On the Proof-Oriented Design of a Context-Switching Service in the Pip Protokernel

Florian Vanhems, Narjes Jomaa, Samuel Hym, David Nowak
CRIStAL, CNRS & Lille University, France

Abstract—**The Pip protokernel is a kernel whose trusted computing base is reduced to its bare bones. The goal of such minimisation is twofold: reduce the attack surface and reduce the cost of the formal proof of security. In particular, multiplexing is not implemented in the kernel but in a partition whose code is executed in user mode. This of course assumes that the kernel provides minimal services dedicated to signal sending. In this paper, we describe a streamlined service designed to allow for inter-partition communication through userland structures that mimic the traditional Interrupt Descriptor Table.**

1. Introduction

Control flow transfer mechanisms are vital components of a system. Transfers occur thousands of times a second in our modern computers. Providing a robust – and efficient – control flow transfer mechanism is therefore paramount in most systems. Yet, numerous software attacks targeting the execution flow exist, notably through memory corruption (use-after-free, buffer overflow, heap overflow, etc.). Formally proving that no illegal accesses are made to the system's memory is a strong protection against those attacks. Pip is a one-of-a-kind kernel that provides formally proven memory isolation. In this paper, we report on the design of Pip's control flow transfer service.

Related work seL4 [7] is the world's first formally verified general usage microkernel. It provides virtual memory management, thread scheduling, inter-process communication and access control. Whereas in the case of the Pip protokernel those last two features are to be implemented in userland. CertiKOS [5] provides a certified programming method to formally develop kernels in Coq, targeting an extension of CompCert Clight and assembly code. It has been used to prove noninterference results [1]. Prosper [2] is a separation kernel that supports dataflow functionalities of the ARMv7 processor, including inter-partition communication, scheduling. Security properties have been proved in HOL4. Bisimulation is used to carry higher-level proofs, based on an abstract model including communication channels, over to a concrete one based on an intermediate language that represents the ARM instruction set [3]. The security property about the communication between isolated entities is the goal of formal verification. This information flow property [4] is verified at the binary code level using HOL4. In [8], the theoretical verification framework XCAP is used to formally verify a realistic x86 implementation of machine context management.

Outline This paper is structured as follows. First, in Section 2, we give an overview of the Pip protokernel and its security properties. Then, in Section 3, we describe how we designed the context switching service and provide some insight on how this service is meant to be used. We focus in Section 4 on the formal proof of security and the impact of our context switching mechanism on it. Finally, we illustrate in Section 5 the usability of this context-switching service.

2. About the Pip protokernel

2.1. Pip's philosophy

Pip is a minimal operating system kernel where the minimisation of its size is motivated by the reduction of both the cost of proof and the attack surface, while allowing for efficiency and usability. The code of Pip is written in Gallina (the language of the Coq proof assistant). More precisely, we restrict ourselves to a monadic code that allows for a word-for-word translation into C code.

The primary goal of Pip is to provide memory isolation to applications that run inside memory partitions (that we simply call *partitions*). At boot, Pip creates a single partition congregating the system's whole memory, called the root partition.
At any time, a partition can decide to create another partition. We refer to the newly created partition as a *child* of the first partition, while we refer to the first partition as the *parent* of the new partition.
In order to create a child partition, a parent has to share a part of its own memory, and provide it to its child. The memory shared this way is accessible to both the child and its parent, except for a tiny amount requisitioned by Pip to maintain its internal structures. The memory requisitioned by Pip is *no longer accessible* to any of the partitions.

These parent-child relation between the partitions create a partition hierarchy called *partition tree*. Fig. 1 shows a partition tree where P_{root} is the root partition, that has P_1 and P_2 as children partitions. In a partition tree, we say that two partitions are *incomparable* if none of them is an ancestor of the other. For example in Fig. 1 P_2 and $P_{1.1}$ are incomparable.

Note that only Pip's code is allowed to run in kernelland. Any other piece of code (such as the code of a partition) runs in userland, with no privilege.

2.2. Pip's security properties

Significant works were carried out to formalise security properties that should be ensured by operating

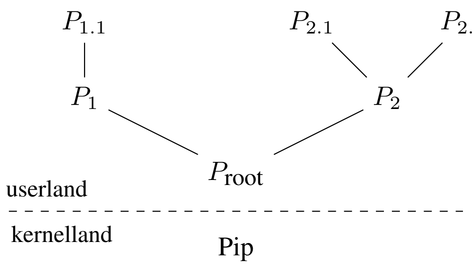

Figure 1. A partition tree example

systems [9], [10]. This includes the isolation between applications as well as controlling the communications between them. Therefore, a first step in this verification process is to require a concise definition of the security properties and to identify hardware and software components on which these properties are founded. In this context, we are interested in the memory isolation property. This property ensures that an application cannot illegally access the memory of another application, nor the kernel's. We consider it the most fundamental security property, because it allows to prove several other security properties such as ensuring the security of communication between processes. We decomposed the isolation property into four lesser properties described below.

Kernel isolation This property mainly defines the isolation of the kernel's data and code from unauthorised access that may occur from userland. In our case, this property is based on memory partitioning. Memory configuration is performed by the kernel and user accesses are controlled by the hardware component MMU (Memory Management Unit).

Horizontal isolation This property states isolation between incomparable partitions (see definition and example at Sec. 2.1). It guarantees that the configuration of the address space of a partition does not allow it to access the physical memory of another incomparable partition. Like the previous one, this property relies on the MMU to control user accesses.

Vertical sharing This property ensures that all pages of a partition are a subset of its parent pages and (recursively) of its ancestors in the partition tree. It is important to manage the hierarchical model of the TCB where the security of a partition relies on the security of its ancestors.

Internal consistency This property states that informations stored in the kernel structures are consistent with the current state of the memory. For example, these structures include the configuration of the MMU for each partition.

3. Designing Pip's control flow service

We will further refer to the context switching service as `switch`. Pip's philosophy is to reduce the TCB to a strict minimum. Pushing the microkernel, then exokernel philosophy further, Pip throws multiplexing outside of the kernel. This allows for a lighter proof cost as well as a tighter attack surface [6].

As such, Pip's sole concern about hardware interrupts is to forward them to the root partition, which will assume the multiplexer's role and process them as it sees fit. In case of a software interrupt, Pip will handle it only if it is a call to one of its own services. It will otherwise forward it to the parent of the caller and pass the execution flow.

It becomes obvious that Pip needs to provide a way for multiplexers - and generally every partition - to send signals to each other, and that partitions should be able to process these signals their own way. This is semantically close to software interrupts, but each partition should be able to configure its interrupt handlers independently from one another. It follows that a per-partition structure is required to hold the configuration. We decided that `switch` would be built on this observation.

3.1. VIDT and contexts

`switch` uses a structure called "Virtual Interrupt Descriptor Table" (VIDT). This structure lives in userland, in the partition's virtual address space, to allow partitions to manipulate it at will. The VIDT holds pointers to CPU states (mainly snapshots of the CPU registers) that we call *contexts*. These contexts are stored in the partition's own memory. Each interrupt number is associated to a context as seen on Fig. 2.

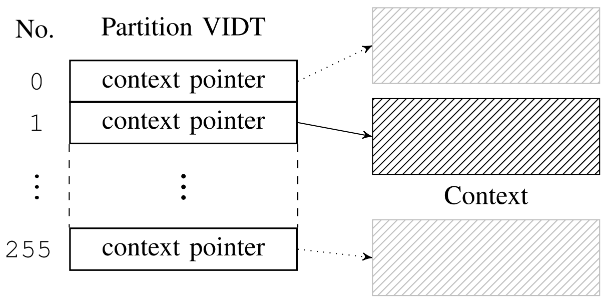

Figure 2. A partition VIDT and its pointers to contexts

In order to pass the execution flow from a partition A to another partition B, `switch` first creates a context out of partition A's CPU state, and stores it inside partition A's virtual IDT. Next, it switches to the memory space of partition B by modifying the page table root register (CR3) which is stored inside Pip's structures. Last, it restores a context from partition B's VIDT, and lets partition B resume its execution.

Pointers and contexts living in userland implies that some partitions may try to alter them. Even though, partitions can not use this ability to threaten Pip's security measures, if Pip carefully checks the data it uses from the partitions. Those checks include verifying that some pages are accessible to the memory partition, that the pointers Pip dereferences actually points inside the partition, etc. On another side, we do not need to check the "validity" of the contexts we restore, as they would run in user mode. Any illegal access to the memory would be prevented by the MMU, generating a fault.

However, we did find a way to circumvent Pip isolation properties if those checks were not thoroughly made; we will further discuss this in Sec. 4.

3.2. Calling `switch`

In this section, we review the different arguments required to call `switch`, in order to highlight parts of its behaviour.

The caller must first indicate the partition it wants to call. In order to do so, Pip provides a structure called *partition descriptor*, that lives in kernelland. The partition descriptor's address is returned by Pip at creation time, allowing the parent to further refer to its newly created child. The caller hence uses the partition's descriptor virtual address (in its virtual address space) to refer to its callee. This is fine if the callee is any of its child, as their accessible memory resides in the caller's addressing space by design (although the memory requisitioned by Pip is not accessible anymore). On the contrary, a caller has no way to refer to any of its ancestors through an address since its ancestor's partition descriptors are stored outside of its virtual memory (it would otherwise break Pip's recursive tree structure, thus breaking invariants required by the isolation property). That is why we set a particular address (an address that cannot be used as a partition descriptor for a child) to allow a partition to send an interrupt to its parent. We call that particular address the *default address*.

Second, the caller partition must provide the interrupt number it wants to send to the callee. Being able to choose from different interrupt numbers allows a partition to have multiple entrypoints, allowing it to declare specific handlers for specific situations.

Third, it must declare an interrupt number to associate its current context with. That context will be restored whenever said associated service is called. The partition only has to provide an interrupt number. Pip will then read the address located at the corresponding index in the VIDT, and then save the context at this address, as seen on Fig. 3.

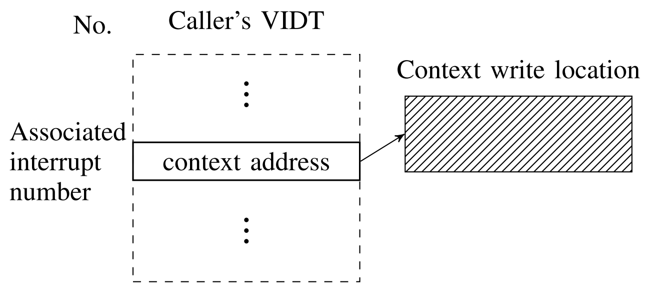

Figure 3. Pip saving a context in the caller's memory

If the address is invalid (for example if some part of the context would be written out of the partition's accessible memory), Pip will cancel the call and throw an error to the caller. If the partition wants its context to be dropped, it can provide an unused interrupt number, where the default address is written. In that case Pip will skip saving the partition's context and will not throw an error.

Fourth, Pip provides a way for partitions to ignore specific interrupts. This allows partitions that do not implement some interrupt handlers to ignore the associated interrupts. It can also be used by partitions to ensure they are not called on specific handlers when being in an unstable state. There are two sets of flags to provide when calling Pip. The first ones are applied to the caller if the execution flow transfer is to be successful. These flags are applied until Pip restores one of the caller's contexts. The second ones are only useful if the caller chose to save its context. In that case, these flags will be inserted in its current context, and restored whenever the context is loaded again. These flags are applied atomically, which means they are effective as soon as Pip did transfer the execution flow. This is akin to the hardware features an OS kernel uses when it needs to ensure atomicity of sequences of instructions (by blocking interrupts for some time).

4. Impact on Pip's isolation properties

Pip's fundamental security properties. The isolation proof of `switch` remains to be written. We can only give some insight, but we believe there will be no major obstacle to the isolation's proof. Indeed, this service does not deal with Pip's internal structures, as opposed to Pip's other services. We briefly explained in Section 2.2 what are Pip's isolation properties, and are going to provide arguments on why we believe this service does not break these properties.

While `switch` needs to access kernel structures to translate the partition's addresses, it does not tinker with them in any way. The service sometimes has to write in memory, but it only does so inside its caller's accessible memory. Moreover, the kernel isolation property assures that – before the service execution – the partition's accessible memory is disjoint to Pip's memory. That implies that structures tracking memory partitioning remain untouched during the service's execution, therefore making the whole set of properties trivial to prove (once we proved that Pip only writes in userland).

Extending Pip's isolation properties. This section will focus on security aspects that are not covered by Pip's fundamental security properties. As pointed out in Section 3.1, without the proper checks, a malicious partition's code could use Pip to read into protected memory.

By making a child's handler context point into kernel pages, and calling that handler, the malicious partition would make Pip load part of its kernel pages as if it were a context. This would most certainly make the child's partition fault, and Pip would dump the faulting context to the child's VIDT (this dumping mechanism is provided by Pip, and is required to let parents handle expected faults - such as stack overflows - from their children). This would allow the malicious parent to retrieve the faulting context, thus reading parts of the kernel's memory. This particular setup is pictured on Fig. 4.

However, this does not break any of Pip's fundamental properties, as this particular attack does not involve configuration pages. It has been shown that those fundamental properties imply some basic information flow properties [6]. We do not want them to be too rigid because each partition should be able to enforce the information flow policy for itself and its descendants. But still, this particular example suggests that we should prove a new security property stating that each address a service dereferences from a partition's accessible memory needs to point to that same partition's accessible memory (beside the eventual default values for which Pip acts differently). This property did not appear when dealing with the other

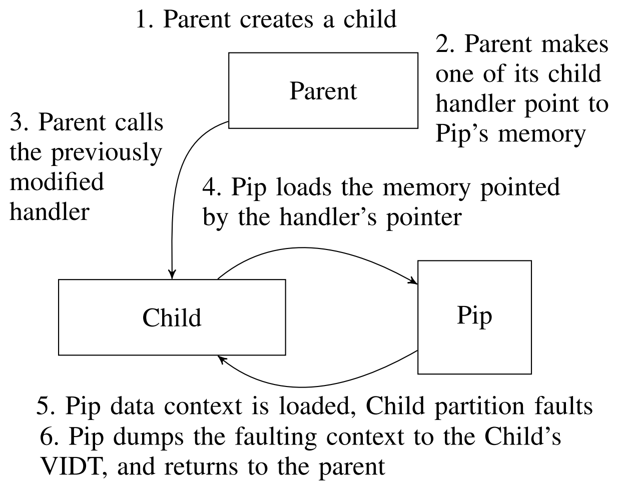
1. Parent creates a child
2. Parent makes one of its child handler point to Pip's memory
Parent
3. Parent calls the previously modified handler
4. Pip loads the memory pointed by the handler's pointer
Child
Pip
5. Pip data context is loaded, Child partition faults
6. Pip dumps the faulting context to the Child's VIDT, and returns to the parent

Figure 4. A partition reading Pip's supposedly inaccessible data

Pip's system calls because they only deal with memory partitioning, and not with the control flow like `switch`.

5. Application to the real world

This section will illustrate how this service can be used with a real world example. We will illustrate how a Linux partition can use Pip to notify a server process that it should close.

Let's assume we have a Linux partition somewhere in Pip's partition tree, and that Linux has spawned processes which are inside their own partitions, including the aforementioned server. Linux uses Pip's VIDT structure to register its system calls, and most notably the `exit()` system call. The server is configured to handle `SIGINT` signals.

The setup is pictured on Fig. 5.

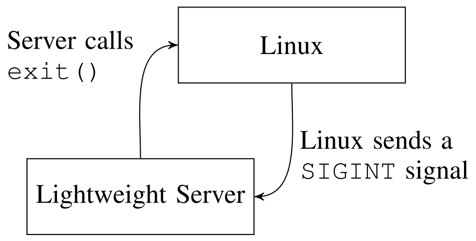

Figure 5. Linux sending a signal to a lightweight server

The model allows for whatever semantic you'd like behind each interrupt, so let's arbitrarily pretend Linux chose interrupt `0xA3` to signal a `SIGINT` and interrupt `0xB7` to handle an `exit()` syscall.

Linux calls Pip with the server's partition descriptor and interrupt number `0xA3` (in this example, flag arguments have no influence whatsoever, they'll be omitted). Pip performs the checks described in Section 3.1, and loads the server's context associated with the `0xA3` interrupt.

Note that the server may have disabled this interrupt through the flags, for whatever reason. Let's assume this is not the case, that the interrupt is not masked and a proper context has been set up to handle the situation correctly.

The server's handler does its housekeeping, closing incoming connections and preparing it for deletion. Once it's done, the handler calls Linux's system call `exit()` to notify its termination. It provides Pip with the default address (to notify its parent), interrupt number `0xB7` (here flags don't matter neither). Once again we suppose that Linux was set up to let this happen.

Linux's context associated with interrupt `0xB7` is hence loaded. Linux then proceeds with the deletion of the process' partition and collects its memory.

6. Conclusions

In this paper we have described a kernel service designed to allow memory isolated processes to communicate through signals. It follows Pip's philosophy as it only provides for the code that must run in kernelland. Anything else, such as a multiplexer, is implemented efficiently in userland by relying on Pip services. We have argued that Pip's fundamental security properties are preserved by the discussed service and that it can be (and will be) formally proved in Coq without any obstacle. We have also noticed that the specificity of system call require an additional information-flow security property.

References

[1] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 648–664. ACM, 2016.

[2] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 223–234. ACM, 2013.

[3] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the armv7 instruction set architecture. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

[5] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, pages 653–669, 2016.

[6] N. Jomaa, P. Torrini, D. Nowak, G. Grimaud, and S. Hym. Proof-oriented design of a separation kernel with minimal trus ted computing base. In *18th International Workshop on Automated Verification of Critical Systems, Oxford, United Kingdom*, Electronic Communications of the EASST Open Access Journal, 2018.

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[8] Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In *Theorem Proving in Higher Order Logics*, pages 189–206, 2007.

[9] J. Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 12–21, 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[10] J. Rushby. A trusted computing base for embedded systems. In *Proceedings 7th DoD/NBS Computer Security Initiative Conference*, pages 294–311, 1984.