ces derniers se traduisent au niveau des bus d'entrées-sorties. La section 3 présente ensuite les principales caractéristiques de notre contrôleur, ainsi que son architecture interne. Nous donnons, en section 4, quelques exemples d'expérimentations que nous avons pu mener avec succès avec notre contrôleur et qu'il aurait été difficile, voire impossible, d'effectuer avec des contrôleurs d'entrées-sorties sur étagère. Finalement, la section 5 conclut cet article et discute d'autres cas d'utilisation que nous avons envisagés pour ce contrôleur, à la fois d'un point de vue offensif et d'un point de vue défensif.

2 Quelques éléments d'architecture

Nous introduisons dans cette section quelques concepts fondamentaux sur les architectures de type PC. Pour commencer, nous présentons cette architecture de manière globale, en explicitant les différents éléments qui

la composent. Nous rappelons ensuite les mécanismes qui leur permettent d'interagir. Finalement, nous détaillons comment ces derniers se matérialisent au niveaux des bus d'entrées-sorties en nous concentrant sur un contrôleur.

2.1 Architecture matérielle PC

Sur une architecture de type PC, les entrées-sorties depuis les processeurs se font au travers d'un ensemble de composants appelé *chipset*. Celui-ci interconnecte les processeurs à d'autres composants matériels tels que la mémoire vive, la carte graphique, les cartes réseau, les disques durs, etc. Il est en général divisé en deux parties : le *northbridge* et le *southbridge*, reliés entre eux par un bus propriétaire appelé *Direct Media Interface* (DMI) dans les *chipsets* Intel (cf. figure 1).

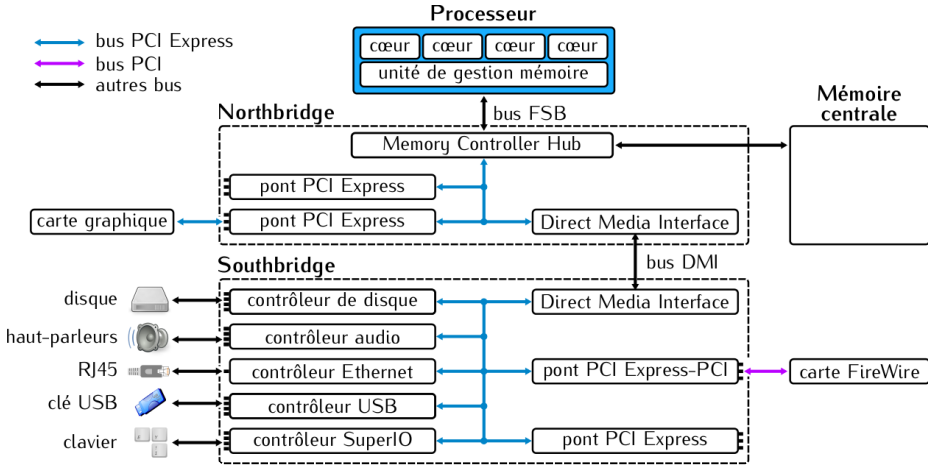

Figure 1. Exemple d'architecture PC Intel

Le *northbridge*, également appelé *Memory Controller Hub* (MCH) ou « pont du nord », relie les processeurs, la mémoire centrale et différents contrôleurs d'entrées-sorties. Ces derniers peuvent être connectés au *northbridge*, soit directement, soit par l'intermédiaire d'un pont PCI Express, lorsque ceux-ci requièrent de larges bandes passantes (ex. les contrôleurs graphiques), soit au travers du *southbridge*, lorsque ceux-ci nécessitent moins de ressources.

Le *southbridge*, également nommé *I/O Controller Hub* (ICH) ou « pont du sud », assure la connexion entre la plupart des contrôleurs d'entrées-

sorties et le *northbridge*. Il fournit des interfaces qui permettent de communiquer sur différents types de bus tels que les bus USB, les bus PCI, les bus ATA/IDE, etc. La communication avec ces bus s'effectue au travers de ponts (ex. un pont PCI Express vers PCI) ou au travers de contrôleurs de bus (ex. des contrôleurs USB, des contrôleurs ATA/IDE, des contrôleurs SMBus).

Au fil des années, l'architecture des *chipsets* a beaucoup évolué. Aujourd'hui, le *northbridge* tend à être directement intégré dans les processeurs et disparait peu à peu des *chipsets*. Par exemple, les processeurs Intel (à partir de Nehalem et surtout Sandy Bridge) et les processeurs AMD (Fusion) actuels intègrent désormais le contrôleur mémoire et quelques contrôleurs rapides. Bien que l'architecture des *chipsets* évolue peu à peu, les concepts que nous présentons dans cette section restent valides.

Comme cet article est dédié aux attaques perpétrées depuis les contrôleurs d'entrées-sorties, nous allons décrire dans la section suivante les différents espaces d'adressage définis dans cette architecture qui permettent aux processeurs et aux différents contrôleurs de communiquer.

2.2 Espaces d'adressage et accès d'entrées-sorties

L'architecture PC définit trois espaces d'adressage distincts, chacun disposant de son propre mécanisme d'accès : l'espace d'adressage principal de la mémoire, l'espace *Port I/O* et l'espace de configuration PCI/PCI Express (cf. figure 2).

L'espace d'adressage principal de la mémoire peut s'étendre jusqu'à 4 gigaoctets dans les systèmes supportant un mode d'adressage de la mémoire sur 32 bits, et jusqu'à 16 exaoctets sur les systèmes supportant un mode d'adressage de la mémoire sur 64 bits. Afin de faciliter les entrées-sorties, le *chipset* projette les registres ou les mémoires internes de certains contrôleurs dans l'espace d'adressage principal de la mémoire. On parle alors de *Memory Mapped I/O* (MMIO). Il rend transparent les accès à ces zones d'entrées-sorties depuis les processeurs et se charge de les aiguiller vers les contrôleurs concernés : les processeurs y accèdent comme ils accèderaient à la mémoire centrale. Afin d'éviter tout conflit avec la mémoire centrale, ces zones d'entrées-sorties sont généralement localisées dans les adresses hautes de l'espace d'adressage principal de la mémoire.

L'espace *Port I/O* (PIO) ou *Port Mapped I/O* (PMIO) est un espace d'adressage logiquement indépendant de l'espace d'adressage principal de la mémoire. Il peut s'étendre théoriquement jusqu'à 4 gigaoctets. Cependant, de nombreuses plate-formes restreignent cet espace d'adressage à

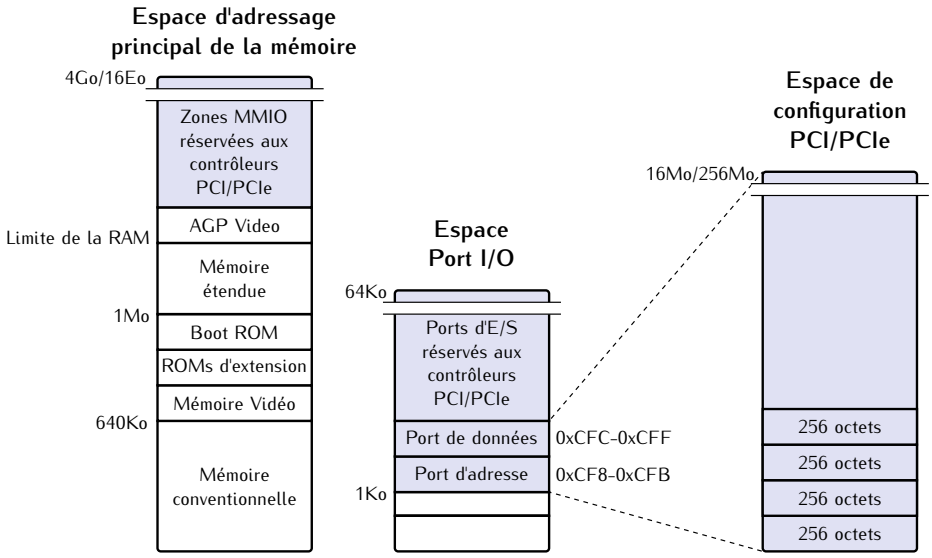

Figure 2. Espaces d'adressage définis dans l'architecture PC

64 kilooctets à cause de la limite d'adressabilité de cet espace (16 bits) dans les processeurs compatibles x86.

L'espace de configuration PCI/PCI Express est un espace d'adressage distinct dans lequel sont projetés les registres de configuration des contrôleurs PCI ou PCI Express. Il s'agit d'un espace d'adressage s'étendant jusqu'à 16 mégaoctets pour le PCI et jusqu'à 256 mégaoctets dans le cas du PCI Express. Les éléments projetés dans cet espace d'adressage permettent, entre autres, d'identifier les contrôleurs et de configurer leur interface PCI ou PCI Express.

Les accès à ces espaces d'adressage peuvent être initiés par les processeurs. Dans le cas particulier où ces accès ciblent les registres ou les mémoires des contrôleurs, on parle d'accès de type *Programmed I/O*. Chaque espace d'adressage dispose d'un mécanisme d'accès depuis les processeurs qui lui est spécifique. Par exemple, l'accès à l'espace PIO se fait par des instructions spécifiques : les instructions `in` (respectivement `out`) en assembleur pour des opérations de lecture (respectivement d'écriture). L'accès à l'espace d'adressage principal de la mémoire, quant à lui, se fait en utilisant des instructions `mov` en assembleur. Finalement, il existe différentes méthodes d'accès à l'espace de configuration. Le lecteur est invité à consulter les spécifications de bus PCI [28] et PCI Express [29] pour davantage de détails à ce sujet.

Il est également possible d'initier des accès à ces espaces d'adressage depuis certains contrôleurs d'entrées-sorties. C'est le cas en particulier des contrôleurs dit *bus-masters* qui sont capables de prendre le contrôle du bus d'entrées-sorties et d'émettre sur ces bus une requête d'accès à l'un de ces espaces d'adressage. Afin de décharger les processeurs des transferts d'entrées-sorties, de nombreux contrôleurs actuels sont capables d'effectuer des accès vers l'espace d'adressage principal de la mémoire. On parle alors d'accès de type *Direct Memory Access* (DMA). Pour terminer, précisons que, pour des raisons de sécurité évidentes, certains accès depuis les contrôleurs d'entrées-sorties peuvent être restreints par le *chipset* [23].

Dans la sous-section suivante, nous allons nous focaliser sur un contrôleur d'entrées-sorties. Nous détaillons son architecture interne et nous décrivons comment se matérialisent les accès à ces différents espaces d'adressage sur les bus d'entrées-sorties actuels qui, nous le rappelons, implémentent le standard de bus PCI Express.

2.3 Architecture et fonctionnement d'un contrôleur PCI Express

Pour transférer des données vers les différents espaces d'adressage, les contrôleurs PCI Express reposent sur des transactions. On distingue trois principales classes de transactions, chacune permettant d'accéder à un espace d'adressage. Les transactions de type *memory*, *I/O* et *configuration* permettent d'accéder respectivement à l'espace d'adressage principal de la mémoire, à l'espace *Port I/O* et à l'espace de configuration PCI/PCI Express. Une dernière classe de transactions, appelée *message*, permet de véhiculer différents types d'informations entre les contrôleurs PCI Express. Elle est utilisée notamment pour signaler des interruptions, des erreurs ou pour véhiculer des messages de gestion d'énergie. La table 1 présente les transactions définies dans le standard PCI Express.

On distingue deux catégories de transactions, *posted* et *non-posted*, selon qu'elle implique ou non une réponse (appelée *completion*) de la part du contrôleur qui traite l'accès. Le paquet *completion* a pour rôle d'acquitter la bonne réception de la requête d'accès et contient éventuellement les données demandées. Les transactions d'écriture correspondent, typiquement, à des transactions *posted* car elles ne requièrent pas de réponse. Les transactions de lecture, quant à elles, correspondent à des transactions *non-posted* car elles requièrent le renvoi d'un paquet *completion* contenant les données demandées.

L'architecture des contrôleurs PCI Express s'inspire fortement du modèle *Open System Interconnection* (OSI). En effet, afin de réduire la com-

Classes	Transactions	Non-Posted or Posted
Memory	Memory Read	Non-Posted
	Memory Write	Posted
	Memory Read Lock	Non-Posted
I/O	IO Read	Non-Posted
	IO Write	Non-Posted
Configuration	Configuration Read (Type 0 et Type 1)	Non-Posted
	Configuration Write (Type 0 et Type 1)	Non-Posted
Message	Message	Posted

Table 1. Transactions PCI Express

plexité de leur logique, ils sont généralement architecturés en couches protocolaires indépendantes (cf. figure 3). Chaque couche utilise les services fournis par les couches inférieures, et définit de nouveaux services pour les couches de niveau supérieur. On distingue principalement trois couches protocolaires : la couche transaction, la couche liaison de données et la couche physique.

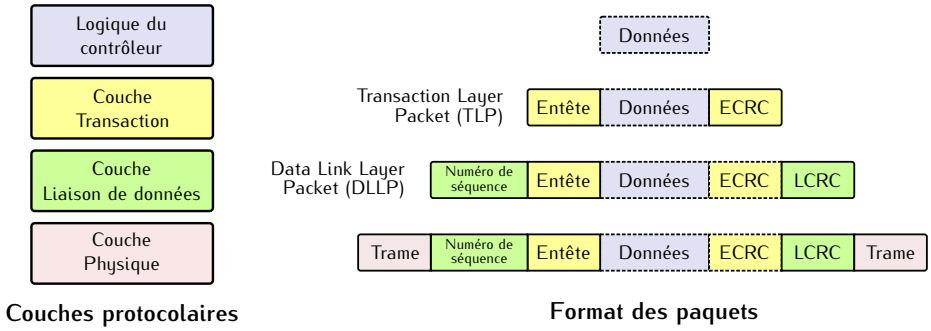

Figure 3. Architecture d'un contrôleur PCI Express

La couche transaction est la couche la plus haute du modèle en couches. Elle se charge principalement du transport de bout en bout des requêtes initiées par la couche logicielle ou des données retournées dans le cas d'une réponse. Elle encapsule ces informations au sein d'un *Transaction Layer Packet* (TLP) et utilise les services fournis par les couches inférieures pour acheminer le paquet jusqu'au contrôleur de destination. Un TLP se compose au minimum d'un entête et contient optionnellement des données

fournies par la couche logicielle et un code correcteur d'erreurs. Les requêtes d'accès en écriture contiennent typiquement des données alors que les requêtes d'accès en lecture n'en nécessitent pas. L'entête dispose d'une partie commune identifiant le type de transaction PCI Express (*memory*, *I/O*, *configuration* ou *message*) contenue dans le paquet. Le reste de l'entête est spécifique au type de transaction. La figure 4 présente un exemple de TLP. Les champs `fmt` et `type` combinés identifient le type de transaction contenu dans le paquet. Il s'agit, dans cet exemple, d'une transaction d'écriture (format 32 bits) [5] à destination de l'espace d'adressage principal de la mémoire. Le champ `RequesterID` précise l'identité du contrôleur qui initie de la transaction, et `Address` indique l'emplacement en mémoire où écrire les données. Le lecteur est invité à se référer aux spécifications du standard PCI Express [29] pour la sémantique des autres champs. Précisons, pour être complet, que la couche transaction fournit également d'autres services tels que le contrôle de flux ou la gestion d'énergie. Nous ne traiterons pas de ces aspects dans cet article.

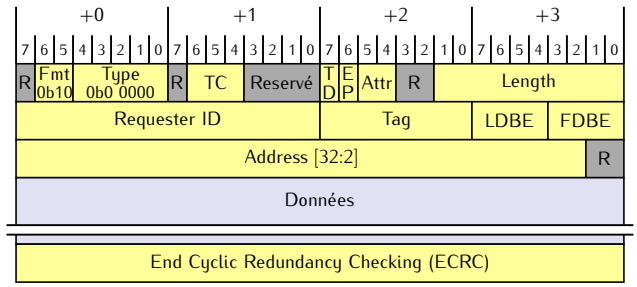

Figure 4. Format d'un TLP de type `MemoryWriteRequest` (format 32 bits)

La couche liaison de données se charge du transport des TLP entre contrôleurs PCI Express adjacents, c'est-à-dire connectés aux mêmes pistes (*lanes*) PCI Express. Elle s'assure en particulier de la fiabilité de ces échanges en utilisant des mécanismes de détection et de correction d'erreurs et gère les éventuels déséquencements de paquets. Les paquets traités au niveau de cette couche sont appelés *Data Link Layer Packets* (DLLP). Finalement, la couche physique se charge d'émettre ou de recevoir les DLLP sur les pistes PCI Express.

5. Nous précisons que pour effectuer des accès aux adresses localisées au delà de 4 gigaoctets, les contrôleurs doivent générer des transactions dans le format 64 bits. Le champ `Address` dans le TLP s'étend alors sur 8 octets.

Dans la section qui suit, nous présentons les caractéristiques principales du contrôleur que nous avons mis au point, et nous détaillons les choix architecturaux que nous avons effectués pour atteindre nos objectifs.

3 Le contrôleur d'entrées-sorties IronHide

Dans l'optique d'identifier exhaustivement, puis de classifier les actions malveillantes qu'un attaquant peut effectuer avec un contrôleur, nous avons mis au point notre propre contrôleur d'entrées-sorties en utilisant les technologies de logique programmable. Nous avons appelé notre prototype IronHide. Dans la sous-section qui suit, nous présentons les caractéristiques principales qui font sa singularité par rapport à d'autres contrôleurs d'entrées-sorties similaires.

3.1 Caractéristiques

Il existe aujourd'hui de nombreux contrôleurs d'entrées-sorties dédiés à l'évaluation de la sécurité des systèmes informatiques qui sont implémentés à partir de technologies de logique programmable [4,8,11,27]. Ces contrôleurs se présentent comme des cartes d'extension PCI que l'on connecte aux *chipsets* actuels par le biais de connecteurs PCI ou PC Card disponibles sur la carte-mère. Les protocoles de bus PCI et PCI Express étant physiquement incompatibles, ces derniers reposent sur un pont PCI vers PCI Express intégré au *chipset* pour se connecter aux bus d'entrées-sorties PCI Express. Étant donné que nous ne maîtrisons pas la manière selon laquelle les requêtes PCI sont traduites sur les bus d'entrées-sorties PCI Express au niveau de ce pont, le fait de réutiliser un de ces contrôleurs ne nous a pas paru une solution adaptée pour identifier exhaustivement les éventuels vecteurs d'attaques depuis les contrôleurs d'entrées-sorties. Aussi, nous avons décidé de développer notre propre contrôleur. Ce dernier se connecte directement aux bus PCI Express via un connecteur PCI Express ou Express Card.

Avec comme objectif de couvrir au maximum les vecteurs d'attaques possibles depuis les contrôleurs d'entrées-sorties, nous avons conçu notre contrôleur afin qu'il puisse générer des requêtes quelconques, à la fois valides mais également invalides, sur les bus PCI Express. Par ailleurs, il est extrêmement polyvalent puisque son comportement est entièrement programmable. Il est envisageable de l'utiliser aussi bien pour faire de l'injection de fautes sur les bus PCI Express, par exemple, que pour émuler

une carte d'extension PCI Express quelconque. Il suffit pour cela de modifier le logiciel, écrit entièrement en langage C, exécuté par le processeur qu'il embarque.

Enfin, nous avons conçu son architecture pour être facilement extensible, en fonction de nos besoins, avec des contrôleurs de périphériques divers. Il est par exemple possible d'y intégrer un contrôleur série RS-232 pour surveiller l'évolution du contrôleur d'entrées-sorties, un contrôleur Ethernet pour interagir avec lui à distance, un contrôleur USB pour enregistrer sur un périphérique USB les résultats de nos expérimentations, etc.

3.2 Architecture

La figure 5 présente l'architecture interne de notre contrôleur. Puisque ce contrôleur a pour vocation de tester les éventuelles fautes d'interactions entre contrôleurs, il nous est indispensable de pouvoir envoyer des paquets PCI Express quelconques depuis la couche protocolaire transaction. Pour cela, nous avons utilisé l'*IP Core Xilinx LogiCORE IP Endpoint Block Plus for PCI Express* [41] qui permet, entre autres, de configurer le bloc PCI Express inclus nativement dans le FPGA et fournit une interface pour émettre ou recevoir des TLP PCI Express. Nous avons ensuite interfacé cette unité logique avec un système embarqué composé d'un processeur, de différentes mémoires, d'un contrôleur d'interruptions et optionnellement de contrôleurs de périphériques divers (ex. des contrôleurs Ethernet, série RS-232). Ce système embarqué constitue la logique du contrôleur.

Il existe deux manières pour interconnecter l'*IP Core* PCI Express avec notre système embarqué. La première méthode consiste à utiliser une autre unité logique commercialisée par Xilinx, appelée *Xilinx LogiCORE IP PLBv46 RC/EP Bridge for PCI Express* [40]. Il s'agit d'une unité logique (payante) qui implémente entièrement le protocole PCI Express. Elle est capable de traiter automatiquement les TLP reçus depuis l'interface PCI Express et fournit une interface pour émettre quelques requêtes d'entrées-sorties prédéfinies (ex. requêtes d'accès à l'espace d'adressage principal de la mémoire pour le mécanisme d'accès direct à la mémoire). Cette interface est cependant très peu flexible, et ne permet pas de générer des requêtes d'entrées-sorties quelconques (en particulier, celles avec un format invalide). Ces raisons nous ont découragé d'utiliser cette solution. Nous avons donc opté pour la méthode d'interconnexion qui consiste à interconnecter les deux composants de manière spécifique, et en développant notre propre *IP Core*. Notre unité logique se présente comme un contrôleur de bus PCI Express que l'on peut connecter à un bus PLBv46. Elle offre une interface pour émettre des paquets quelconques sur les bus PCI

Express ou pour en recevoir. Ces derniers sont stockés temporairement dans une mémoire partagée.

Le processeur intégré dans le système embarqué constitue le cœur de notre contrôleur et nous permet d'obtenir les propriétés de modularité et de polyvalence voulues pour notre contrôleur. Le logiciel qu'il exécute traite les différents événements des contrôleurs de périphériques que nous avons configurés dans le FPGA. Il se charge en particulier de traiter les paquets reçus par l'unité logique PCI Express et implémente partiellement le protocole PCI Express. Concrètement, lorsqu'un paquet est reçu depuis les bus PCI Express, le pont PCIe vers PLBv46 le stocke temporairement dans une mémoire partagée avec le processeur et déclenche une interruption pour l'informer de cet événement. Une routine d'interruption vient alors lire ce paquet puis effectue les traitements associés. Elle vérifie que les différents champs de ce paquet (c'est-à-dire les entêtes et éventuelles données) sont conformes au protocole et construit un paquet *completion* lorsqu'une réponse est requise. Ce paquet *completion* est copié dans une autre mémoire partagée et est émise automatiquement par le pont PCIe vers PLBv46. Notez que le processeur embarqué peut également utiliser cette seconde mémoire partagée pour initier de lui-même des requêtes d'entrées-sorties valides ou invalides sur les bus PCI Express.

Finalement, pour surveiller le comportement du système embarqué et pour lui permettre de communiquer avec l'extérieur (ex. une machine distante), l'architecture de notre contrôleur d'entrées-sorties peut être étendue par des contrôleurs de périphériques divers, tels que des contrôleurs Ethernet, des contrôleurs séries RS-232 ou des contrôleurs USB. À terme, il serait intéressant d'utiliser ces contrôleurs divers pour implémenter des cartes d'extension PCI Express à part entière, tel que des cartes Ethernet, des cartes USB, etc.

3.3 Implémentation

Cette architecture a été implémentée avec succès sur une carte de développement Pico E-17 [30] vendue par Pico Computing. Il s'agit d'une carte de développement au format Express Card qui dispose d'un FPGA Xilinx Virtex-5 FX70T, d'une mémoire vive (DDR2-SDRAM) de 256 mégaoctets, d'une mémoire flash de 64 mégaoctets et de différents contrôleurs de périphériques (ex. plusieurs contrôleurs Ethernet et un contrôleur série RS-232). Pour programmer cette carte, nous avons principalement utilisé les outils de développement commercialisés par Xilinx. Nous avons spécifié, par exemple, la logique du pont PCI Express vers PLBv46 en utilisant *Xilinx Integrated Software Environment* (ISE). Nous rappelons

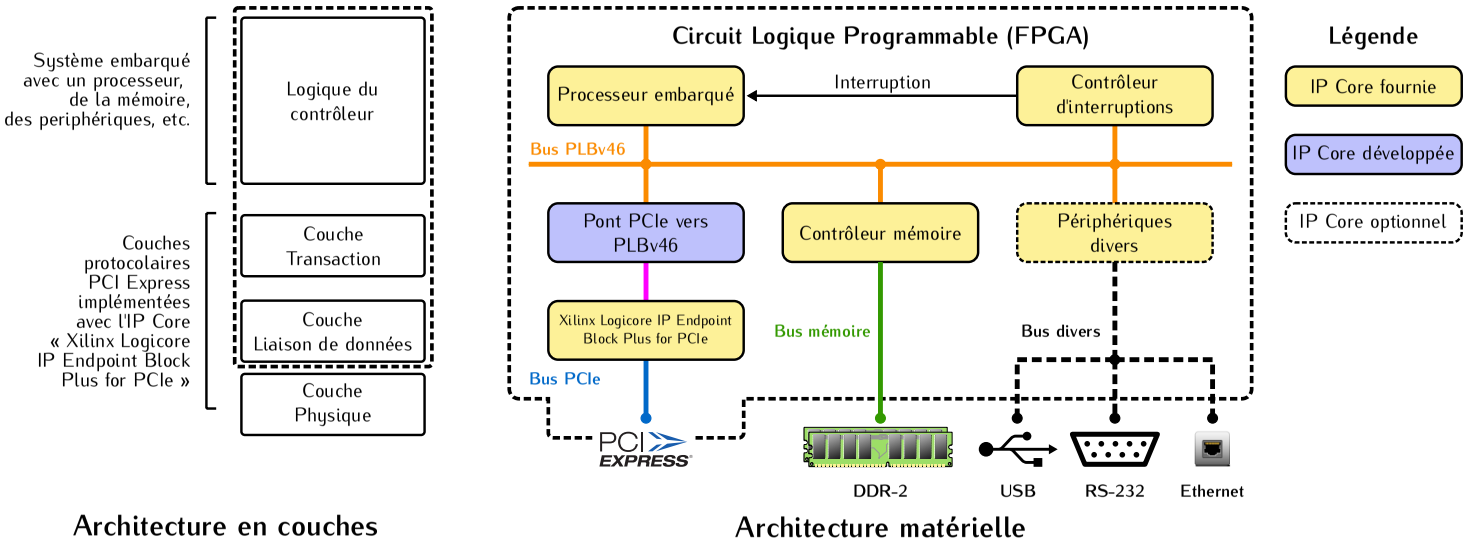

FIGURE 5. Architecture de notre contrôleur

que cet outil permet d'analyser et de faire la synthèse de composants décrits en *Hardware Description Language* (HDL). Nous avons ensuite interconnecté les différents éléments qui composent le système embarqué en utilisant *Xilinx Embedded Development Kit* (EDK). Finalement, le logiciel exécuté par le processeur embarqué, entièrement écrit en langage C, est compilé puis chargé dans la carte de développement avec *Xilinx Platform Studio Software Development Kit* (SDK).

Notre implémentation est compatible avec toutes les cartes de développement à base de FPGA Xilinx Virtex-5. En adaptant la configuration spécifique à la carte de développement que nous avons utilisée (c'est-à-dire les contraintes temporelles, de placement, etc.), il est théoriquement possible de porter notre implémentation sur une autre carte de développement. Une adaptation de notre implémentation sur la carte de développement MLX-1000-XC5V [26] fabriquée par ModularLogix (respectant le format PCI Express) est actuellement en cours.

4 Quelques exemples d'expérimentations

Cette section détaille les premières expérimentations que nous avons effectuées avec notre contrôleur une fois que celui-ci a été mis au point. D'autres expérimentations sont actuellement en cours et nous espérons obtenir des résultats tout aussi intéressants dans les mois à venir. Comme nos travaux visent à étudier exhaustivement les attaques depuis les entrées-sorties, et que l'interface fournie par la majorité des contrôleurs sur étagère est généralement restreinte aux requêtes d'accès à l'espace d'adressage principal de la mémoire (ex. pour le mécanisme d'accès direct à la mémoire), nous avons cherché à présenter dans cette section des expérimentations difficiles, voire impossibles, à mettre en œuvre depuis des contrôleurs conventionnels. Ainsi, les prochaines sous-sections détaillent ces expérimentations atypiques ainsi que les résultats associés. Afin de mieux les cerner, nous commençons par décrire la plate-forme d'expérimentation.

4.1 Description de la plate-forme d'expérimentation

La figure 6 présente la plate-forme que nous avons utilisée pour nos expérimentations. Elle se compose principalement de deux machines qui jouent respectivement le rôle d'une *cible* et d'un *moniteur*. La machine *cible* représente la machine pour laquelle nous souhaitons étudier le comportement du *chipset* en réponse à des requêtes d'entrées-sorties diverses.

Ces requêtes d'entrées-sorties sont initiées par notre contrôleur qui est relié à l'un des connecteurs d'extension PCI Express ou Express Card disponibles sur la carte-mère de la machine étudiée. Nous l'avons programmé pour servir de serveur mandataire (*proxy*) PCI Express pour la machine *moniteur* : il émet tous les paquets reçus depuis l'interface Ethernet en provenance de la machine *moniteur* vers les bus PCI Express de la machine *cible* et renvoie vers la machine *moniteur* les réponses éventuelles. Pour cela, le logiciel embarqué dans le contrôleur implémente une pile TCP/IP et contient un serveur TCP qui relaie les paquets entre la machine *moniteur* et les bus d'entrées-sorties. Le contrôleur est configuré avec une adresse IP statique (`192.168.1.10`) et le serveur TCP embarqué attend de nouvelles connexions sur le port `65535`. Il est à noter que nous aurions pu gagner en performance en choisissant l'utilisation d'un serveur UDP plutôt qu'un serveur TCP. Nous avons préféré utiliser le protocole TCP, de façon à fiabiliser les échanges et ainsi éviter les problèmes liés aux pertes de paquets réseau par congestion du contrôleur ou de la machine *moniteur* lors de nos expérimentations. Afin d'éviter toute ambiguïté, nous précisons que les adresses réseau que nous avons utilisées sont propres au contrôleur et ne sont pas liées aux éventuelles cartes réseau de la machine *cible*. Les trames reçues par le contrôleur sont directement traitées par sa pile TCP/IP interne et ne sont pas vues par le système d'exploitation de la machine *cible*. En plus d'une interface Ethernet, nous avons ajouté au contrôleur une interface série RS-232 sur laquelle les activités du contrôleur sont régulièrement reportées.

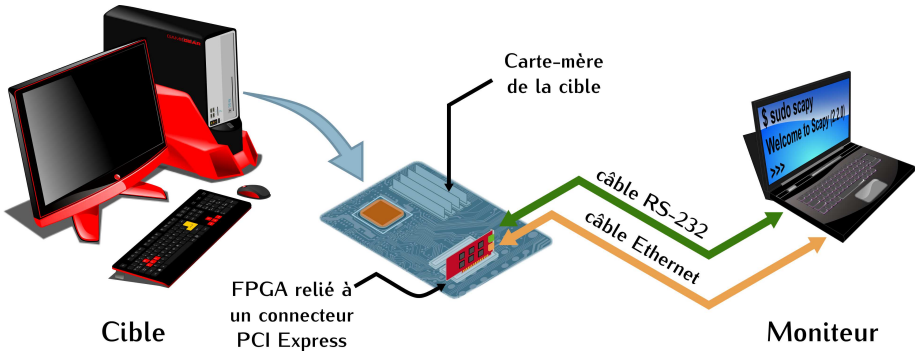

Figure 6. Plate-forme d'expérimentation

La machine *moniteur* est celle qui va définir les transactions PCI Express à émettre sur la machine *cible*. Pour construire et disséquer les *Tran-*

saction Layer Packets (TLP) PCI Express, nous utilisons l'outil réseau Scapy [6]. Comme ce dernier n'implémente pas nativement le protocole PCI Express, nous l'avons étendu pour intégrer ce nouveau protocole [6]. Scapy est également utilisé pour encapsuler les TLP PCI Express forgés dans un paquet TCP et se charge de les envoyer au contrôleur connecté à la machine *cible*.

4.2 Résultats expérimentaux

Dans cette sous-section, nous détaillons quelques expérimentations que nous avons menées avec notre contrôleur. Nous pouvons les classer parmi les attaques hybrides qui détournent des fonctionnalités légitimes du matériel (cf. section 1). Elles illustrent les possibilités d'attaques par entrées-sorties qui s'offrent à un attaquant avec un tel contrôleur.

Écoute sur les bus d'entrées-sorties. La première expérimentation que nous avons mise en place avec notre contrôleur consiste à effectuer de l'écoute passive sur les bus d'entrées-sorties. Nous souhaitons identifier, par cette expérience, les différentes interactions qui peuvent exister entre un contrôleur quelconque et les autres composants. Avant d'analyser les résultats obtenus, nous précisons que les bus PCI Express sont organisés de manière hiérarchique, et que les différents contrôleurs PCI Express sont interconnectés par des liaisons point-à-point. La notion de bus d'entrées-sorties est alors purement logique, et les contrôleurs en amont dans la hiérarchie (généralement des ponts PCI Express) se chargent de commuter les paquets entre différentes liaisons point-à-point et donnent ainsi l'illusion d'être sur un bus physique. Par conséquent, nous recevons, sur l'interface PCI Express de notre contrôleur, uniquement les transactions qui lui sont destinées et les éventuelles transactions diffusées à tous les contrôleurs.

La figure 7 présente notre contrôleur tel qu'il est vu par le système d'exploitation dans la machine *cible*. Pour nos expérimentations, nous avons gardé les identifiants par défaut de notre carte de développement (Pico E-17 assemblée par Pico Computing). Notez que notre contrôleur ne dispose pas de mémoire interne projetée dans l'espace d'adressage mémoire principale ni dans l'espace *Port I/O*. Bien que nous n'ayons développé ni installé de pilote pour cette carte de développement sur la machine *cible*,

6. Nous pensons que notre extension peut éventuellement être utilisée dans un autre contexte. Aussi, nous planifions de diffuser les codes-sources associés et une première version sera disponible très prochainement.

remarquez que celle-ci est malgré tout reconnue et activée par défaut par le système d'exploitation.

```
user@cible$ lspci -v
[...]
0e:00.0 Memory controller: Pico Computing Device 0e17 (rev 01)
    Subsystem: Device 4658:0000
    Flags: bus master, fast devsel, latency 0, IRQ 10
    Expansion ROM at f1e00000 [disabled] [size=1M]
    Capabilities: <access denied>
```

Figure 7. Détection de notre contrôleur par la machine *cible*

La figure 8 présente un extrait des commandes exécutées sur la machine *moniteur* et les résultats qui lui ont été renvoyés. Puisque nous recevons un nombre important de paquets du contrôleur, nous avons uniquement gardé, sur cette figure, deux exemples de transactions que nous avons reçues.

Nous remarquons que notre contrôleur reçoit deux types de transactions : des transactions de type *memory* (à gauche) et des transactions de type *message* (à droite). Nous rappelons que les transactions de type *memory* sont utilisées pour des accès aux mémoires internes des contrôleurs qui sont projetées dans l'espace d'adressage principal de la mémoire. Cet accès est effectué uniquement au démarrage de la machine *cible*, et correspond en réalité au chargement des ROM d'extensions par le BIOS, lesquelles contiennent des routines d'initialisation des contrôleurs fournies par les constructeurs, et exécutées par le BIOS. Le champ `reqID` précise que cet accès est initié par le processeur, au travers du contrôleur mémoire situé dans le *northbridge* dont l'identifiant de bus est `00:00.0` (c'est-à-dire, *bus* 00, *device* 00 et *function* 0).

Nous recevons également régulièrement des transactions de type *message*. Ce type de transactions véhiculent différents types d'informations entre les contrôleurs PCI Express. Elles sont utilisées notamment pour signaler des interruptions, des erreurs ou pour véhiculer des messages de gestion d'énergie. Il s'agit ici d'une requête propre aux *chipsets* Intel. En effet, le champ `msgcode` positionné à la valeur `0x7f` désigne un message spécifique aux constructeurs, et nous reconnaissons le constructeur Intel par le `vendorID` positionné à `0x8086`. Malheureusement, ce type de transaction *message* n'est pas documenté dans les spécifications du *chipset* que

```
user@moniteur$ sudo scapy
Welcome to Scapy (2.2.0)
>>> pkts = sniff(count=0, filter='tcp and src port 65535',
...              prn=lambda p:TLP(str(p[TCP].payload)).show())
[...]
###[ MemoryRequest ]###        ###[ MessageRequest ]###
  fmt        = 3DW HDR no data    fmt        = 4DW HDR with data
  type       = MRd32              type       = MsgD
  rsvd0      = 0x0L               rsvd0      = 0x0L
  tc         = 0x0L               tc         = 0x0L
  rsvd1      = 0x0L               rsvd1      = 0x0L
  attr       = 0x0L               attr       = 0x0L
  rsvd2      = 0x0L               rsvd2      = 0x0L
  th         = 0x0L               th         = 0x0L
  td         = 0x0L               td         = 0x0L
  ep         = 0x0L               ep         = 0x0L
  attr2      = 0x0L               attr2      = 0x0L
  at         = 0x0L               at         = 0x0L
  length     = 1L                 length     = 1L
  reqID      = 0:0.0              reqID      = 0:0.0
  tag        = 0x18               tag        = 0x0
  ldbe       = 0x0L               msgcode    = 0x7FL
  fdbe       = 0xfL               bfdID      = 0:0.0
  address32  = 0xf1e00000L        vendorID   = 0x8086L
  rsvd3      = 0x0L               specific   = 0x00000080L
                                  data       = ')\x00\x00\x00'
```

Figure 8. Quelques exemples de requêtes reçues par notre contrôleur

nous avons étudié. Nous ne savons pas, pour l'instant, à quoi correspond cette transaction ni les données qu'elle contient.

Non-respect de la configuration matérielle. Pour qu'un contrôleur sur étagère puisse effectuer des accès de type DMA, il est nécessaire que le pilote qui lui est associé positionne le bit *Bus Master Enable* (BME) dans la configuration du contrôleur. Le respect (ou le non-respect) de cette configuration n'engage malheureusement que le contrôleur. Nous avons voulu déterminer, par cette expérimentation, si un mécanisme dans le *chipset*, en l'absence d'une I/O MMU, est en mesure de détecter et de parer tout accès frauduleux initié depuis un contrôleur préalablement configuré par le système d'exploitation pour ne pas pouvoir initier, de lui même, des accès à la mémoire centrale. La figure 9 présente les commandes exécutées sur la machine *cible* pour désactiver la fonctionnalité de *bus-mastering* dans le contrôleur.

Une fois la fonctionnalité désactivée dans le contrôleur, nous avons programmé des accès de type DMA depuis la machine *moniteur*. La figure 10

```
user@cible$ lspci -v
[...]
0e:00.0 Memory controller: Pico Computing Device 0e17 (rev 01)
    Subsystem: Device 4658:0000
    Flags: bus master, fast devsel, latency 0, IRQ 10
    Expansion ROM at f1e00000 [disabled] [size=1M]
    Capabilities: <access denied>
user@cible$ sudo setpci -s 0e:00.0 COMMAND.W=0
user@cible$ lspci -v | grep -B 3 Flags
[...]
0e:00.0 Memory controller: Pico Computing Device 0e17 (rev 01)
    Subsystem: Device 4658:0000
    Flags: fast devsel, latency 0, IRQ 10
```

Figure 9. Désactivation du bit *Bus Master Enable* (BME) sur un contrôleur

présente le type de requêtes que nous avons initié depuis le contrôleur. À notre grand regret, les requêtes d'accès vers l'espace d'adressage principal de la mémoire que nous avons générées ont toutes abouties, et il nous a été possible d'effectuer des accès DMA bien que le système d'exploitation ait délibérément configuré le contrôleur pour empêcher ce type d'accès. L'utilisation d'une I/O MMU est alors indispensable pour bloquer (au moins partiellement) les éventuelles attaques initiées depuis les contrôleurs. Nous rappelons que les I/O MMU désignent des unités de gestion de la mémoire physique dédiées aux contrôleurs d'entrées-sorties. Elles ont été initialement conçues pour virtualiser l'espace d'adressage principal de la mémoire pour les contrôleurs. Avec l'avènement de la virtualisation, elles ont été étendues afin d'assurer également des propriétés d'isolation. Il est aujourd'hui possible de configurer les I/O MMU pour associer chaque contrôleur à un ou plusieurs domaines distincts. Ces dernières s'assurent en particulier qu'un attaquant ne détourne pas un contrôleur associé à un domaine pour accéder frauduleusement (par un accès de type DMA) aux régions de mémoire associées à un autre domaine. Dans notre exemple, le contrôleur n'est associé à aucun domaine étant donné que le bit BME n'est plus positionné. L'I/O MMU (si elle est correctement configurée) bloque alors tous les accès DMA de ce contrôleur.

Usurpation d'identité sur les bus d'entrées-sorties. Une des techniques pour passer au travers du contrôle d'une I/O MMU consiste à usurper l'identité d'un autre contrôleur [32]. En effet, les technologies I/O MMU se basent uniquement sur l'adresse de bus fournie dans le pa-

quet PCI Express (champ `Requester ID`) pour identifier le contrôleur à l'origine de l'accès. Il est alors possible de modifier un contrôleur de façon à ce qu'il émette des paquets utilisant l'identifiant d'un autre contrôleur et d'obtenir en conséquence les mêmes droits d'accès à l'espace d'adressage principal de la mémoire que le contrôleur ciblé. Par cette expérimentation, nous avons voulu identifier les limites pratiques qu'un attaquant pouvait rencontrer avec un tel moyen d'attaque. La figure 10 présente le type de requêtes que nous émettons sur les bus d'entrées-sorties grâce à notre contrôleur. Il s'agit d'un accès en écriture à l'espace d'adressage principal de la mémoire dans lequel nous usurpons l'identité du contrôleur mémoire situé dans le *northbridge*.

```
user@moniteur$ sudo scapy
Welcome to Scapy (2.2.0)
>>> ip = IP(dst='192.168.1.10')
[...]
>>> tlp = MWr32(fmt=0x2, type=0x0, reqID='0:0.0',
...             address32=0x0, data='\xde\xad\xbe\xef')
>>> tlp.show2()
###[ MemoryRequest ]###
  fmt       = 3DW HDR with data
  type      = MWr32
  rsvd0     = 0x0L
  tc        = 0x0L
  rsvd1     = 0x0L
  attr      = 0x0L
  rsvd2     = 0x0L
  th        = 0x0L
  td        = 0x0L
  ep        = 0x0L
  attr2     = 0x0L
  at        = 0x0L
  length    = 1L
  reqID     = 0:0.0
  tag       = 0x2
  ldbe      = 0x0L
  fdbe      = 0x1L
  address32 = 0x0L
  rsvd3     = 0x0L
  data      = '\xde\xad\xbe\xef'
>>> send(ip/TCP(dport=65535, sport=synack.dport,
...             seq=synack.ack, ack=synack.seq+1)/tlp)
.
Sent 1 packets.
```

Figure 10. Transaction `MemoryWriteRequest` avec un identifiant quelconque

Dans cet exemple, nous avons volontairement effectué un accès invalide afin que l'I/O MMU le détecte et le rapporte au système d'exploitation. Puisque le contrôleur mémoire ne dispose pas de tables de configuration qui lui sont associées, l'I/O MMU considère cet accès comme frauduleux et le bloque. Nous nous servons de cet accès invalide pour vérifier que nous avons effectivement usurpé l'identité du contrôleur mémoire. La figure 11 présente un extrait des journaux systèmes rapportant cet accès frauduleux à l'espace d'adressage principal de la mémoire. Nous remarquons que l'I/O MMU accuse à tort le contrôleur mémoire, dont l'identifiant de bus est `00:00.0`, comme étant à l'origine de l'accès frauduleux. Cet identifiant correspond bien à celui que nous avons usurpé avec notre contrôleur d'entrées-sorties.

```
user@cible$ dmesg | grep DMAR
[...]
DMAR:[DMA Write] Request device [00:00.0] fault addr 000000000
DMAR:[fault reason 05] PTE Write access is not set
```

Figure 11. Exemple d'accès frauduleux détecté par l'I/O MMU

Afin d'étudier l'impact d'une usurpation d'identité sur un bus d'entrées-sorties, nous avons connecté notre contrôleur à différents emplacements dans le *chipset*, puis nous avons généré des accès en lecture et en écriture vers l'espace d'adressage mémoire principal en utilisant l'identité d'autres contrôleurs. Nous avons observé qu'un attaquant qui utilise cette technique pour tromper le contrôle d'accès d'une I/O MMU reste tout de même restreint à un nombre limité d'actions malveillantes. Il peut, par exemple, écrire dans les régions de mémoire pour lesquels le contrôleur dont on usurpe l'identité dispose de permissions d'accès en écriture. Il peut également initier des accès en lecture aux régions de mémoire pour lesquels le contrôleur ciblé dispose de permissions d'accès en lecture. En revanche, il ne recevra pas le paquet *completion* en réponse à sa requête d'accès, celui-ci étant acheminé par le *chipset* au contrôleur dont on a usurpé l'identité. Cela s'explique par la manière dont les diverses transactions PCI Express sont routées par le *chipset*. En effet, les transactions de type *memory* sont routées par le *chipset* en fonction de l'adresse à laquelle on souhaite accéder. En revanche, les éventuelles réponses associées (c'est-à-dire, les paquets *completion*) sont routées en fonction de l'identifiant du contrôleur qui a initié la requête. Comme nous avons usurpé

l'identité d'un autre contrôleur, la réponse va naturellement être routée vers le contrôleur pour lequel nous essayons de nous faire passer. Précisons que cette attaque aurait pu être parée si les extensions matérielles *Acess Control Services* (ACS) étaient activées et correctement configurées. Nous rappelons que les ACS désignent une technologie de contrôle d'accès implémentée dans certains contrôleurs PCI Express du *chipset* (son implémentation étant optionnelle). Il est possible, par exemple, d'activer le service *ACS Source Validation* dans les différents contrôleurs PCI Express du *chipset* afin de leur faire vérifier systématiquement que l'identité utilisée par un contrôleur correspond bien à celle qui lui a été attribuée, empêchant ainsi toute possibilité d'usurpation d'identité.

Enregistreur de frappe. Dans cette expérimentation, nous avons profité des fonctionnalités de notre contrôleur pour effectuer des accès vers l'espace *Port I/O*. Nous rappelons qu'il est difficile de mettre en œuvre une telle expérimentation depuis des contrôleurs d'entrées-sorties sur étagère, ces derniers étant généralement restreints aux requêtes d'accès à l'espace d'adressage mémoire principal.

Nous avons observé des résultats différents en fonction des *chipsets* sur lesquels nous avons mené nos expérimentations. Par exemple, sur les *chipsets* supportant les accès *peer-to-peer*, nous avons réussi à faire une image de l'espace *Port I/O* (cf. figure 12). En revanche, sur les autres *chipsets*, le *northbridge* ou le *southbridge* répondent explicitement que la transaction n'est pas supportée. En effet, ces derniers renvoient directement un paquet *completion* avec le statut de transaction positionné à *Unsupported Request*.

Bien que l'espace *Port I/O* soit aujourd'hui peu usité, nous observons que des contrôleurs (ex. le contrôleur de clavier, les contrôleurs VGA, SATA, USB) continuent à y projeter certains de leurs registres. Le contrôleur de clavier projette, par exemple, des registres aux adresses `0x60` et `0x64` qui informent le système d'exploitation des événements du clavier ou de la souris. Nous avons exploité la possibilité de lire ces ports d'entrées-sorties depuis notre contrôleur sur le *chipset* Intel x58 (cf. figure 12) pour implémenter un enregistreur de frappe au clavier. Actuellement, nous sommes capables d'enregistrer les frappes de claviers de type PS/2 et éventuellement des claviers de type USB lorsque l'option *USB Legacy Support* est activée dans le BIOS. Nous rappelons que cette option permet d'utiliser certains périphériques USB (ex. un clavier, une souris, un périphérique de stockage) au démarrage de la machine, bien avant que les contrôleurs USB ne soient initialisés par le système d'exploitation. Dans

```
user@moniteur $ od -t x1 dump-pio-x58-via-controleur
00000000 8c 0a 0f af 24 92 04 ff 00 ff ff ff ff ff ff 00
*
00000020 01 ff ff ff 01 ff ff ff 01 ff ff ff 01 ff aa 00
00000030 01 ff ff ff 01 ff ff ff 01 ff ff ff 01 ff ff ff
00000040 98 10 ac ff ff ff ff ff ff ff ff ff ff ff ff ff
00000050 d0 06 03 ff ff ff ff ff ff ff ff ff ff ff ff ff
00000060 fe 2c ff ff 74 ff ff ff ff ff ff ff ff ff ff ff
00000070 ff 02 7d 01 0b 02 7d 01 ff ff ff ff ff ff ff ff
00000080 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000090 ff 01 00 00 ff ff ff 00 ff 00 00 00 ff ff ff 00
000000a0 0c ff ff ff 0c ff ff ff 0c ff ff ff 0c ff ff ff
000000b0 0c ff a0 7f 0c ff ff ff 0c ff ff ff 0c ff ff ff
000000c0 00 80 d6 69 84 00 ff bf 02 00 fe ef 06 00 fa b5
000000d0 00 00 ff ff ff ff ff ff ff ff ff ff ff ff 00 00
000000e0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
000000f0 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
[...]
```

Figure 12. Espace *Port I/O* vu de notre contrôleur connecté à un *chipset* Intel x58

cette configuration, le BIOS (ou plus précisément la routine de traitement de la SMI) présente les claviers ou les souris USB au système d'exploitation comme étant des périphériques PS/2. Cette expérimentation exploite la possibilité d'initier des accès *peer-to-peer* sur certains *chipsets*. Cette attaque aurait également pu être parée par une configuration adéquate des ACS. En particulier, il aurait été possible de bloquer tout accès *peer-to-peer* en activant le service *ACS Egress Control*.

Ces expérimentations esquissent l'étendue des possibilités qui s'offrent à nous (et aux attaquants) avec un tel contrôleur. Dans les expérimentations présentées dans cet article, nous nous sommes principalement intéressés aux requêtes bien formées et qui ne dévient pas ou qui dévient peu par rapport au protocole PCI Express. Nous utilisons actuellement IronHide pour évaluer la robustesse des *chipsets* vis-à-vis de requêtes invalides (et explicitement interdites) par le protocole PCI Express [7]. Précisons, pour terminer, qu'IronHide a été conçu avec la capacité d'« émuler » un contrôleur PCI Express quelconque. Il pourrait alors être personnalisé pour répondre à des besoins (d'attaques ou de défenses) bien spécifiques. En adaptant son logiciel embarqué, il est possible de l'utiliser, par exemple,

7. Nous devrions pouvoir présenter les premiers résultats de ces nouvelles expérimentations lors de la conférence SSTIC.

pour évaluer la robustesse des pilotes de contrôleurs d'entrées-sorties ou de périphériques.

5 Conclusion et perspectives

Aujourd'hui, maîtriser l'ensemble des composants logiciels s'exécutant sur un système informatique ne suffit plus pour assurer la sécurité de ce système. En effet, les scénarios d'attaques ciblant ce type de systèmes évoluent et deviennent, chaque jour, plus complexes. On a pu observer, ces dix dernières années, l'émergence d'attaques hybrides qui utilisent conjointement des vecteurs d'attaques à la fois logiciels et matériels. Face à ces nouvelles menaces, il devient essentiel de bien connaître la plate-forme matérielle utilisée par le système informatique et de veiller au bon comportement des éléments hétérogènes qui la composent, tels que le *chipset*, les contrôleurs d'entrées-sorties, ou les périphériques. Dans l'optique d'étudier exhaustivement les malveillances qui s'appuient sur l'utilisation des contrôleurs d'entrées-sorties, nous avons mis au point notre propre contrôleur d'entrées-sorties en utilisant les technologies de logique programmable. Ce contrôleur, que nous avons nommé IronHide, présente l'avantage de s'interfacer directement avec les principaux bus d'entrées-sorties constituant l'architecture PC actuelle, à savoir les bus PCI Express, et est capable de générer sur ces bus des requêtes valides mais surtout invalides. De plus, son comportement est entièrement programmable grâce à un processeur embarqué dans le contrôleur, le rendant ainsi polyvalent. Dans cet article, nous avons décrit les choix architecturaux qui nous ont permis d'obtenir les fonctionnalités que nous souhaitions pour notre contrôleur. Nous avons également présenté les résultats de quelques expérimentations que nous avons pu mener avec succès en utilisant ce contrôleur. Ces résultats préliminaires concluants nous motivent à poursuivre notre évaluation des *chipsets* actuels, et nous sommes convaincus d'obtenir d'autres résultats plus avancés et tout aussi intéressants dans les mois à venir.

Nous rappelons qu'il est important d'activer systématiquement les technologies I/O MMU et *Access Control Services* conjointement dès lors qu'elles sont disponibles. Comme nous avons pu l'entrevoir au travers de nos expérimentations, ces deux technologies se complètent et ne sont efficaces qu'en collaborant ensemble. Malheureusement, certains systèmes d'exploitation prennent en compte partiellement (ou ignorent purement et simplement) ces extensions matérielles.

Le caractère polyvalent de notre contrôleur nous permet d'envisager d'autres cas d'utilisation pour celui-ci. En perspective à nos travaux, nous projetons d'exploiter prochainement les capacités de notre contrôleur pour faire du *fuzz-testing* sur le *chipset*. Nous espérons, par cette technique, évaluer la robustesse des *chipsets* actuels et des composants de sécurité qu'ils embarquent vis-à-vis de requêtes d'entrées-sorties invalides. Ces travaux futurs nous permettront d'identifier éventuellement des vulnérabilités au sein de ceux-ci, voire des fonctionnalités cachées. Comme IronHide possède la capacité de se faire passer pour un contrôleur PCI Express quelconque, nous envisageons également de l'utiliser pour implémenter une preuve de concept de contrôleur malveillant qui imiterait le comportement d'un contrôleur sur étagère (ex. une carte Ethernet). Nous pensons en effet qu'un attaquant, avec un budget moyen, peut créer (et commercialiser) des contrôleurs malveillants identiques en tout point à des contrôleurs sur étagère (supposés dépourvus de portes-dérobées), et qu'il est indispensable de prendre en compte ces scénarios d'attaques dans les problématiques d'informatique de confiance.

Finalement, nous avons pensé à utiliser notre contrôleur comme composant de sécurité, en l'intercalant entre d'autres contrôleurs et les bus d'entrées-sorties. Il pourrait alors endosser, par exemple, le rôle d'un outil de détection d'intrusion par analyse comportementale dédié aux entrées-sorties, ou d'un composant cryptographique qui chiffrerait et déchiffrerait les flux entre les différents composants pour parer toute écoute passive sur les bus d'entrées-sorties.

6 Remerciements

Nous souhaitons remercier les différents relecteurs pour leurs conseils et leurs remarques constructives qui ont permis d'améliorer la qualité de cet article. Merci également à Matthieu Herrb, ingénieur de recherche au LAAS-CNRS, pour ses avis et conseils éclairés, et à LiangLiang Huai, étudiant en Master à l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE), pour ses contributions à l'implémentation de ce prototype et pour son aide précieuse concernant les technologies de logique programmable.

Références

1. Advanced Micro Devices (AMD). *AMD I/O Virtualization Technology (IOMMU) - Architectural specification*, February 2009. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf.

2. .aLS and Alfredo. Persistent BIOS infection. *Phrack*, 13(66), June 2009. http://www.phrack.com/issues.html?issue=66&id=7&mode=txt.
3. Damien Aumaitre. Voyage au coeur de la mémoire. In *Actes du 6ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2008)*, pages 378–437, Rennes, June 2008. https://www.sstic.org/media/SSTIC2008/SSTIC-actes/Voyage_Coeur_Memoire/.
4. Damien Aumaitre and Christophe Devine. Virtdbg : Un débogueur noyau utilisant la technologie de virtualisation matérielle VT-x. In *Actes du 8ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2010)*, pages 74–133, Rennes, 9-11 June 2010. https://www.sstic.org/media/SSTIC2010/SSTIC-actes/virtdbg/.
5. Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire - all your memory are belong to us. In *CanSecWest/core05*, 4–5 May 2005. http://md.hudora.de/presentations/#firewire-cansecwest.
6. Philippe Biondi. Scapy. http://www.secdev.org/projects/scapy/.
7. Adam Boileau. Hit by a Bus : Physical Access Attacks with FireWire. In *RUXCON 2006*, October 2006. http://www.ruxcon.org.au/files/2006/firewire_attacks.pdf.
8. Brian Carrier and Joe Grand. A Hardware-based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation*, 1(1) :50–60, February 2004. http://www.digital-evidence.org/papers/tribble-preprint.pdf.
9. K. Chen. Reversing and exploiting an Apple firmware update. In *Blackhat USA*, 29-30 July 2009. http://www.blackhat.com/presentations/bh-usa-09/CHEN/BHUSA09-Chen-RevAppleFirm-PAPER.pdf.
10. Guillaume Delugré. Closer to metal : reverse-engineering the Broadcom NetExtreme's firmware. In *Hack.lu*, Luxembourg, 27-29 October 2010. http://esec-lab.sogeti.com/dotclear/public/publications/10-hack.lu-nicreverse_slides.pdf.
11. Christophe Devine and Guillaume Vissian. Compromission physique par le bus PCI. In *Actes du 7ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2009)*, pages 169–193, Rennes, 3-5 June 2009. https://www.sstic.org/media/SSTIC2009/SSTIC-actes/Compromission_physique_par_le_bus_PCI/.
12. Maximillian Dornseif. 0wned by an iPod - hacking by Firewire. In *PacSec/core04*, 11-12 November 2004. http://md.hudora.de/presentations/#firewire-pacsec.
13. Loïc Duflot. *Contribution à la sécurité des systèmes d'exploitation et des microprocesseurs*. PhD thesis, Université de Paris XI, October 2007. http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lti/these-duflot.pdf.
14. Loïc Duflot and Olivier Levillain. ACPI et routine de traitement de la SMI : des limites à l'informatique de confiance ? In *Actes du 7ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2009)*, pages 131–167, Rennes, 3-5 June 2009. https://www.sstic.org/media/SSTIC2009/SSTIC-actes/ACPI_et_routine_de_traitement_de_la_SMI_des_limite/.
15. Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Quelques éléments en matière de sécurité des cartes réseau. In *Actes du 8ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC*

2010), pages 213–235, Rennes, 9-11June 2010. https://www.sstic.org/media/SSTIC2010/SSTIC-actes/Peut_on_faire_confiance_aux_cartes_reseau/.

16. Alexandre Gazet. Sticky fingers & KBC Custom Shop. In *Actes du 9ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2011)*, pages 180–193, Rennes, 8-10 June 2011. https://www.sstic.org/media/SSTIC2011/SSTIC-actes/sticky_fingers_and_kbc_custom_shop/.
17. Marco Giuliani. Mebromi : the first BIOS rootkit in the wild, 3 September 2011. http://blog.webroot.com/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/.
18. John Heasman. Implementing and detecting an ACPI BIOS rootkit. In *BlackHat DC*, 25-26 January 2006. http://www.securitytechnet.com/resource/rsc-center2/presentation/black/Federal06/BH-Fed-06-Heasman.pdf.
19. John Heasman. Implementing and Detecting a PCI Rootkit. In *BlackHat DC*, 25-26 January 2007. http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf.
20. Intel Corporation. *Intel Virtualization Technology for Directed I/O - Architecture Specification*. Intel Corporation, September 2008. http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf.
21. Éric Lacombe. *Sécurité des noyaux de systèmes d'exploitation*. PhD thesis, INSA de Toulouse, 2009. http://tel.archives-ouvertes.fr/docs/00/46/25/34/PDF/these-eric_lacombe.pdf.
22. Fernand Lone Sang, Éric Lacombe, Vincent Nicomette, and Yves Deswarte. Analyse de l'efficacité du service fourni par une IOMMU. In *Actes du 8ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2010)*, pages 189–214, Rennes, 9-11 June 2010. http://www.sstic.org/media/SSTIC2010/SSTIC-actes/Analyse_de_l_efficacite_du_service_fourni_par_une_/.
23. Fernand Lone Sang, Vincent Nicomette, Yves Deswarte, and Loïc Duflot. Attaques DMA *peer-to-peer* et contremesures. In *Actes du 9ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2011)*, pages 150–179, Rennes, 8-10 June 2011. https://www.sstic.org/media/SSTIC2011/SSTIC-actes/attaques_dma_peer-to-peer_et_contremesures/.
24. Antonio Martin. FireWire memory dump of a Windows XP computer : a forensic approach. Technical report, FriendsGlobal, 2007. http://www.friendsglobal.com/papers/FireWire%20Memory%20Dump%20of%20Windows%20XP.pdf.
25. David Maynor. 0wn3d by everything else - USB/PCMCIA Issues. In *CanSecWest/core05*, 4-5 May 2005. http://cansecwest.com/core05/DMA.ppt.
26. ModularLogix. MLX-1000-XC5V Module – Product Data Sheet, 4 October 2010. http://files.modularlogix.com/Documents/MLX-1000-XC5V/MLX-1000-XC5V%20Datasheet.pdf.
27. Jr. Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium*, 9-13 August 2004. http://www.usenix.org/events/sec04/tech/full_papers/petroni/petroni.pdf.
28. PCI Special Interest Group. *PCI Local Bus Specification – revision 2.3*. PCI Special Interest Group, March 2002. http://www.pcisig.com/specifications/conventional/conventional_pci_23/.

29. PCI Special Interest Group. *PCI Express^{TM} Base Specification – Revision 1.1*. PCI Special Interest Group, March 2005. http://www.pcisig.com/specifications/pciexpress/specifications/.

30. Pico Computing. Pico Computing - the FPGA Computing Experts, 2011. http://www.picocomputing.com/e_series.html.

31. Paul Rascagneres. State of the art of Linux buffers overflows protections. In *Rencontres Mondiales des Logiciels Libres (RMLL)*, 9-14 July 2011. http://2011.rmll.info/IMG/odp/rascagneres.odp.

32. Joanna Rutkowska. Thoughts on Client Systems Security. In *Actes du 9ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 2011)*, 8-10 June 2011. https://www.sstic.org/media/SSTIC2011/SSTIC-actes/ThoughtsonClientSystems/.

33. Joanna Rutkowska and Rafal Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions. In *Blackhat USA*, 7 August 2008. http://invisiblethingslab.com/resources/bh08/part2-full.pdf.

34. Arrigo Triulzi. Project Maux Mk.II - « I 0wn the NIC, Now I want a Shell ! ». In *PacSec/core08*, 12-13 November 2008. http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf.

35. Arrigo Triulzi. The Jedi Packet Trick takes over the Deathstar (or : « Taking NIC Backdoors to the Next Level »). In *CanSecWest/-core10*, 24-26 March 2010. http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III.pdf.

36. Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel Trusted Execution Technology. In *Blackhat DC*, 16-17 February 2009. http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20slides.pdf.

37. Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit : Software Attacks against Intel VT-d. Technical report, Invisible Things Lab (ITL), May 2011. http://www.invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf.

38. Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another Way to Circumvent Intel Trusted Execution Technology - Tricking SENTER into misconfiguring VT-d via SINIT bug exploitation. Technical report, Invisible Things Labs, December 2009. http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf.

39. Rafal Wojtczuk and Alexander Tereshkin. Attacking Intel BIOS. In *Blackhat USA*, 30 July 2009. http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf.

40. Xilinx. LogiCORE IP PLBv46 RC/EP Bridge for PCI Express – Product Specification, 14 December 2010. http://www.xilinx.com/support/documentation/ip_documentation/plbv46_pcie.pdf.

41. Xilinx. LogiCORE IP Endpoint Block Plus for PCI Express – Product Specification, 22 June 2011. http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus/v1_15/pcie_blk_plus_ds551.pdf.