ProTIP : You Should Know What to Expect From Your Peripherals

Marion Daubignard, Yves-Alexis Perez
marion.daubignard@ssi.gouv.fr
yves-alexis.perez@ssi.gouv.fr

ANSSI

Abstract. Rogue peripherals are not reserved to elite attackers with physical access to the machine anymore: practical tools are regularly published and virtualization has led to widespread use of remote services provided by third-parties.
In this paper, we advocate for a new more systematic approach in analyzing PCIe device security. Related works comprise attacks, from proof of concepts to more systematic tools, and the IronHide fuzzing device. But to our knowledge, no formalization or methodical analysis has yet been published. To fill in this gap, we introduce ProTIP, a **Pro**log **T**ester of **I**nformation Flow in **P**CIe networks. This open-source tool implements a model comprising all PCIe components, the CPU, I/OMMU and system RAM. It uses the constraint solving ability of the Prolog engine to enumerate all possible transactions between components.
ProTIP naturally rediscovers all problems listed previously in the litterature, including the need for Access Control Services in switches and their limits. It also highlights two subtleties: firstly, a race condition enabling an attacker to arbitrarily answer read requests and secondly, that I/OMMUs base their access right check on an ID that can change.
While the tool currently models specification-conformant behaviors and rogue endpoints, it is meant to be extended, e.g. with potential specific defects of given hardware, to evaluate the severity of their security impact. Thus, we believe ProTIP can quite efficiently be combined with other tools such as IronHide.

1 Motivation of Our Approach

This paper[1] initiates a new approach to systematically characterize the security issues raised by PCI Express peripherals.

1.1 Rogue Peripherals: an Underestimated Threat ?

While it is widely acknowledged that peripherals form a very concrete threat to computer security, it remains commonplace to think that only

[1] An updated version of this paper (with the URL of the tool) is available on the conference website (`https://www.sstic.org/2017/presentation/protip/`).

attackers with physical access to equipments can make use of rogue peripherals. We argue that this point of view is simplistic.

First of all, hardware is rarely meant to remain in its initial state of deployment. Indeed, for practical reasons, hardware relies on firmware to support all kinds of execution environments. Firmware is a very low-level software embedded in hardware components. It is often meant to be updatable, on both functional and economical accounts. As a result, even a trusted and perfectly sound device can *become* malicious. This should be factored in a pragmatic security analysis. The BadUSB attack [12] has acted as a spectacular reminder of this fact a couple of years ago. The extreme difficulty to efficiently protect oneself against such attacks has renewed interest in the topic. Even so-called controlled information systems massively use USB devices, which tends to make the former remotely accessible. We do not dive into details about USB sticks here, since this is not the topic of our research. In our work, we rather focus on PCI Express (PCIe) devices, which have been shown to suffer from the same problems. Indeed, in [13], Perez et al. pave the way to the remote exploitation of a faulty firmware to gain control of a Network Interface Card (NIC), and then control its host using a Direct Memory Access (DMA) operation to write in system RAM.

Secondly, the current tendency is to extensively use services hosted in datacenters and clouds of various kinds. These solutions rely on virtualization technologies to efficiently share hardware amongst end users, which get access to virtual machines. Service providers have to let users remotely use their hardware by design. In this context, attackers with no physical access to the servers obviously qualify. Providers have to guarantee customers that they can safely ignore that other – potentially ill-intended – customers are hosted on the same machine. In order to hamper the consequences of their potential compromission, peripherals should be compartmentalized. This security goal seems as desirable as user isolation. Providers are aware of the threats, as illustrates a recent publication of Google Cloud Platform [6]. Perhaps less popular but still relevant, multi-level security systems such as the Xen-based solution Qubes [16] or deployments based on Linux-KVM also have to address the same security concerns when it comes to compartmentalization and defense in depth.

1.2 PCI Express Devices and Why They Matter

PCIe is often advertised as a bus protocol, because it was designed to be fully compatible with its predecessor, the PCI bus protocol. While PCI

really deals with components on buses, PCIe components form a *switched fabric of peer devices*. They communicate with each other by sending packets routed by the switches interconnecting them. PCIe peripherals are called endpoint devices. The CPU and memory controller of the system RAM in a board do not directly send PCIe packets on the fabric: they do not have PCIe interfaces. Instead, their demands go through a *root complex*, which translate them into PCIe packets sent in the fabric[2] (Figure 1). The fact that a PCIe endpoint device can communicate with the memory controller of system RAM without intervention of the CPU is the reason why DMA is efficient. Indeed, the CPU does not waste time scheduling the slow interaction with the peripheral. This architectural choice has a serious security impact though. In layman's terms, the Memory Management Unit (MMU) in the CPU remains out of the loop during DMA transactions; hence, it cannot enforce memory access rights. As a result, the whole memory can be read or written by a rogue device. This can be prevented by separating the address space of the peripherals from that of the system memory. In other words, peripherals need to use their own virtual addresses. This requires a component dedicated to performing address translations and access right checks. This is the role of the I/OMMU, placed between peripherals and system RAM.

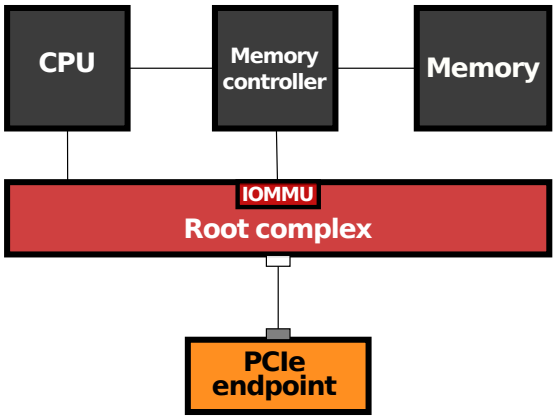

Fig. 1. Simple PCI Express topology (based on work © Mliu92 CC-BY-SA-4.0)

Being an extremely efficient communication protocol, PCI Express is generally used by network interfaces and graphics cards. In addition to that, Thunderbolt devices support PCI Express and are available on the fronts of a lot of commodity computers. Efficient tools and proofs of

[2] To simplify here, we consider one CPU, one memory controller and one root complex, but there can be several instances of each on a given machine.

concepts demonstrating potential threats posed by PCIe peripherals have kept surfacing regularly in the last few years. When in 2010 DMA attacks were rated difficult to carry out and reserved to high-skilled attackers, efficient and available tools now turn them into a practical attack scenario that the knowledgeable majority can perform successfully. The NSA playset features Slotscreamer [3], an inexpensive device based on the PLX USB3380 component converting USB commands into PCI DMA read and write requests. This concept was built on by Ulf Frisk who has developed PCILeech [4, 5], a stable framework to dump the target system memory in a reliable and fast manner. Inception [11] is another tool using DMA writes to allow to log into target machines using an arbitrary password. Closely related to PCIe, Thunderbolt has been proven to be vulnerable to the same attacks [14]. Besides, as a Non-Volatile Memory Express device is basically flash memory attached to a PCIe bus, Ramtin Amin has also been quite creative in its use of the JTAG to relay DMA transactions to the controller of such a device [1].

1.3 Towards a Systematic Analysis of Information Flow in PCIe Fabric

The PCIe standard defines a three layer protocol providing integrity and efficiency guarantees. As all standards, the PCI-SIG specification is rather involved and it can prove difficult to make sense of some requirements. Complexity of the design, backward compatibility issues and the cultural gap between hardware and software communities make it likely that there are some unintended consequences when it comes to security. To work towards a thorough assessment of security issues with PCIe devices, we propose a formal model of possible communication between devices and the rest of the system, to compute all possible flows, including unexpected ones. We have identified four important entities, namely, the CPU, the PCIe network components, the system RAM and the I/OMMU, and abstracted away the rest. Our model is implemented in **ProTIP**, our **Pro**log **T**ester of **I**nformation flow in **P**CIe networks. It currently encompasses legitimate flows – in the sense that it models behaviors conformant to the PCI-SIG specification – and traffic generated by rogue endpoints. It is meant to be extended to capture the specification at a better granularity.

The first goal of the formalization is to understand how to mitigate the harm caused by a rogue peripheral. We mentioned the I/OMMU earlier, but, unsurprisingly enough, ProTIP highlights a few reasons why they are not the panacea they are said to be. Switches need to implement filtering functionalities called Access Control Services (ACS).

Moreover, the standard introduces Address Translation Services, which enables devices to bypass an activated I/OMMU. This feature, designed for performance reasons, obviously violates the guarantees provided by a perfectly configured I/OMMU. The effects of ATS can be thwarted by a careful configuration of ACS in switches and the root complexes. ProTIP enables to characterize configurations offering the best possible compartmentalization.

The tool can also be extended to evaluate the impact of non-conformant or bugged device implementations. We see it as a complementary approach to tools such as IronHide, a PCIe endpoint device able to emit all kinds of PCIe requests to fuzz its counterparts. IronHide was developed by Fernand Lone Sang et al. [7, 9], who were the first to unearth security concerns raised by ACS and ATS. Such a device is needed to test out scenarios found by ProTIP in a real setting.

The second motivation behind ProTIP is to provide insight on how to implement safe peripheral compartmentalization, especially in virtualization settings. Hardware is made available to userland programs through drivers. They execute complex tasks and deal with lots of corner cases, while classically enjoying high privileges. Therefore, they are infamously known to be susceptible to contain all kinds of exploitable bugs. This accounts for the efforts meant to implement or integrate drivers while enforcing the principle of least privilege. The Linux userland framework VFIO [17] or virtual machines dedicated to running drivers illustrate these efforts. These solutions are based on the principle of PCI device delegation, or PCI passthrough, which is basically the idea of allowing access to the raw PCIe device to a less privileged entity. Quite obviously, performance also motivates the will to run less privileged drivers: it lessens the number of costly context-switches. We have established that low-privileged, unfettered access to PCIe devices threatens the system integrity. Thus, we aim at characterizing fair restrictions to impose in order to enforce security.

To do so, ProTIP has to model the consequences of accesses to PCI configuration space. It partly does so already, since it supports the issuance of some configuration requests by the CPU, and their impact on the configuration of the PCIe fabric. This approach has highlighted quite an important subtlety, which is fully documented in the specification but seems rarely used in commodity OSes. A device is indexed by an identifier (called a BDF, for Bus, Device and Function, and also refered to as an ID in the rest of the paper), which is used by the I/OMMU to reference it. However, this identifier is not fixed, and it can legitimately be changed by a configuration request. This identifier impacts routing in the fabric, and

changing it virtually changes the topology of the fabric. We have checked that it can be done coherently. This allows a device to assume the identity of another one, and thus to enjoy its memory access rights. The reconfiguration of a device ID requires access to *both* its configuration space and that of the switch routing requests to it. Consequently, we do not expect that real-life deployments even use this delegation configuration, let alone rely on compartmentalization properties brought by such a configuration. However, it is supported in some distributions such as Xen in permissive mode and possibly in NetBSD. To be fair, both distributions warn against dangers of PCI passthrough. Details are provided in section 5.3.

In our opinion, little known facts like these make the case for the need for a more systematic approach such as the one we propose. It provides valuable insight as to safe practices to implement compartmentalization. To our knowledge, this is the first work in this perspective.

2 Prologue to the Formal Introduction of ProTIP

2.1 About Choosing Prolog to Implement our Model

We were not particularly versed in Prolog before starting this project. We expect that some readers are not familiar with this language. We explain briefly how it works and why it serves our purpose well.

Brief Overview of Prolog Prolog is very different from usual programming languages, such as imperative or functional languages. It is a declarative language used for constraint logic programming. It does not apply functions or procedures to compute an output given some inputs. It rather works with logical statements.

A basic Prolog workflow is as follows. The user writes a *knowledge base*, which is a list of logical formulas; namely, facts or ways to deduce them from one another. Then, the Prolog engine is queried, after launching the interpreter and loading the base manually. A query is a logical predicate provided by the user, and the engine is supposed to find out its truth value. It is possible that the engine never terminates, and it is far from being a detail, but we omit this possibility here. Possible answers of the engine depend on the presence of variables in the query. In their absence, output is simply true or false. Otherwise, the engine enumerates the set of values which can be taken by variables so that the query holds, or false if there are none. This process relies on unification, which offers a way to solve SAT problems in a complete manner: when terminating, the algorithm finds all possible solutions.

Prolog has only one data type, *terms*. Terms are built from atoms, variables and numbers, which can of course be composed together. Anything can be used to define an atom: they have no predefined meaning. The only difference between atoms and variables is syntactic: the latter start by an upper-case letter or an underscore. Figure 1 shows an example of knowledge base, where `good` or `evil` are atoms, while `Character` is a variable.

```
evil(bowser).
friend(yoshi,mario).
friend(peach,mario).
friend(wario,bowser).
brother(luigi,mario).
good(mario).
good(OtherCharacter) :- friend(OtherCharacter,Character), good(
    Character).
lifepts(X) :- X in 0..7.
damage(X) :- X in 1..9.
life(mario,X) :- lifepts(X).
damages(bowser,Y) :- damage(Y), Y#<3.
damages(bowser,Y) :- damage(Y), Y#>=7.
mario_survives(X,Y) :- life(mario,X), damages(bowser,Y), Y#=<X.
```

Listing 1. Example of Prolog program

It comprises *clauses*, one per line. The first six are *facts*, all formed on the same pattern: an atom taking arguments, called a *functor*, is applied to other atoms. The seventh clause is a *rule*. A rule is of the form `conclusion :- things_to_satisfy`. In the program above, the only predefined symbols used are `:-`, to separate both parts of a rule, and the `,` symbol, which is a logical and. The rule captures that a character who is the friend of a good character is a good character, and this is expressed by a conjunction. Notice that in a clause, all occurences of the same variable refer to the same thing. The Prolog engine, when trying to use the clause, instantiates all occurences of the same variable with the same value.

There exists a lot of implementations of Prolog. In fact, it is the object of two ISO standards. ProTIP has been developed using the SWI-Prolog implementation [15]. This implementation comprises a module to deal with finite domains, called CLPFD (for constraint logic programming over finite domains). It allows to express constraints over integers in a simple manner and solve them to obtain sets of integers satisfying the constraints.

In the example, `lifepts` and `damage` hold for variables ranging over finite domains. The `damages` predicate formalizes that bowser can cause damages below 3 life points or above 7 life points. When `mario` encounters `bowser`, its survival depends on his initial number of life points, which

must exceed the damages actually caused by `bowser`. This is modeled by clause `mario_survives`.

Let us see how our toy example can be used. The interpreter provides a command-line interface to query the engine. Example queries are listed in Listing 2. Symbol `?-` is the prompt of the interpreter. Each time a query is performed, the engine tries to find values for variables in the query so that the formula is true. It answers with a first satisfactory valuation of variables, and the user presses `;` to see the next possible instantiation of variables. Symbol `.` marks the end of enumeration. For the first two queries, the engine just has to match with predicates declared in the source file. The second query illustrates that not all arguments of a predicate have to appear as variables in a query. The third query is meant to show all `good` characters, and the engine enumerates `mario`, *declared* as `good`, and `yoshi` and `peach`, which are *deduced* to be `good` from the rule[3]. Eventually, the CLPFD library is put to use in our last query, meant to evaluate the conditions on which `mario` can survive a battle with `bowser`. We observe that the engine deduces that two scenarios yield happy endings. This illustrates that it is able to output preconditions on finite sets of integers which are necessary for a goal to be verified.

```
?- evil(Who).
Who = bowser.

?- friend(X,mario).
X = yoshi ;
X = peach.

?- good(X).
X = mario ;
X = yoshi ;
X = peach ;
false.

?- mario_survives(Life,Damage).
Life in 1..7,
Life#>=Damage,
Damage in 1..2 ;
Life = Damage, Damage = 7.
```

Listing 2. Example of queries to the SWI-Prolog interpreter

A very important thing to keep in mind is that this language relies on what is called the *close world hypothesis*. This means that all trials of proofs exclusively use knowledge facts loaded into the engine. This

[3] This query ends with `false` when we have manually excluded all satisfactory values for `X`, due to the appearance of `good` as a rule conclusion.

has pros and cons, which can both be expressed by the same statement: only what is specified is taken into account. In the second example, this means that `luigi` is not going to be used as a possible instance of a `good` character: there is no occurence of `good(luigi)` in the base, nor is there a way to derive anything from `brother(luigi,mario)`.

Of course, this section only illustrates extremely basic principles about the way the Prolog engine can be used, to provide intuition to a reader who is not familiar with proof engines and logic programming.

How ProTIP benefits from the Prolog engine We need to formalize interactions between elements of a network. This means that we need to model reactions of the elements to triggers. *Trigger events* consist either in the reception of data or in the initiation of an interaction. *Resulting events* are either the emission of data towards an other element or the successful termination of a transaction. These events are formalized by predicates, which are then coupled with physical port numbers in *hop predicates* `hop(TriggerEvt, InPort, ResultEv, OutPort)`. When a hop predicate is true, it captures that `TriggerEvt` happens at `InPort` and that it results in `ResultEv` for `OutPort`. In first approximation, the reader can think of our program as a set of Prolog deduction rules setting conditions to establish a single predicate, `hop(...)`. We call these *hop rules*. Three cases can occur as a result of `TriggerEvt`:

- if the element does not react to the trigger event, typically when it drops data, we do not write any rule: no further transaction is generated. This happens when the input is ill-formed.
- if the element consumes data but does not output anything, `ResultEv` is an acceptance event, and `OutPort` is the physical port which is the recipient of the data.
- lastly, the element can output some data meant for another physical port. In such a case, `OutPort` is the recipient port number and the emitted data is captured by `ResultEv`. This is the case when switches route packets.

The behavior on an element does not have to fit in one rule: for example, switches consume packets intended for them and can route others to one of their ports. This is described by several rules.

Hops are good, but what really matters to us is to list all possible sequences of hops between two points of a network. Provided with hop rules, the Prolog engine perfectly fits our needs.

2.2 Organization of Tool Presentation

In the rest of the article, we adopt an iterative approach to present our work and tool. We choose to alternate presentations of PCIe fundamental concepts with the explanations of how they are formalized in ProTIP and how results are obtained.

Firstly, we introduce in details how PCIe networks proceed to route read and write requests, and what the former look like. Secondly, we focus on fine-grained switches and I/OMMU features that can allow defeating PCIe peripheral compartmentalization if not activated: these are the ACS and ATS that we mentioned in section 1.3. Thirdly, we introduce PCIe configuration requests and the side effects which they can entail.

3 Modeling Read and Write Requests in a PCIe Fabric

3.1 PCI Express Standard Basics

The PCIe specifications are not freely available, but they are well-documented (e.g. in [2]). Here we introduce the fundamental elements needed later on. An example of typical PCIe topology is depicted in Figure 2. The *root complex*, root of the whole hierarchy, connects the device tree to the memory and CPU subsystems. The leaves are called *endpoints* and usually consist in graphics and network interface cards or storage controllers. Endpoints are connected to the root complex[4], either to one of its *root port* directly or through *switches*, responsible for routing traffic. Finally, the *PCIe link* is the physical connection between components.

In a PCIe topology, the terms *up*, *above*, *top* refer to data flowing towards the root complex, while *down*, *below* and *bottom* refer to traffic oriented towards endpoints. It is also possible for two endpoints to achieve peer-to-peer communication. In this case traffic flows upwards from the source endpoint to the closest common ancestor switch, then downwards to the destination endpoint.

Communication between PCIe components is done using a network-like protocol stack with three layers: *physical*, *data link* and *transaction*. We are only interested in the transaction layer in this work.

[4] There are exceptions to this general rule: *root complex integrated endpoints* which are, as their name suggests, directly inside the root complex.

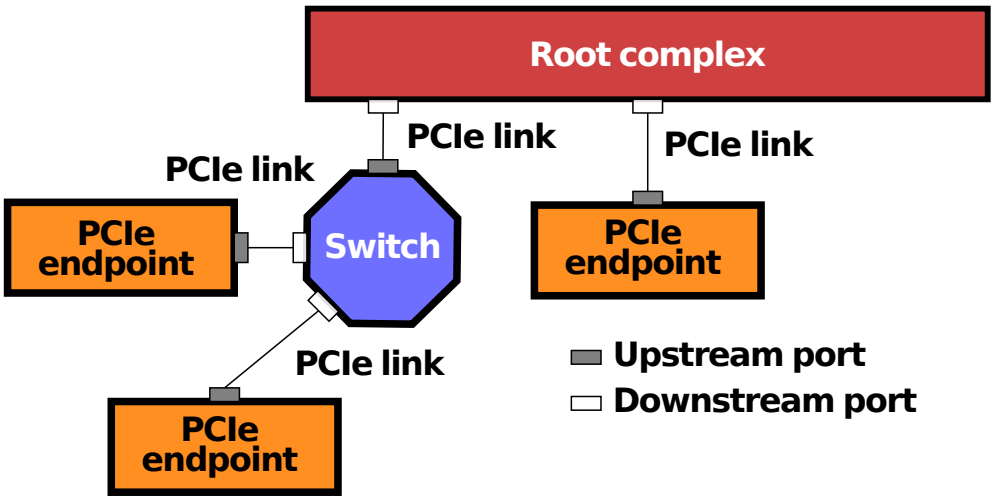

Fig. 2. Example PCIe topology (based on work © Mliu92 CC-BY-SA-4.0)

The transaction layer supports full peer-to-peer communications between endpoints. Packets at this layer are called *transaction layer packets* or *TLP*s.

There are four basic types of transactions: *memory*, *configuration*, *I/O* and *messages*. Transactions start by a *request* which can be *posted* or *non-posted*. While a posted request constitute a transaction on its own, a non-posted request requires a *completion* to finish the transaction. Devices use completions to return data on reception of a read request, or to acknowledge a write request.

Packet routing is achieved using three different mechanisms but only *address-based* and *ID-based* are of interest here[5]. The specification imposes when to use which mechanism. A read transaction is composed of a *memory read request*, routed by address, and the corresponding *completion*, routed by ID. A memory write request is routed by address, but does not involve a completion: it is posted.

We illustrate our description of routing strategies with the example of a CPU access to a memory area exposed by an endpoint. We use the topology depicted on Figure 3, which displays the *base configuration* used throughout the paper. A transaction is emitted by the root complex on behalf of the CPU and targeted at endpoint `04:0.0`. We assume the fabric is already fully initialized; configuration mechanisms are described later in this article. Elements of configuration relevant to routing are stored inside each component in their *configuration space header*.

[5] The third one, *implicit routing*, is only used for specific messages not studied here.

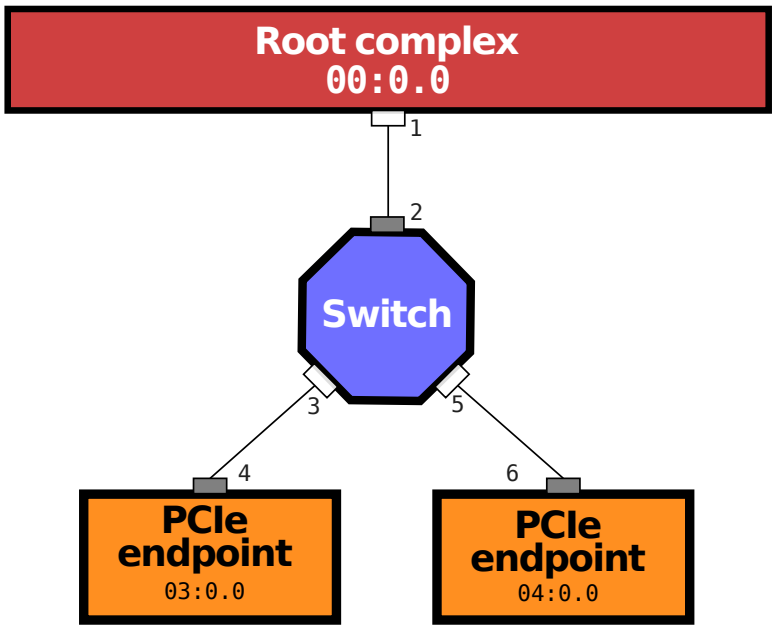

Fig. 3. Base configuration used in our examples

Address-based routing A memory read request TLP (Figure 4) includes a destination *address*, used to route the packet, and a *requester ID* identifying the source device. Port selection inside a switch is pictured on Figure 5. The request bears the address `0xf100400c`, the switch *upstream port* claims it because the address falls within its memory range, defined by the *memory base* and *memory limit* fields of the configuration space header (here `0xf1000000-0xf1005fff`). The switch has two *downstream ports*, each covering a separate address range, so only one is handling the destination address: port 5, with range `0xf1004000-0xf1005fff`. The TLP is forwarded on the attached PCIe link where it reaches an endpoint. The destination address is inside the memory area exposed by the device (defined by the *base address register* field in the configuration space header), leading the endpoint to accept the TLP and process the request.

When a switch receives a request on a downstream port and the destination is neither local nor handled by another downstream port, the request is transmitted on the upstream link. This is what allows peer-to-peer requests to travel upstream to a common ancestor of both peers, which is the first switch having a downstream port configuration matching the destination.

ID-based routing In our example, after processing the read request, the destination endpoint emits a completion (Figure 6) destined to the original

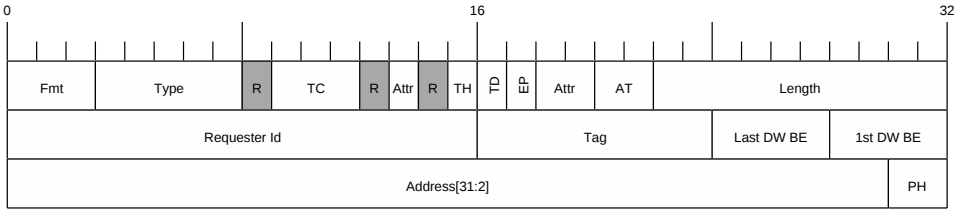

Fig. 4. TLP header (Memory Read Request)

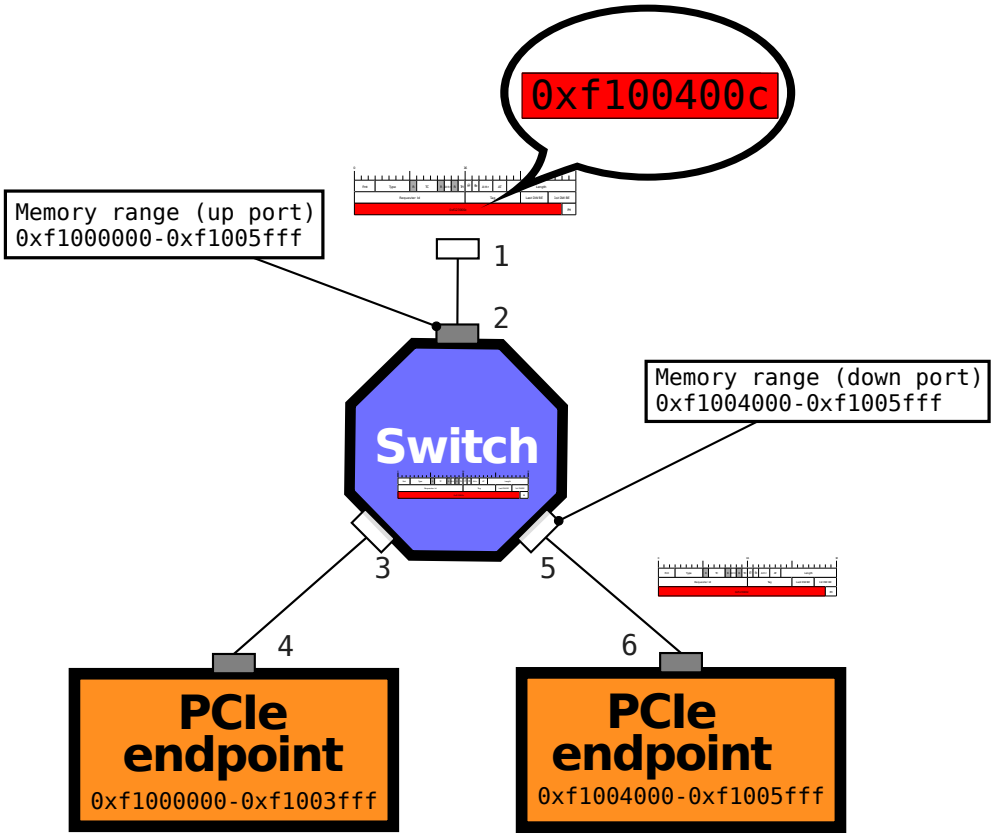

Fig. 5. PCIe address-based routing

source. In this completion packet, the destination field is the requester ID taken from the request packet (`00:0.0` since the request is generated by the root complex), while the *completer ID* matches the endpoint's own identifier (`04:0.0`). Of the three numbers present in this field (Figure 7), only the *bus number* is used for routing.

The completion flows upward in the topology (Figure 8) until reaching the destination or a switch with a downstream port leading to it. Downstream port identification inside a switch uses the requester ID as well as two fields from the port configuration space header: *secondary bus number* and *subordinate bus number*. The secondary bus number is that of the PCIe link directly attached to a port, while the subordinate bus number designates the highest bus number present beneath that port. The resulting interval defines the *bus aperture* of the port. A downstream port is selected when the requester ID from the packet belongs to its bus aperture. When the endpoint is directly attached to the link, the secondary and subordinate bus numbers are identical.

As for address-based routing, upwards completions are forwarded to the upstream link if the final destination is not local and no downstream port contains the requester ID in its bus aperture.

The I/OMMU is a vendor-specific component, absent from the PCIe specifications. It is thus completely optional in a PCIe system and is usually found in virtualization-intended architectures. Most popular I/OMMU implementations include Intel VT-d and AMD-Vi.

An I/OMMU logically sits between the root complex and the memory controller, policing requests from I/O devices targeted at memory ranges in the system RAM. A PCIe fabric can have several of them, each one managing a set of switches and endpoints beneath.

The primary role of an I/OMMU is to add an abstraction layer to the memory space, like its counterpart in the CPU world, the MMU. As is the case for processes in commodity operating systems, using an I/OMMU enables the CPU and each I/O device to have their own memory map. For example, this is useful in case the system has more than 4GB of system RAM and the I/O devices only use 32-bit addresses. All I/O devices target the same range under 4GB but the requests are actually translated by the I/OMMU to different ranges. This is also useful for assigning a device to a specific process (resp. virtual machine), preventing it from accessing memory managed by another process (resp. virtual machine).

The CPU is tasked with configuring the I/OMMU by filling page tables in the system RAM. For each memory request initiated by an endpoint

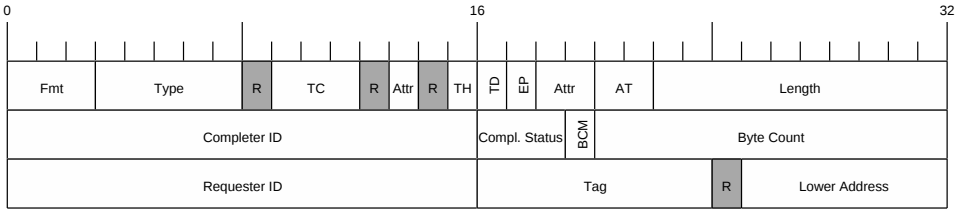

Fig. 6. TLP header (Completion)

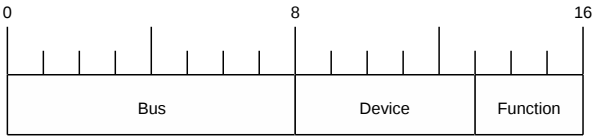

Fig. 7. Device ID

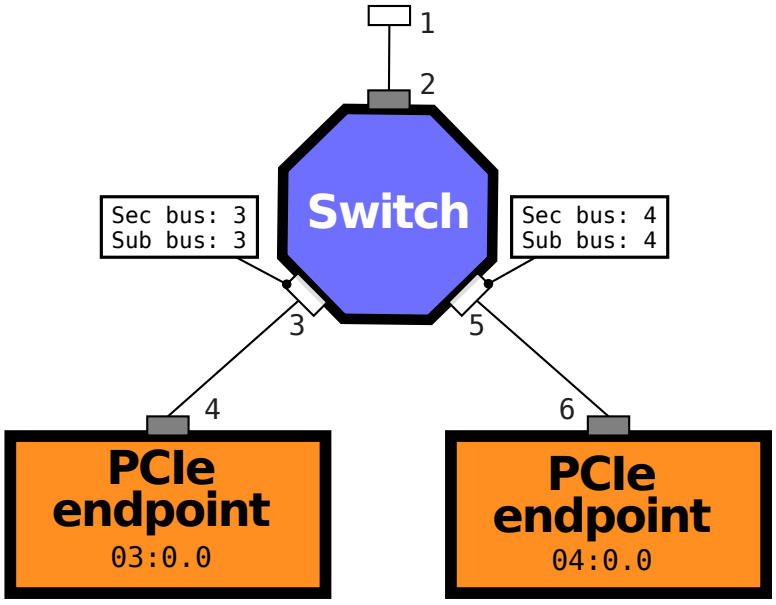

Fig. 8. PCIe ID-based routing

and directed towards the system RAM, the I/OMMU performs a page walk to obtain the final physical address. Page tables are indexed by the device identifier and contain both access rights and the host physical address. Upon reception of a PCIe memory request, the I/OMMU uses the requester ID to find the correct entry and rules on the legitimacy of the request. Authorized requests then entail emission of the corresponding command to the memory controller.

3.2 Modeling Memory Requests in ProTIP

Elements Modeled and Tool Input Our work aims at formalizing commodity architectures while abstracting away all constituents that are not relevant to PCIe peripheral security. To do so, our tool is meant to help users characterize all possible information flows in *their own* fabric instance, so that they can check whether these match their expectations. Even physically identical PCIe networks can bear different configurations which change their properties. As a result, it requires our framework to take into account the actual parameters of a given fabric. The latter constitute the input of our tool.

More specifically, ProTIP works on a set of physical port numbers representing ports of components actually present in a fabric. The tool needs information about the physical links existing between them as well as the contents of their configuration headers that pertain to packet routing. This encompasses BARs for all components, completed with bus numbers, memory base address and limit fields for switches. The user can also specify whether an I/OMMU is present, and which filtering rules it is supposed to enforce. While several I/OMMU components can be used to take charge of different parts of the fabric, we currently represent them as a single element that is present or not. We also consider the CPU to be a single monolithic component.

For the time being, the tool input is encoded manually in an ad-hoc format. We plan on implementing parsers for PCIe configuration as obtained by usual enumeration tools (such as the Linux tool `lspci`[6]). The tool parses input configurations to build an abstract representation of the fabric, which we call a `State` in the rest of the paper.

Details About Rules for PCIe components There is one event predicate for each kind of TLP packets, i.e. one for each type of transaction. The information captured is fully contained in TLP headers. The actual

[6] However, I/OMMU information is not part of what is provided by `lspci`.

payload is abstracted, as we are only interested in whether a TLP can succesfully be routed from one point to another. A read request is formalized by predicate `mem_read(Address, ReqBDF, Tag, AT)`, where `Address` is the targeted address and `ReqBDF` is the requester ID. `Tag` is a tag value used by the requester to associate an inbound completion with a pending request. We skip over the `AT` bit, discussed in section 4. Similarly, completions (resp. write requests) are captured by `completion(ReqID, ComplID, Tag)` (resp. `mem_write(Address, BDF, AT)`).

The variables in these predicates are all encoded as elements of finite domains, whose nice properties have been introduced in section 2.1. This is particularly relevant for addresses, that are represented as elements of an interval, rather than a set of individual values. This is no detail when it comes to enumerating possible traces or listing them as an output. The ability to compute on packets such as `mem_read(Address, bdf(3,0,0), 1, 1)` with `Address in 0xc0000000..0xc000ffff`, by suppressing the need to try all values, renders the tool practical on real-life configurations.

For each TLP type and each kind of PCIe component, ProTIP contains one or more rules describing how the component acts upon reception of the packet. As outlined in section 2.1, PCIe component rules are all built on the same pattern, to capture results entailed by event triggered.

To give the reader an intuition of how it works, we develop the example of a rule capturing what happens when a `Completion` TLP is received by an endpoint device. The specification prescribes that the device accepts a completion under the condition that it corresponds to an outstanding read request that the device placed. This is where tags come into play. Before emitting a read request `mem_read(Address, BDF, Tag, AT)`, a device generates a unique 8-bit `Tag` value, and stores the pair `(Tag,Address)`. If everything goes according to plan, the completion received by the device features its ID in the RequesterID field. However, according to the specification, a completion packet does not reference the address which was read, but bears the tag value. Thus, when a completion is received, the device should compare this tag value to its list of outstanding read requests. If there is a match, then the address value that was read is deduced from the tag list stored by the device.

```
completion_hops(completion(ReqBDF,BDF,Tag), Port, TagList,
                    accept(compl,Address), Port, NTagList, State) :-
      is_endpoint(Port, State, ReqBDF),
      outstanding_request(Port, Tag, TagList, Address, NTagList).
```

Listing 3. Completion rule

Listing 3 shows the sample rule, slightly simplified for the sake of clarity. We note that hop rules actually have seven arguments, contrary to the simplified introduction of section 2.1. However, the intuition provided then still holds. The first six arguments can be divided into two groups, each comprising an event predicate, a physical port and a set of tag lists. The first group represents a trigger event occurence, while the second represents the consequences it entails. The last argument `State` models the configuration of the PCIe network under analysis.

The rule captures that `completion_hops` holds when `is_endpoint` and `outstanding_request` are both true. For example, the predicate `outstanding_request(Port, Tag, TagList, Address, NTagList)` is true when `TagList` lists the pair `(Tag,Address)` under the outstanding request list for `Port`, and `NTagList` is an identical list of tags except that `(Tag,Address)` is removed. It is important to bear in mind that everywhere a variable appears in a given formula, the unification algorithm tries to instantiate the variable with the same term. Thus, it is the appearance of `TagList` in both predicate `outstanding_request` and `completion_hops` which ensures that we check what we are supposed to. Acceptance of the completion is captured by the event predicate `accept`, which references the address matched by the tag value. It is important to note that it is the address which was read, rather than the address to which the completion is copied.

About Non-PCIe Elements The CPU has the particularity to be able to initiate requests, just like the PCIe endpoints are. Let us set aside the I/OMMU problem for a moment. When the root complex receives CPU requests, they are translated into TLPs and are sent downstream through the relevant root ports. We model the root complex as a collection of root ports. The event predicate capturing the reception by a root port of a read (resp. write) request from the CPU is denoted `gen_read(Address)` (resp. `gen_write(Address)`). These predicates appear as the first components of hops modeling root port rules. The CPU itself is denoted as a specific, constant physical port denoted `cpu`. The interaction with system RAM is similarly formalized. Either RAM appears as the constant physical port `ram` when it is the final destination of a request (e.g. when an endpoint writes to system memory), or it is abstracted when a root port receives a read request meant to be completed. In the latter case, relevant root port rules model both reception of a read request and response with a completion.

On top of this, the I/OMMU is captured as a couple of predicates imposing additional conditions to check when testing whether root port rules apply. This is natural for a component architecturally placed between root ports and RAM. We only model access rights applied to different memory regions, but not computation of address translations. These latter computations do not impact our security evaluation per se, only the actual occurence of translation does.

Eventually, we complete the set of rules with non-compliant behaviors that a compromised endpoint can exhibit. We allow these to flood the network with arbitrary packets. To differentiate resulting traces, such rules use event predicate `generate`, rather than `gen_write` or `gen_read`.

3.3 Building ProTIP outputs

A PCIe fabric is the combination of its physical reality and the logical network built on top of it by the transaction layer abstraction. Given a fixed configuration, expressing security properties solely based on the logical view of the network does not allow to properly capture design flaws enabling spoofing attacks. Moreover, the security criterion of interest to users is compartmentalization. It means that ProTIP must provide a description of the actual connectivity of a fabric, both physically and logically. Simply put, users need to know which device can influence which other. Model-wise, this means finding all possible hop sequences ending with a device accepting a packet. Our search algorithm should be sound – meaning that we do not over-approximate possible interactions – and has to be complete – meaning that we do not miss any possible information flow.

A naive enumeration of hop sequences fails to solve our problem: it does not terminate. Indeed, when thinking of a hop sequence, we naturally tend to picture a kind of *minimal trace*. A minimal trace consists in a sequence of hops which starts with a generation event, terminates with an acceptance event and where hops form a causal chain: the trigger event in a hop is a resulting event of the previous one in the sequence. Nevertheless, an arbitrary acceptance-terminated hop sequence exhibits a series of possibly unrelated hops, dealing with transactions that do not necessarily originate in a generation event, and possibly comprises several acceptance events. There is an infinite number of such traces.

To prune our search space while remaining complete, we rather build our search strategy on weakest precondition calculus. Since we are interested in packet acceptance, we first set out to generate all sets of necessary conditions resulting in acceptance, and then check whether these are

satisfiable. We need to decide on a set of tag lists from which to start our backtracking. Acceptance causes *removal* of elements from tag lists, and we are computing backwards. We thus choose to backtrack from empty tag lists for every port to avoid introducing side effects on the computation of preconditions. Our backtracking yields causal chains of hops resulting in acceptance, which we stop at the first generation event occurrence.

Two cases can arise then. Either the tag lists related to this generation event are empty, which corresponds to the initial state for our fabric, or there is one tag list containing a tag-address pair. No other kind of initial set of tag lists occurs in practice. Indeed, only acceptance events augment tag lists in our backtracking process and they never appear as trigger events of a rule. This is not an artefact of modelization but a property of our distributed system. The fact that some traces start on non-empty tag lists is expected. After initialization, the fabric starts with an empty set of tag lists, but other packets can go through it before our trace starts. To check on reachability of our non-empty tag list, we use backtracking again, until we find a generation event which, coupled with the initial set of empty tag lists, results in our intermediate set. This provides the list of all satisfactory trace prefixes.

In the end, the tool outputs these traces in a file so that the user can analyze them. For the time being, since this is the very beginning of our formalization effort, trace analysis is reduced to (not so) pretty-printing. Traces are classified very crudely to check for specification compliance, according to the sequences of events they comprise. In particular, any trace which includes a generate event is labeled as requiring attention.

We emphasize that the security goal of compartmentalization is orthogonal to specification compliance: typically, it is fully compliant to allow every device to read and write all system memory and to communicate with every other device. Eventually, the user is the only one capable to decide whether exhibited traces raise security issues for his own deployment. ProTIP is mainly a very precise PCI configuration audit tool raising warnings. By soundness of our search algorithm, all traces are worth examining: they can actually happen.

3.4 Security Issues Underlined

For the remainder of this article, we repeatedly present results obtained when running the tool on a few variants of the base configuration illustrated in Figure 5 and 8. In addition to information depicted, we precise that the memory range of the root port is `0xc0000000..0xffffffff` in the

base configuration, and that switch downport number 3 (resp. 5) has a BAR of `0xc0000000..0xc0000fff` (resp. `0xc0001000..0xc0001fff`).

Listing 4 shows samples of actual trace listings obtained as a result of running ProTIP with the base configuration as an input, in the absence of I/OMMU. In output logs, each trace is displayed followed by constraints computed by the CLPFD module for its variables[7]. Traces are displayed as sequences of triples formed by event, physical port number and a set of tag lists when the event occurs for the port number. Namely, when two triples are separated with by `->`, it means that a hop rule involving the first and second triple was applied to build the trace.

The first trace shows that endpoint 4 can generate write requests that can successfully reach system RAM adresses comprised between `0x0` and `0xbfffffff`, using an arbitrary requester ID `bdf(B,D,F)`. We remind that the `generate` event predicate is meant to differentiate arbitrary packet generation from specification-conformant behavior. Tag lists are represented as a list of six empty lists, one for each port. Even though this is not modeled at this point, switches can emit read requests and receive completions. The fact that we transit directly from port 1 to RAM has been explained in section 3.2, but the attentive reader may note that port 2 does not appear. This comes from the manner in which switches are modeled: one of their port receives a packet as input, and if it is not dropped or accepted, the outcome is the emission of a packet to another PCIe component. The specification provides little details about what should happen inside switches, rather focusing on what they should output. This is mirrored by modeling switches as wholes.

For the time being, no post-treatment is applied to output traces. There is one log entry per way to build a given hop sequence from the rules. Hence, multiple identical traces or traces refining one another can appear. Moreover, several interval slices correspond to several log entries. The address range in the log must *not* be interpreted as the largest set of addresses that can be written in RAM by endpoint 4 using an arbitrary requester ID. The upper limit comes from the presence of a BAR starting in `0xc0000000` on port 3. Any write request addressing the BAR is accepted by the switch and not routed to RAM.

Diving further in the output logs, we find traces to confirm that endpoints can perform DMA read and write requests to any arbitrary RAM address but those routed elsewhere. The second trace in Listing 4

[7] Real logs differ slightly from those in the listing. Except for `Address`, variables still appear with counter-intuitive, automatically generated variable names, and integers have decimal representation.

provides an example of a legitimate read request from endpoint 4 for RAM addresses. This issue is a well-known security vulnerability in the absence of an I/OMMU.

The third trace illustrates the possibility of generating arbitrary completion packets. The specification does not impose any specific check on completion reception, except that it should match an element in the tag list of the port. This is reflected by the initial set of tag lists of the trace: the tag list of the root port contains one pair. As explained in section 3.3, we must then find prefixes resulting in a such a set of tag lists. The last trace in the log extract of Listing 4 shows one example of such a prefix. The address interval `0xc0000000..0xffffffff` corresponds to the root port memory range. The combination of both traces points out the possibility for a rogue endpoint of racing any legitimate completer. When winning the race, the spoofing device can answer arbitrary data to any read request which the CPU can place, if it correctly guesses the 8-bit tag value.

The achievability and severity of this potential vulnerability have yet to be examined. As mentioned previously, we have not had the opportunity and necessary equipment to inject arbitrary PCIe traffic into a fabric. Since the address being read is not under the control of an adversary, this does not seem easy to exploit except for blunt denial of service attacks. We plan on investigating this further in future work.

```
Trace : port ram accepts write with a WEIRD TRACE
generate,4,[[],[],[],[],[],[]]
 -> mem_write(Address,bdf(B,D,F),1),3,[[],[],[],[],[],[]]
 -> mem_write(Address,bdf(B,D,F),1),1,[[],[],[],[],[],[]]
 -> accept(write,Address),ram,[[],[],[],[],[],[]]
and constraints :
clpfd: (Address in 0.. 0xbfffffff), clpfd: (B in 0..256),
clpfd: (D in 0..32), clpfd: (F in 0..8)

Trace : port 4 accepts completion with a usual read trace :
gen_read(Address),4,[[],[],[],[],[],[]]
 -> mem_read(Address,bdf(3,0,0),Tag,0),3,
                                    [[],[],[],[(Tag,Address)],[],[]]
 -> mem_read(Address,bdf(3,0,0),Tag,0),1,
                                    [[],[],[],[(Tag,Address)],[],[]]
 -> completion(bdf(3,0,0),bdf(0,1,0),Tag),2,
                                    [[],[],[],[(Tag,Address)],[],[]]
 -> completion(bdf(3,0,0),bdf(0,1,0),Tag),4,
                                    [[],[],[],[(Tag,Address)],[],[]]
 -> accept(compl,Address),4,[[],[],[],[],[],[]]
and constraints :
clpfd: (Address in 0..0xbfffffff), clpfd: (Tag in 0..25)

Trace : port cpu accepts completion with a WEIRD TRACE
generate,4,[[ (Tag,Address)], [],[],[],[],[]]
```

```
 -> completion(bdf(0,1,0),bdf(B,D,F),Tag),3,
                                        [[(Tag,Address)],[],[],[],[],[]]
 -> completion(bdf(0,1,0),bdf(B,D,F),Tag),1,
                                        [[(Tag,Address)],[],[],[],[],[]]
 -> accept(compl,Address),cpu,[[],[],[],[],[],[]]
and constraints :
clpfd: (Tag in 0..25), clpfd: (B in 0..256), clpfd: (D in 0..32),
    clpfd: (F in 0..8)

Possible prefix:
gen_read(Address),cpu,[[],[],[],[],[],[]]
 -> mem_read(Address,bdf(0,1,0),Tag, AT),2,
                                        [[(Tag,Address)],[],[],[],[],[]]
and constraints :
clpfd: (Address in 0xc0000000..0xffffffff), clpfd: (Tag in 0..25),
    clpfd: (AT in 0..1)
```

Listing 4. Extracts of ProTIP output on the base configuration

When ProTIP analyzes the base configuration in the presence of an I/OMMU, it rediscovers ID-spoofing attacks exposed in [8,10]. Namely, the first trace in the previous example is not possible for any requester ID anymore: the I/OMMU checks that a transaction is allowed to reach RAM based on its requester ID. That being said, no mechanism in the configuration compels the device to use its own ID. As a result, a rogue endpoint, spoofing an ID listed in the I/OMMU configuration, is granted access to the memory regions initially authorized for the spoofed device.

Last but not least, regardless of the presence of an I/OMMU, peer-to-peer communications between endpoints are possible. In a similar fashion to the CPU example detailed earlier, alternative completions from rogue endpoints can successfully target other endpoints, as soon as the latter emit a read request.

4 Addressing Compartmentalization Issues With Fine-Grained Configuration of Routing Components

4.1 Introduction to Access Control Services and Address Translation Services

One interesting feature of PCIe is the ability for endpoints to exchange data with the system memory without involving the CPU: since those transfers are slow, stalling the CPU during them would be costly. Direct memory access (DMA) is needed for improving performance for high speed devices like hard drive controllers and network interface adapters. But as is now widely known, DMA raises security concerns, which the I/OMMU is meant to address. It translates *device addresses* into *physical addresses* by

performing a page walk on each memory transaction. While this is faster than having the CPU do the transfer, it is still too slow for some usages. I/OMMU implementations commonly include some internal memory to cache translated addresses (like a TLB for the MMU). Nevertheless, it remains a bottleneck, specifically when multiple devices need to access memory simultaneously. The PCIe specification has been extended with *Address Translation Services* (ATS) to limit performance impact.

ATS delegate the caching part of the translation to the endpoints themselves in order to distribute the load. When emitting memory requests, instead of using device addresses (the only address type previously known by an endpoint), an endpoint can now use an already translated physical address. When using pretranslated addresses in a TLP, an emitter must set the *Address Type (AT)* bit.

A device can obtain translated addresses using a dedicated transaction, consisting of a *Translation Request* and its associated *Translation Completion*. The requester stores the translation result in a local cache (also called *I/OTLB*). The translated address can then be used to compose a TLP later on. When the AT bit is set, the I/OMMU translation is bypassed.

The specification explicitely forbids a device to issue a memory request with the AT bit set if the address in the destination field is not obtained as a result of the translation protocol. Yet, a compromised endpoint does not necessarily conform to specifications. Any request received by the root complex with the AT bit set bypasses the I/OMMU. Thus, a rogue component whose requests are routed to the root complex is basically granted unrestricted access to the system RAM.

ATS are coupled with *Access Control Services* (ACS), allowing a host to inhibit the use of translated addresses to some extent. Included in the PCIe specification, ACS allow filtering of TLPs in the routing components of the PCIe topology. In particular, they apply to root ports and switch downstream ports, grouped under the terminology *downports*. By specification, implementing ACS is optional. ACS support by routing component of a fabric is not required to be homogeneous. For each TLP handled by a component with ACS support, the decision to route it normally, block it or redirect it can be made based on rules called *access controls*. There are seven access controls and each one can be enabled separately on any downport. In this article, we focus on five access control services. These are *Source Validation*, *Translation Blocking*, as well as controls related to peer-to-peer transactions: *P2P Request Redirect*, *P2P*

Completion Redirect, and *Upstream forwarding*. *P2P Egress Control* and *Direct Translated P2P* have not been formalized yet, so we do not detail their effect. ACS violations are supposed to be reported to both the root port and the offending device.

Source validation forces a downstream port to check if the requester ID in a request belongs to its bus aperture. When *translation blocking* in enabled in a downport, the latter drops any request with the AT bit set. *P2P request* and *completion redirects* are used to force validation of peer-to-peer transactions by a component in the root complex implementing *Redirected Request Validation Logic* (RRVL). When both P2P request redirects and P2P completion redirects are enabled in a switch downstream port, a packet that would normally be routed to another downstream port of the same switch (a peer-to-peer TLP) is instead forwarded to the upstream port towards the root complex for validation. Such a packet is transmitted unmodified on the upstream link. The parent device then receives a packet by its downport, packet that it would normally route downstream through the very same downport. Different behaviors can occur. In case the TLP is dropped, the P2P transaction is lost. If the TLP is reemitted downstream by the parent downport, it skips validation. *Upstream forwarding* is intended to fix that problem. When enabled, it forces a switch to forward upward this new kind of packets, received on a given downstream port which claims them. For P2P redirects to actually enforce access control checks, it has to be coherently configured along the paths of packets. However, nothing in the specification imposes restrictions on acceptable ACS configurations in a fabric. It is up to the BIOS or the operating system to configure access control in a suitable manner.

4.2 Extending ProTIP with ACS and ATS

While still in progress, ACS modelisation has already produced results. The ATS protocol to query the I/OMMU is not modeled. We do not plan on doing it in the initial version of the tool, because requesting translated addresses is a completely independent process from the device ability to leverage the AT bit in memory requests. Obviously, the configuration of the fabric used by ProTIP must be extended to take into account ACS settings on all relevant physical ports. Setting all ACS configuration bits to 0 disregards these services on components of the fabric which do not support them.

Then, hop rules are completed with ACS predicates modeling conditions which should be checked according to the specification. To model the

proper handling by the root complex of requests ensuing from redirection of peer-to-peer requests, we need the specification of the Redirect Request Validation Logic, to which the PCIe specification delegates the access control decision. We have yet to find a proper specification for this root complex component, but we expect this to be vendor specific. For the time being, it is modeled like the I/OMMU: permissions are indexed by requester IDs. There are also grey areas around the criteria used to decide whether a request or a completion received is meant to be routed through the RRVL or upwards to the CPU or memory controller. A priori, knowing both the memory range in the root port configuration and the requester ID is sufficient to choose the proper outcome. We have yet to dig further in the specifications to find eventual clarifications. It is the prerogative of formalization efforts to shed light on specification imprecisions.

Eventually, as far as trace search is concerned, the extensions introduced for ACS do not require any change to our algorithm.

4.3 New Security Flaws

We have run ProTIP on our base configuration augmented with various ACS settings.

Firstly, it correctly reports that when no Translation Block bit is set on any downport receiving a request with the AT bit set, this request successfully reaches its recipient. In particular, it allows a rogue endpoint to write to all system RAM addresses not claimed by any other device. This was clearly identified by Fernand Lone Sang, and exposed in his PhD thesis [7].

Secondly, it shows that when the source validation bit is set on the downstream port of a switch directly facing an endpoint, ID-spoofing requests are blocked. However, this only holds on two conditions. First, source validation must be activated on the first downport upstream of an endpoint. Second, the downport has to be configured so that its bus aperture only comprises the endpoint's bus number. We remind that source validation merely imposes that the requester ID belongs to the bus aperture of a downport. Thus, to be successfully routed, an endpoint request only needs to have a requester ID belonging to the bus apertures of downports with activated source validation.

This highlights a property of configurations classically generated, at least in Linux systems. Memory ranges and bus apertures of downports are set so that they do not include values which are not truly claimed by downstream endpoints. We say that such configurations are *tight*. They respect the principle of least privilege.

Eventually, as far as complete compartmentalization of every endpoint is concerned, we could expect the existence of a safe way to position ACS bits on tight configurations. By safe, we mean that all possible flows between devices, be it peers or non-PCIe components, would go through the I/OMMU or the RRVL to check their conformance to some comparmentalization policy. However, this is not the case. We have run ProTIP on our base configuration, augmented with source validation, request and completion redirection, and translation blocking activated on all downports in the fabric. It exhibits the same illegitimate completions opportunities as those detailed in section 3.4. Completion races can occur on a peer-to-peer level as well as from an endpoint to the CPU. This is not surprising, since the only access control service acting on completions is P2P completion redirects, which exists for scheduling purposes only. The specification states that "the intent of ACS P2P Completion Redirect is to avoid ordering rule violations."

5 Extending the Model With Configuration Requests

5.1 Details on Configuration Request Issues

As mentioned in section 3.1, PCIe devices expose internal parameters as a collection of registers called the *configuration space header*. For example, this configuration space contains the routing information of ports. It is not immutable, but rather dynamically set by the host system. The configuration data is sent to the device and stored in the internal registers using *configuration write requests*. Data can also be read using *configuration read requests* and associated completions.

The PCIe configuration space contains the configuration space headers of all devices. It is I/O-mapped at a base address provided by the system firmware (on x86 architectures by the `MCFG` ACPI table)[8]. When the CPU writes to an address belonging to that specific range, the root complex receives it instead of the system RAM. The root complex then generates a configuration request directed to the device associated with the offset in the PCIe range. These requests are routed by ID, with the BDF identifier encoded in the memory address, as displayed in Figure 9.

A device does not store its BDF in its configuration space header, but rather in a private memory area. Nevertheless, it is influenced by configuration write requests. Each time it accepts a configuration write request, a device must update its device and bus number to reflect the

[8] The legacy PCI access method using I/O ports `0xCF8` and `0xCFC` is omitted here.

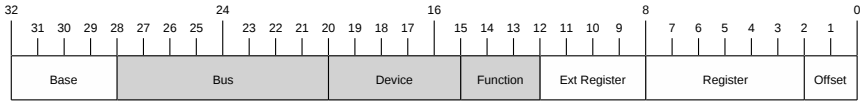

Fig. 9. BDF encoding in a memory-mapped PCIe configuration space access

target ID appearing in the request. Informally, the device deduces its new ID from the last one used to send it configuration information. More precisely, upon reception of a configuration write request, a PCIe device reacts according to the type of the request. The specification states that a *type 0* request is always accepted by the device receiving it, while a *type 1* should be routed further. Therefore, every time a device receives a *type 0* configuration request, the request contains its – potentially new – BDF.

The CPU has no control over the configuration request type generated by the root complex. Unless targeting a directly attached endpoint, the root complex always emits a type 1 configuration request. To route such a request, a downport compares the target ID of the request with its own secondary bus number. If they match, the downport converts the type 1 request to a type 0 and emits it downstream, and the device immediatly below accepts it. Otherwise, the downport forwards the type 1 request downstream without changing it.

Changing an endpoint source identifier is thus a two step operation. Let us provide a concrete example. We use the base configuration depicted in Figure 8, and describe actions required to change the identifier of port 6 from `04:00.0` to `05:00.0`.

First, code running on the CPU updates the configuration of port 5, to set its secondary and subordinate bus number registers to `5`. This is done by accessing the port configuration space header, mapped in the CPU memory. The addresses where the CPU must write are obtained by combining the PCIe base address, the identifier of port 5 and the offset of the registers.

Then, the CPU writes to any register in the endpoint configuration space header using the *new* BDF identifier (`05:00.0`) to build the destination address. This new identifier becomes the target ID of the type 1 request created by the root complex. The request is converted into a type 0 request at port 5, since its secondary bus number has just been updated with `5`. The endpoint accepts the type 0 configuration request and updates its internal configuration. Packets emitted later on by the endpoint use `05:00.0` as a requester ID, if the endpoint conforms to specification.

5.2 Towards the evaluation of compartmentalization when delegating peripherals

The ability to issue configuration requests after the system has booted is needed to ensure that devices can be hotplugged or pulled at any time during system execution. Drivers can also use access to some specific registers – not modeled in ProTIP – to fine-tune the behavior of a device. Nevertheless, changing the value in the configuration registers modeled in our tool can totally change the logical topology of the fabric and severely impact the compartmentalization properties actually enforced in the system. Thus, from a security point of view, it seems quite appropriate to try and evaluate the consequences of configuration requests.

That being said, it is a fact that the legitimate system (say, the host kernel running on our machine) can reconfigure everything, and this is not what really raises problems. The idea is rather to assess the impacts of requests that an attacker could use to compromise expected compartmentalization properties, in a deprivileged setting such as peripheral delegation. Existing PCI-passthrough implementations generally tackle this problem by applying filtering operations to configuration space accesses. It would be useful to be able to assess these rules and warn users about potential security gaps. This amounts to reducing the set of configuration requests of interest to those that an attacker can issue.

Therefore, we would like to have a tool listing all traces in a given fabric ensuing from adding to the usual traffic a given set of configuration requests from the CPU. Unfortunately, ProTIP only answers part of this challenge at the time being. The problem that we have to address is that our search algorithm is not satisfactory anymore: `State` can change. However, the CLPFD module enables ProTIP to work with configurations exhibiting variables in place of instantiated integer values in configuration registers. What we set out to do is thus to compute a set of reachable configuration states, represented by a state comprising variables in finite domains where the configuration can be updated. Then, this variable configuration is provided as an input to our previous search algorithm. This global search algorithm is *not complete*: configuration changes are taken into account for traces transitioning at most once by each physical port. In other words, we emphasize that we *do not* enumerate all traces formed by interleaving configuration requests and memory requests. Traces presenting packets going at least twice through ports whose configuration has been updated are not taken into account by our algorithm. However, this already allows us to uncover potential threats.

The variable state computation is performed by an extension of ProTIP, which comprises configuration rules. It uses the input configuration given by the user, to iteratively compute possible consequences of configuration requests, transforming registers into variables at each step. Since side effects from one component to another only exist between parent and children, a fix point is always reached. Our implementation is a proof-of-concept in need of more testing and stabilization.

5.3 Preventing New Security Issues

The beginning of our study on illegitimate configuration updates has highlighted potential security issues in case of delegation of a routing device. Judging from existing open-source implementations studied below, we honestly do not think that this has been used in production scenarios to this day. With the appearance of peripheral delegation in cloud settings, we prefer to preventively raise awareness about potential consequences of delegation of switches.

When a downport configuration can be updated, at least two ways of circumventing compartmentalization exist. The first is *configuration untightening*. As was explained in section 4.3, the unnecessary inclusion of elements in memory ranges or bus apertures of switches can allow unwanted transactions to be routed. A good example is the modification by an attacker of a subordinate bus number. It can entail the routing of ID-spoofing request of a child device, even if the ACS Source Validation bit is set and a filtering policy prevents its modification. The second problem lies in *legitimate ID-spoofing*. As explained in section 5.1, if the attacker is able to provoke the issuance of two requests allowing to update in a legitimate manner the ID of a device, then this latter assumes this ID. This endangers compartmentalization when the ID belongs to another device whose information flow can reach the updated one – which depends on the topology of the fabric.

We have examined several implementations of PCI-passthrough: VFIO for Linux-KVM, Xen-Linux and Xen-NetBSD. All of them do let access to some part of a configuration register (typically the cache line size register, although it does not do anything and is just a PCI artefact). Currently, VFIO does not allow for routing device delegation – though comments in the code can lead to believe that support is planned in the future. Besides, VFIO carefully filters writing operations accessing configuration space. By default, Xen does not enable routing device delegation. With default settings, it emulates a PCIe root complex for hosts to which passthroughed device appear attached. In another available mode (by changing an

argument at the relevant driver start), Xen allows to delegate routing devices. In default configuration, filtering is implemented on register access. Setting Xen in permissive mode allows for unfiltered access. Permissive mode is a threat to security, basically every configuration register is exposed. NetBSD seems to do something similar to the permissive mode in Xen.

The feasibility of breaking compartmentalization using legitimate ID-spoofing has not been carefully examined yet in these settings. ID-spoofing has been carried out successfully as root user on Debian Jessie with an activated I/OMMU, and has resulted in the possibility for a Slotscreamer device with no system RAM access authorized by the I/OMMU to read and write in the memory reserved to the NIC. This confirms feasibility in principle, but not in our attacker model. In case there is a Virtual Machine (VM) to which both a routing device and one of its child devices are delegated, we still need the hypervisor to let the VM write in at least one register of the child *at the address of its targeted ID*. We anticipate that it should not be possible to perform this last step. Indeed, it seems that the address actually transmitted to the root complex is computed by the backend driver, from the bus, device and function numbers stored at delegation time. If all this is true, then the preservation of this property in future implementations ensures the attack remains impractical.

6 Conclusion

In this article, we have presented our current formalization effort to evaluate the compartmentalization of devices in a PCIe fabric in a systematic manner. We have highlighted already known defaults of the communication protocol, along with some new technical details to be explored further in real-life settings. To do so, we need to couple our approach with the use of an IronHide-like fuzzer to try out our scenarios. Such devices can also find bugs in real components bugs, which can then be modeled in ProTIP to evaluate their consequences. Both approaches complement each other. Besides limitations of the current implementation, our work suffers from classic limits of formalization efforts, such as the difficulty to verify the adequation between model and reality. In our case, the closed world hypothesis of Prolog makes it difficult to validate the implementation. If a trace that we expect is missing from the logs, we can notice it, but if we are not expecting it, we risk missing attacks. In the future, we wish to continue down this path, to assess the problems already uncovered, perfect the tool search strategies, and refine the model with new features.

Acknowledgements The authors thank Arnaud Fontaine for helpful technical discussions on the formal model, and Pierre Capillon and Mickaël Salaün for lengthy discussions about this work and its perspectives.

References

1. R. Amin. Demystifying the i-Device NVMe NAND. `https://ramtin-amin.fr/#nvmepcie`, 2016.
2. R. Budruk, D. Anderson, and T. Shanley. *PCI Express System Architecture*. Mindshare PC System Architecture. Addison-Wesley, 2004.
3. J. Fitzpatrick and M. Crabill. Slotscreamer. `https://github.com/NSAPlayset/SLOTSCREAMER`, 2016.
4. U. Frisk. DEFCON, Direct Memory Attack the KERNEL. `https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Ulf-Frisk-Direct-Memory-Attack-the-Kernel.pdf`, 2016.
5. U. Frisk. PCILeech. `https://github.com/ufrisk/pcileech`, 2016.
6. J. Hansbrough. Fuzzing PCI Express: security in plaintext. `https://cloudplatform.googleblog.com/2017/02/fuzzing-PCI-Express-security-in-plaintext.html`.
7. F. Lone Sang. *Protection des systèmes informatiques contre les attaques par entrées-sorties.* PhD thesis.
8. F. Lone Sang, É. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an I/OMMU vulnerability. In *5th International Conference on Malicious and Unwanted Software, MALWARE 2010, Nancy, France, October 19-20, 2010*, pages 7–14, 2010.
9. F. Lone Sang, V. Nicomette, and Y. Deswarte. IronHide : plate-forme d'attaques par entrées-sorties.
10. F. Lone Sang, V. Nicomette, Y. Deswarte, and L. Duflot. Attaques DMA peer-to-peer et contre-mesures. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*, pages 150–179, 2011.
11. C. Maartmann-Moe. Inception. `https://github.com/carmaa/inception`, 2011.
12. K. Nohl, S. Krissler, and J. Lell. BlackHat, BadUSB – On Accessories that Turn Evil. `https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf`, 2014.
13. Y.-A. Perez, L. Duflot, O. Levillain, and G. Valadon. Quelques éléments en matière de sécurité des cartes réseau. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*, pages 213–234, 2010.
14. R. Sevinsky. BlackHat, Funderbolt: Adventures in Thunderbolt DMA Attacks. `https://media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf`, 2013.
15. SWI-Prolog. SWI-Prolog Reference Manual. `http://www.swi-prolog.org/pldoc/doc_for?object=manual`.
16. The Qubes OS Project. Qubes-OS. `https://www.qubes-os.org/`.
17. A. Williamson et al. Vfio. `https://www.kernel.org/doc/Documentation/vfio.txt`.