# Execution Platform Contracts

Dorian Bourgeoisat, Ulrich Kühne, Florian Brandner

# Execution Platform Contracts

Dorian Bourgeoisat
*LTCI*
*Télécom Paris*
*Institut Polytechnique de Paris*
Palaiseau, France
0000-0002-2121-939X

Ulrich Kühne
*LTCI*
*Télécom Paris*
*Institut Polytechnique de Paris*
Palaiseau, France
0000-0002-0855-8223

Florian Brandner
*LTCI*
*Télécom Paris*
*Institut Polytechnique de Paris*
Palaiseau, France
0000-0002-2493-7864

***Abstract*—Confidentiality is a crucial security property for many critical applications. As a response to the discovery of numerous micro-architectural side channel attacks such as Spectre, allowing an attacker to extract secret information in pernicious ways, the notion of hardware/software contracts was proposed to formalise the guarantees provided by the hardware to the software. In this paper, we propose to extend this notion to include the guarantees provided by the operating system (OS), so far unspecified in such contracts. We formalize an attacker model adapted to a typical execution model on a shared platform. More precisely, we formalize common thread and memory management policies provided by the OS on top of a hardware model and explore the consequences of potential leaks emerging on such a *platform*. Our investigation shows that the OS policies play a crucial role in providing security guarantees to code processing sensitive data and thus have to be taken into consideration when writing such code through *platform contracts*.**

***Index Terms*—Side Channels, Formal Models, Operating Systems**

## I. Introduction

With the discovery of micro-architectural side channels, e.g. Spectre [1] and Meltdown [2], a new wave of research has started investigating the interactions between *security-sensitive* code and potentially *untrusted* code executing on the same machine. Such interactions, and the resulting attacks, threaten ubiquitous application scenarios, be it cryptographic keys processed on a smartphone, untrusted virtual machines sharing a physical machine in the cloud, or untrusted code executing in a web browser. Often two application scenarios are distinguished [3]: (1) sandboxing of untrusted code and (2) secure execution of code processing sensitive data.

This work is concerned with the latter case. Here, a common notion to prevent secret data from leaking to a potential attacker is *constant time execution*: The execution time of the sensitive code must not depend on the secret [4]. While there exist approaches to prove a given code constant time [5], [6], they rely on simplistic assumptions on the execution of the program on a processor. These hypotheses are invalidated by micro-architectural side channels, allowing attacks even on code validated by these approaches.

A key concept recently proposed to remedy these threats are so-called *hardware/software contracts* [3]. These contracts specify rules on the usage of instructions and their operands at the *instruction set architecture* (ISA) level. Under the condition that the code respects these rules, the hardware (micro-architecture) ensures that an attacker cannot observe a difference between executions and thus cannot extract sensitive information – effectively thwarting attacks.

However, these contracts, so far, leave out a critical player: the *operating system* (OS). Indeed, a first response to the aforementioned hardware side channels were counter-measures at the OS level [7], which indicates the importance of the OS. On the other hand, the OS may also inadvertently leak (secret) information between concurrently running programs/threads by itself. One notable example are optimizations of context switching, where certain register sets – e.g. of the *floating point unit* (FPU) – are only saved lazily [8]. The idea is to disable the FPU instructions instead of saving and restoring the respective registers upon a context switch. Execution after the context switch then continues as usual until the new execution thread attempts to execute an FPU instruction. This triggers an interrupt, which allows the operating system to transparently save the old thread's FPU state and restore the current thread's FPU state. However, the content of the old FPU state may leak via a cache side channel [8]. The issue here is that the leak is caused by an optimization of the OS. For security-sensitive code that needs to use the FPU, it is thus not enough to adhere to a hardware/software contract, which would in all likelihood prohibit FPU usage altogether due to potential leakage. The OS has to ensure that FPU states are properly switched even when the security sensitive code is not/no longer active – it has to be part of the contract.

This work represents a first step towards formalizing the role of the OS in such a *platform contract*. We first investigate which OS policies are prone to similar leaks and then propose to extend the notion of hardware/software contracts in order to express OS-level guarantees that application code should be able to rely on. More precisely, the main contributions of this work are:

- a formalization of a realistic attacker model,
- a leakage model at the hardware level that is combined with
- a leakage model at the OS level considering thread management and memory management primitives,
- the formulation of realistic platform contracts, and

This work is supported by the "France 2030" government investment plan managed by the French National Research Agency (ANR-22-PECY-0004) within the ARSENE project.

- a discussion on the robustness of secure policies.

The paper is structured as follows. In Section II we first review the existing primitives for thread and memory management through the example of FreeBSD.[1] Next, we provide an overview of our approach in Section III, before briefly covering the formal underpinning and implementation in more detail. Section V presents an evaluation of variants of the OS policies that we modeled in our framework and we discuss the implications of our findings in Section VI. Finally, we discuss related work in Section VII before concluding.

## II. Review of OS Primitives

Let's first review the main primitives of an OS that could be relevant to a *platform contract* through the example of FreeBSD. We focus on primitives regarding thread and memory management at this point.[2] Under FreeBSD the main execution unit is a *thread*, which is associated with a memory address space, holding data and code, and an execution context – its status (`RUNNING`, `DONE`, `READY`, ...) and the processor's hardware state. Threads executing in different memory address spaces are isolated from each other using hardware protection mechanisms. However, new memory regions may be added to an address space and thus become accessible to a given thread – this may concern data of other threads, or even the kernel itself, that is no longer used. New threads can be created and threads may terminate using dedicated system calls. A central component is the scheduler, which picks a `READY` thread and resumes its execution. Usually, this implies a context switch, i.e., saving the processor's hardware state of the current thread to a data structure called `PCB` (Process Control Block) and likewise restoring the context of the newly selected `READY` thread from its `PCB`. A context switch may occur due to a (timer) interrupt or when the currently running thread yields the processor (e.g., on a blocking system call or upon termination).

In the following, we will discuss the relevant primitives and system calls considered in the remainder of this work:

(i) `thr_new`:[3] Creates a thread by allocating kernel data structures and a new kernel stack. A hardware-dependent callback `cpu_thread_alloc` allocates a new `PCB`, usually on the kernel stack. The processor's hardware state is then copied from the `PCB` of the current thread using `cpu_copy_thread`. Since `thr_new`, a system call, does not trigger a context switch, the values of the source `PCB` passed to `cpu_copy_thread`, may stem from the thread's *last* context switch.

(ii) `thr_exit`:[4] Terminates the current thread and frees kernel data structures associated with it – notably the kernel stack and along with it the `PCB`. This system call eventually leads to the activation of a new thread, which is either done through the callback `cpu_switch` or `cpu_throw`, explained next.

(iii) `cpu_throw`:[5] Restores the context of a `READY` thread from its `PCB`. The state of the current thread is discarded, which means that the processor's hardware state is *not* updated in the current thread's `PCB`, which consequently preserves the state from the *last* context switch.

(iv) `cpu_switch`:[6] Switches the context from the current thread to a `READY` thread, saving/restoring the processor's hardware states to/from the respective `PCB`s. This primitive is usually invoked by an interrupt or when the running thread is blocked.

(v) `mmap`:[7] Expands the current address space by either adding new pages to the address space or mapping a file/device into the address space. In our case, we are only interested in the case when new pages are added to the address space and thus become available to the threads executing within that address space. We note that while it seems FreeBSD zeroes pages when mapping anonymous memory to a new process, such guarantee is *not* provided in the documentation.

The above discussion is specific to the primitives of FreeBSD. In the next section we will discuss how to build general *OS policies* from variants of these primitives and present an approach to investigate the impact of such policies with regard to potential *platform contracts*.

## III. System Overview

We implemented a system model in the formal language Rosette [9]. Rosette is a variant of Racket, a LISP dialect, that is capable of symbolic execution and is tightly integrated with the Z3 SMT solver [10]. This simplifies the development of abstract models and allows to verify assertions on those models. Our model is comprised of an ISA, a hardware, and an OS model. Sensitive information is explicitly tracked, which allows us to detect information leakage from a victim thread to an attacker thread through both hardware or OS-level side channels. The actions of both threads are governed by a *platform contract*, which enforces rules similar to traditional hardware/software contracts, but in addition also covers the OS.

*a) Hardware Model:* The hardware model implements a simple micro-architecture that exhibits three commonly known hardware side channels: instructions with input-dependent latencies, conditional branches, and cache/memory accesses with decoupled memory protection checks.

The hardware state consists of two parts. The architectural state that is accessible through regular *user-mode* instructions, which includes the *program counter* (`pc`), 2 registers (`rega`, `regb`), a data memory, as well as a separate instruction memory. In addition, the micro-architectural state includes internal registers of the pipeline, the data cache, and memory protection mechanisms. These are not directly accessible through the *user-mode* instructions, but might still be observable indirectly.

[1] https://www.freebsd.org/

[2] We will revisit some potential open issues in the discussion.

[3] https://man.freebsd.org/cgi/man.cgi?query=thr_new

[4] https://man.freebsd.org/cgi/man.cgi?query=thr_exit

[5] https://man.freebsd.org/cgi/man.cgi?query=cpu_throw

[6] https://man.freebsd.org/cgi/man.cgi?query=cpu_switch

[7] https://man.freebsd.org/cgi/man.cgi?query=mmap

The processor executes a *trace*, i.e., a sequence of instructions, emerging from code stored in the instruction memory. The instructions are specified by a simple abstract ISA, which covers user-mode operations on the registers, memory access operations, and conditional branches. Instructions usually take a single clock cycle to execute. The latency of multiplications is input-dependent. Likewise, memory accesses have a varying latency depending on cache state. OS primitives, discussed below, are modeled as *special* instructions and thus are explicitly represented in traces. The hardware provides memory protection through configuration registers, which can only be modified by the *privileged* OS primitives, similar to RISC-V's physical memory protection (PMP) [11].

*b) OS Model:* The OS manages the hardware in order for multiple threads to share the execution platform using special instructions representing the execution of privileged code. It schedules and manages threads, saves/restores the threads' execution contexts, and ensures isolation between the OS and the attacker/victim threads by configuring the hardware's memory protection mechanisms. All these operations are implemented as special *privileged* instructions, which are included in the hardware execution trace (see above) following the primitives identified in Section II. The behavior of the primitives can be adapted in order to realize different *OS policies*. This *customization* of policies, roughly corresponds to different ways of using or implementing the hardware-dependent callbacks in FreeBSD (`cpu_thread_alloc`, `cpu_copy_thread`, `cpu_switch`, and `cpu_throw`). While executing a primitive, the OS has full access to the entire hardware state and can manipulate it at will – contrary to the user-mode instructions available to threads, which only have limited access to the architecture state.

*c) Attacker Model:* Our attacker model allows for execution of arbitrary code by the attacker. However, it can do so only *after* its context is restored by the OS, within the limits of potential hardware/OS protection mechanisms, and only *until* the next preemption. The attacker's actions are thus controlled to some extent by the OS's policy. The attacker's observations are *exclusively* performed through regular instructions available in user mode. It may thus only *directly* observe non-privileged/unprotected parts of the architectural state, notably the general-purpose registers and memory (subject to the OS policy). Still, it is possible to *indirectly* observe side effects through the architectural state emerging from the micro-architecture (pipeline registers, cache, ...). In particular, it may measure time and observe the time instant *when* a value becomes visible at the architecture level. In addition, the attacker may invoke OS primitives. Specifically, it may ask the OS for more memory.

*d) Contracts:* The hardware model discussed above is supplemented by an ISA model, which executes the same trace sequentially and can be used to express platform contracts in a generic fashion at the ISA/architecture level. Both, the ISA and micro-architecture models, explicitly track which parts of the ISA/micro-architectural state potentially contain sensitive information. This roughly corresponds to a classical taint analysis [12], albeit with optimizations exploiting Rosette's execution model. More importantly, it is possible to formulate assertions in Rosette, which given a set of starting hypotheses can be used to verify system properties or to derive counter examples violating them. Having both the ISA model and micro-architectural hardware model at hand has several advantages.

For one, the ISA model allows us to express coding conventions and policies (cf. hardware/software contracts [3] and programming conventions for constant time code [5]) and thus effectively exclude certain kinds of programs during the verification procedure. For instance, we can formulate an assertion saying *"No conditional branch depending on a secret is allowed"*.

On the other hand, the micro-architectural hardware model can be used to verify the usual non-interference property [4], i.e., the attacker thread cannot observe any sensitive information. The same procedure can also be used to synthesize specific attack scenarios. In other terms, *assuming* that the ISA execution follows a given convention, we can *assert* that at no point in time, while the attacker is executing, can it distinguish between two possible secret values by looking at architecturally accessible registers as defined by the ISA.

This construction follows the intuitive idea that the contract between the code and the platform allows the programmer to reason about safety without having to reason about the micro-architectural details, as long as the code follows the given conventions.

### A. Trivial Example

Let's consider a trivial example of an – evidently – *unsafe* OS policy that does not ensure memory protection: all threads have access to the entire memory at all times, where the victim thread computes a secret in a register and stores that secret to a fixed address (`@0x100`). Obviously the attacker thread can, under such a weak OS policy, read from that address and thus gain access to the secret. In our framework this obvious leak is represented by the trace shown in Table I.

The execution trace shows that first the attacker thread is created, which merely executes a `nop` instruction. Then the victim thread is created, which computes a secret in register `rega` and stores that secret value into the memory using a store instruction (`st`). The micro-architecture and ISA models both trace which parts of the hardware state potentially depend on that secret value. First, only register `rega` is concerned, then also the memory address where the secret was written to (`@0x100`). After the next context switch (`cpu_switch`) the attacker thread continues execution – note that the value of *its* register `rega` was restored and so does not depend on the secret. However, it can – due to the absence of memory protection – simply read the secret from memory using a `load` instruction. The attacker's register **`rega`** consequently now holds the secret, which allows the attacker to do whatever they want to do with it. Our system thus detects a leak.

A platform contract in the presence of such a weak OS policy would need to forbid storing any information that

TABLE I
TRIVIAL INFORMATION LEAK VIA UNPROTECTED MEMORY: THE VICTIM THREAD COMPUTES A SECRET, AND STORES IT AT ADDRESS `0x100`, THE ATTACKER LATER READS FROM THAT ADDRESS AND RECOVERS THE SECRET.

| Trace | Tainted Hardware State | Comment |
|---|---|---|
| `thr_new [attacker]` | | Create attacker thread |
| `nop` | | Do nothing |
| `thr_new [victim]` | | Create and execute victim thread |
| `rega = ...` | `rega` | Compute in `rega` |
| `st [@0x100] = rega` | `@0x100, rega` | Write secret into memory (`@0x100`) |
| `cpu_switch` | `@0x100` | Switch context to attacker thread |
| `ld rega = [@0x100]` | `@0x100, rega` | Read secret from memory (`@0x100`) |

Attacker thread in red, victim thread in blue.

depends on the secret in memory – which would, of course, be too restrictive for real-world applications. However, it is easy to see that stronger OS policies are possible. Notably, one could imagine the strongest possible OS policy, which erases any secret information from the (micro-)architectural state. In this case the platform contract would not impose any restrictions on code processing sensitive data. Such a strong policy, similar to the weak policy of the example, appears unrealistic. The objective of this work is thus to investigate the relevance of conventional hardware/software contracts and coding guidelines in the presence of different OS policies.

## IV. IMPLEMENTATION CONSIDERATIONS

In this section we provide further details beyond the system-level overview from Section III. Due to space considerations we limit our discussion to implementation details on taint tracking and the cache and memory protection of the micro-architecture.

### A. Taint Tracking

All our models, including the ISA, micro-architecture, and OS, are implemented using a generic framework over functions operating on Booleans and bit vectors. All functions return a value and, in addition, a *weak taint*. Functions can be composed freely such that the taint is tracked (semi-)automatically. The models are implemented as *taint-carrying* state machines, where a *step* function takes an ISA/micro-architectural state as input and provides a new state as output – where all elements of the input/output states carry taint information. The weak taint is an over-approximation of what we call a *strong taint* – which, in turn, corresponds to the usual non-interference property [4].

The strong taint is a hyper property over two execution traces of a regular state machine, where the individual states produced over time are compared against each other. Since we are interested in the potential differences when processing different secret values, the two traces may start from arbitrary valid initial states such that *only* elements of these initial state holding the secret may be different and all other state elements have to be the same. When jointly running these two state machines, any element of the two states that differs at a given time instant is *strongly tainted*. Note that the existence of such a strongly tainted state element does not (yet) imply a leak. However, a leak is only possible when two traces exist where at least one element of the two states caries a strong taint. Proving the absence of leaks in such a model quickly leads to state explosion.

Our implementation thus does not require the construction of two traces and instead over-approximates the strong taint using propagation rules. For instance, adding a tainted value to another value produces a sum that itself is tainted. We use the usual propagation rules found in the literature for our logic and bit-vector operations [13], [12]. In addition, however, we exploit the unique features of Rosette in our implementation. For one, we use non-interpreted functions [14] and symbolic execution wherever possible in order to reduce the verification overhead. Moreover, we use Rosette's meta-programming capabilities, which allow us to invoke the SMT solver during symbolic execution and take the solver's response into consideration. Notably, this allows us to refine our taint propagation rules depending on whether the solver is able to prove certain properties or not. We will illustrate this below when discussing the cache implementation of our micro-architecture model.

### B. Micro-architectural Model

Our micro-architecture implementation is a simple two-stage pipeline (`IF`, `EX_MEM`) exhibiting various problematic features, as outlined before. In particular, we implement PMP-style memory protection, where the OS can define a lower and an upper bound on the admissible memory addresses. The corresponding configuration registers (`tl` and `th`) are not accessible through the user-mode instructions and can only be manipulated by the OS primitives.

We assume separate instruction and data memories, where instruction fetches always take a single cycle. Load (`ld`) and store (`st`) instructions compute their address and perform the actual memory/data cache access in the `EX_MEM` stage. Cache hits complete in a single cycle, while misses may take several cycles. The memory protection checks are decoupled from the cache access – similar to modern designs prone to Meltdown [2]. In case of a memory access violation the cache state is updated. However, loads retrieve a default value (`0`), while stores are discarded.

The cache design is very simple, merely holding a single tag/address and data word (32 bits). Consequently, a hit occurs when the cache's tag corresponds to the address of the current memory access, and a miss otherwise. This can be captured rather easily using traditional taint tracking rules. However,

results may be overly conservative. We thus refine the taint analysis for the tag comparison logic (and similarly also in other parts of the micro-architecture) as shown in Figure 1. The function `t-eq` compares the cache's tag with the requested memory address – both associated with a weak taint – and returns a tainted Boolean. By default, the taint of the result is obtained by computing the disjunction between the taint of the two inputs (cf l. 8). However, if the SMT solver manages to prove (in)equality between the two input addresses (l. 3/6 respectively), it is certain that all executions exhibit the same behavior – either always HIT/MISS (l. 4/7) – independent of the secret (cf. strong taint).

## V. Evaluation

In order to evaluate the impact of OS policies on potential platform contracts we apply our formal model on a set of execution scenarios exercising different hardware/OS features. The scenarios have been determined previously through the help of the system itself. The code has been implemented using Racket version 8.14-1 and a recent development snapshot of Rosette (commit @cf703c6, June 17, 2024), which relies on the Z3 SMT solver version 4.8.8 (64 bit).

We define execution scenarios, i.e., sequences of instructions and calls to OS primitives, that are prone to information leakage. A scenario usually starts by creating the attacker and/or victim thread, with a few instructions associated with each thread – similar to the example from Table I. We assume for all scenarios that the secret is placed in the address space of the victim at the moment when its thread is created. A leak is detected when the attacker is able to access a tainted user-mode register when its thread is running (cf. the highlighted **`rega`** in Table I). The following eight scenarios are considered in our experiments:

- `mul_var`: where the victim thread operates on a secret using a *variable latency* instruction.
- `br_var`: the victim executes a *conditional branch* whose outcome depends on the secret.
- `ld_var`: the victim *accesses an address* depending on the secret that *alters* its own execution time/cache history.
- `ld_cst`: the victim *accesses an address* depending on the secret *without altering* execution time/cache history.
- `exit_mem`: the victim terminates *without clearing the secret from its own memory space*.
- `exit_reg`: the victim clears its memory before terminating, but *without clearing its registers*.
- `preempt_clr`: the victim is *preempted* before clearing its memory/registers and before terminating.
- `thr_clone`: the victim is *preempted* before clearing its memory/registers and *spawning* the attacker thread.

The first four scenarios violate traditional hardware/software contracts, while the last four scenarios do not. Each scenario represents a potential element of a platform contract.

We apply each of the above scenarios to a set of OS policies in order to verify the robustness of the respective policy with regard to information leakage. Each policy is composed of the OS primitives identified in Section II, where different variants are possible:

(i) `thr_new_zero`: Allocates and *initializes* an entire `PCB` to zero in *protected* kernel space; sets the thread's user-mode address space up, including memory protection; invokes `cpu_switch`.

(i) `thr_new_copy`: Same as `thr_new_zero`, but *copies* the `PCB` of the current thread instead of clearing it.

(ii) `thr_exit_switch`: First, invokes `cpu_switch`; then, frees the `PCB` in kernel space and the thread's user-mode address space; neither `PCB` nor memory are cleared.

(ii) `thr_exit_throw`: Same as `thr_exit_switch`, but invokes `cpu_throw` instead of `cpu_switch`.

(ii) `thr_exit_safe`: Same as `thr_exit_switch`, but *clears* the `PCB` before freeing its kernel space.

(iii) `cpu_throw`: Chooses a new `READY` thread and set its state to `RUNNING`; restore the new thread's register and memory protection (`pc`, `rega`, ...); continue execution of new thread.

(iv) `cpu_switch_base`: Same as `cpu_throw`, but *first* saves the context of the current (old) thread's registers and memory protection to `PCB` (`pc`, `rega`, ...) and resets its status to `READY` (or `DONE`).

(iv) `cpu_switch_clear`: Same as `cpu_switch_base`, but in addition clears cache.

(v) `mmap_base`: Expand the user-mode address space of the current thread, updating memory protection accordingly. For simplicity, we assume contiguous address spaces, where expansion can either go towards lower (`tl`) or higher (`th`) physical addresses.

[8] As documented
[9] As implemented

```
1 (define (t-eq a b)
2   (if (unsat?             ; call out to SMT solver
3     (verify (assert (eq? (val a) (val b)))))
4       (tracked #t #f)     ; Always HIT
5   (if (unsat?             ; call out to SMT solver
6     (verify (assert (not (eq? (val a) (val b))))))
7       (tracked #f #f)     ; Always MISS
8       (t-apply f-eq a b) ; Hit or Miss possible
9 )))
```

Fig. 1. Taint tracking when comparing the cache's tag with the address of a new memory access.

TABLE II
OS policies considered in the experiments.

| | Primitives | | | | |
|---|---|---|---|---|---|
| OS policy | thr_new | thr_exit | cpu_switch | mmap | OS/Scheduler |
| POpt | _copy | _throw | _base | _base | BSD BSD4[8] |
| PZOpt | _copy | _throw | _base | _zero | BSD BSD4[9] |
| PBase | _copy | _switch | _base | _base | BSD ULE / Linux CFS |
| PTSafe | _copy | _safe | _base | _base | – |
| PCSafe | _copy | _safe | _clear | _base | – |
| PSafe | _zero | _safe | _clear | _base | – |
| PZSafe | _zero | _safe | _clear | _zero | – |

TABLE III
RESULTS OF VERIFICATION EXPERIMENTS.

| Scenario | POpt | PZOpt | PBase | PISafe | PCSafe | PSafe | PZSafe |
|---|---|---|---|---|---|---|---|
| | OS Policy | | | | | | |
| mul_var | × | × | × | × | × | × | × |
| br_var | × | × | × | × | × | × | × |
| ld_var | × | × | × | × | × | × | × |
| ld_cst | × | × | × | × | ✓ | ✓ | ✓ |
| exit_mem | × | ✓ | × | × | × | × | ✓ |
| exit_reg | × | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| preempt_clr | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| thr_clone | × | × | × | × | × | ✓ | ✓ |

× : Leakage detected ✓: No leakage detected

(v) `mmap_zero`: Same as `mmap_base`, but zero-fills the newly allocated memory.

From these variants of OS primitives we construct seven OS policies, shown in Table II, which correspond to the behavior of major open-source operating systems and their schedulers. `POpt` corresponds to the `BSD4` scheduler in FreeBSD with an optimized context switch procedure, while `PBase` corresponds to the `ULE`/`CFS` schedulers from FreeBSD/Linux respectively. It has to be noted though that the `ULE` scheduler only recently replaced[10] `cpu_throw` by `cpu_switch` and, depending on the computer architecture, still performs optimized context switches of FPU/SIMD registers. The `mmap` behaviour for anonymous mappings in `POpt` and `PBase` allows allocation of previously used memory without clearing it, as permitted by the FreeBSD documentation[11]. In practice, the system implementation zero-fills allocated memory, which is modelled in the `PZOpt` policy.

The results of our verification experiments are summarized in Table III. In the next subsections, we will discuss these results. We will first focus on scenarios that are typically covered by hardware/software contracts and that are insensitive to the considered OS policy. Next, we highlight scenarios that are not covered by hardware/software contracts, but where the OS policy may induce leakage. Finally, we study how the OS policy may partially relax usual hardware/software contracts.

### A. Policy-Insensitive Scenarios

For several execution scenarios (`mul_var`, `br_var`, and `ld_var`) the considered OS policies were not able to contain information leakage (first three lines of Table III). This is due to the fact that *execution time* depends on the secret in these scenarios, which can be measured by the attacker using the `time` instruction (l. 7). Table IV illustrates this kind of scenarios, where the secret is first loaded from memory (`@0x6`) and then used in a multiplication (`mul`, l. 5), whose execution time in our micro-architecture model depends on its input operands. Consequently, the program counter (`pc`) is tainted after the multiplication, i.e., an attacker may observe a different value in the `pc` at a given time instant depending on the secret input operand to the multiplication. The context switch clears the taint of the registers, which are saved and restored by the `cpu_switch` primitive (l. 6), but does not affect the taint of the `pc`. Finally, by measuring the time an attacker may now observe a difference in its own register (`rega`, l. 7) depending on the secret.

Since none of the considered policies is able to protect against leakage for these scenarios, they have to be excluded from any code processing secrets. Similar to traditional hardware/software contracts, our platform contracts forbid the corresponding code patterns.

### B. Policy-Induced Scenarios

The OS policy *itself* may induce information leakage, e.g., for the scenarios `exit_mem`, `exit_reg`, `preempt_clr`, and `thr_clone`. The three first scenarios have in common that the attacker may exploit the `mmap` OS primitive in order to allocate more memory. In the case of the `exit_mem` scenario this may lead to a direct leak, when, after the termination of the victim, the OS simply allocates its physical memory space to the attacker. In a similar fashion, this may occur in a more indirect way through the OS. For `exit_reg` the victim does not clear its own registers before termination. The secret information is thus eventually stored in the `PCB`, which is initially located in protected kernel space, but may get allocated to the attacker once the `PCB` is freed. Both of these scenarios can be addressed by a relatively simple platform contract, which requires that all memory and registers are cleared before the termination of the victim thread.

However, this contract is not sufficient under all circumstances. Scenario `preempt_clr`, shown in Table V, would satisfy this contract, but still leads to a leak under the policy

[10]The change was motivated by a race condition in certain hypervisors and *not* by security concerns

[11]https://man.freebsd.org/cgi/man.cgi?mmap(2)

TABLE IV
TRACE OF THE `MUL_VAR` SCENARIO, LEAKING INFORMATION THROUGH A VARIABLE-LATENCY MULTIPLICATION.

| Trace | Tainted Hardware State |
|---|---|
| 1. `thr_new [attacker]` | |
| 2. `nop` | |
| 3. `thr_new [victim]` | `@0x6` |
| 4. `ld regb = [@0x6]` | `@0x6, regb` |
| 5. `mul rega = 3 * regb` | `@0x6, pc, rega, regb` |
| 6. `cpu_switch` | `@0x6, pc` |
| 7. `time` | `@0x6, pc,` **`rega`** |

TABLE V
TRACE OF THE `PREEMPT_CLR` SCENARIO UNDER THE `POPT` POLICY, LEAKING INFORMATION THROUGH THE `PCB` STORED IN KERNEL SPACE.

| Trace | Tainted Hardware State |
|---|---|
| 1. `thr_new [victim]` | `@0x7` |
| 2. `ld rega = [@0x7]` | `@0x7, rega` |
| 3. `thr_new [attacker]` | `@0xA, @0x7` |
| 4. `nop` | `@0xA, @0x7` |
| 5. `cpu_switch` | `@0xA, @0x7, rega` |
| 6. `ldi rega = 0` | `@0xA, @0x7` |
| 7. `st [@0x7] = 0` | `@0xA` |
| 8. `thr_end [victim]` | `@0xA` |
| 9. `mmap` | `@0xA` |
| 10. `ld rega = [@0xA]` | `@0xA,` **`rega`** |

POpt. The victim thread starts by loading the secret into a register, before being preempted by the newly created attacker thread (thr_new, l. 3). At this point the secret is saved to the thread's PCB at address @0xA. A second context switch preempts the attacker. The victim now clears the secret from its registers and memory before terminating (l. 8). Contrary to the other policies, POpt invokes the cpu_throw primitive to switch the context back to the attacker thread *without touching* the victim's PCB. The PCB, however, still holds the outdated register values with the secret. Consequently, the attacker can again use the mmap primitive and attempt to allocate the physical memory space that previously held the victim's PCB.

In order to address this issue a platform contract might impose that the victim forces a cpu_switch manually after clearing the secret. However, with the currently available OS implementations this contract cannot be implemented reliably. While FreeBSD, in principle, offers a system call sched_yield, [12] a context switch is not guaranteed. The current thread might be picked immediately by the scheduler, which then simply skips the context switch. For the POpt policy, it is not possible to define a suitable platform contract.

To remedy these problems stronger policies are needed. The mmap implementation of FreeBSD (and many other OSes), for instance, zero-fills newly allocated memory space, which prevents the attacker from retrieving the secret in the above scenarios. This is shown by the results of the PZOpt policy, where the aforementioned scenarios do not lead to leakage.

The last scenario, thr_clone, is similar to preempt_clr. The main difference is that the victim does not terminate but instead *spawns* the attacker thread. All policies, except PSafe and its derivative PZSafe, copy the PCB of the victim thread, but, similar to preempt_clr, operate on an outdated PCB stemming from the last preemption. When the attacker thread gets activated it finds the secret in its register. This issue is present in some, but not all, implementations of cpu_copy_thread on FreeBSD.

PZOpt is not able to prevent the leakage in this case as the secret retrieval does not happen *via* the mmap primitive, but by the operations of thr_new which clones the PCB. Thus only PZSafe, which zero-fills both allocated memory and the registers of a new thread, prevents those four scenarios.

### C. Policy-Sensitive Scenarios

For the last remaining scenario, ld_cst, the OS policy allows for a more relaxed platform contract compared to the usual hardware/software contract (cf. Table VI). In this scenario, the victim thread uses the secret as an address for a memory load instruction (l. 5). Consequently, the tag of the cache now contains the secret, as indicated by the taint analysis. Without a counter measure, this allows an attacker to extract parts of the secret – depending on the cache's organization – by *guessing* addresses (represented by the address @_ on l. 7) and measuring the time needed for accessing those addresses. Recall that handling a cache miss and checking the address are decoupled in our micro-architecture model – similar to existing hardware side channels such a Meltdown [2].

[12] https://man.freebsd.org/cgi/man.cgi?query=sched_yield

TABLE VI
TRACE OF THE LD_CST SCENARIO UNDER THE PTSAFE SCENARIO, LEAKING INFORMATION THROUGH THE CACHE.

| Trace | Tainted Hardware State |
|---|---|
| 1. thr_new [victim] | @0x4 |
| 2. ld rega = [@0x4] | @0x4, rega |
| 3. or rega = rega \| 1 | @0x4, rega |
| 4. la rega = @0x4 + rega | @0x4, rega |
| 5. ld rega = [rega] | @0x4, tag, rega |
| 6. thr_new [attacker] | @0x4, tag |
| 7. ld rega = [@_] | @0x4, tag |
| 8. time | @0x4, tag, **rega** |

Recall that our simple data cache model always starts from an empty cache and holds only a single cache block. Rosette's symbolic execution is thus able to prove, for this example, that @0x4 is *always* different from @0x4 + (rega | 1). The cache history and execution time of the victim thread itself is thus independent from the secret. The cache (tag) is, consequently, the only side channel.

The PCSafe/PSafe/PZSafe policies even prohibit the cache side channel, since the cache is cleared on every context switch. Consequently, under these policies the tag is no longer tainted starting from l. 6. The attacker's memory access (l. 8) thus always results in a cache miss, which eliminates the leak altogether. However, in order to ensure the absence of leaks a corresponding platform contract would still need to ensure that the cache history/execution time are independent from the secret. For our simple cache model this can be achieved through a platform contract such as: for any successive memory accesses to addresses @a and @b with at least one taint, either a = b always holds or a ≠ b always holds. For realistic cache designs similar contracts can be established, e.g., by performing a cache hit/miss [15] or conflict-set analysis [16], [17]. These contracts naturally would depend on cache parameters, such as the associativity, number of sets, and cache line size and thus would only hold for platforms with identical cache parameters. When the secret-dependent address ranges are small, relaxed contracts referring to lower bounds of the cache parameters appear to be feasible. For instance, for a secret-dependent address range spanning 4 KB, cache conflict sets [18] could be used to compute a lower bound on the associativity required, assuming that the product of the number of sets and the cache line size is larger than 4 KB.

## VI. DISCUSSION

Our results show that, using a traditional non-interference property, we can verify the presence of potential secret data leakage paths across multiple policies and scenarios. Nevertheless, while this helps to ascertain whether a given policy allows for a more permissive platform contract or not, it does not seem to entirely characterise the *robustness* of the system.

Based on the insights of the previous section, we tried to mount an attack on a real system (FreeBSD 15.0 with BSD4 scheduler, `qemu-system-riscv64`). We observe, via a memory dump after victim execution, that the victim's secret remains indeed in kernel space. This is true even after the termination of the thread and after the scheduler has reclaimed the thread's `PCB` structure in its internal memory pool. We also observe that, after forcing a low memory event by executing the shell command `sysctl debug.vm_lowmem=1`, scheduler data structures previously allocated for the victim are released from this internal pool and their memory may be allocated elsewhere (e.g., for `mmap`). However, when an attacker tries to get access to this memory, for instance, via `mmap`, zero-filling prevents a direct extraction of the secret.

FreeBSD thus clearly implements the `PZOpt` policy, which in the previous section was shown to be safe. Nevertheless, it is also evident that the secret still remains accessible within freed kernel memory. It thus might be subject to bugs or mishandling by any other kernel code, which represents a considerable attack surface,[13] potentially for a very long time. This might even invalidate formal proofs, for example when the system is subject to a physical attack such as fault injection. Not only does this behaviour highlight the need for clear specifications of OS guaranties in order to properly reason about the necessary rules to follow to avoid data leakage, but it also raises the issue of secret persistence in the system state.

It seems clear that policies allowing the persistence of secrets in freed system memory are less robust than those clearing the potential secrets as soon as possible - even when a leak is impossible. This raises the question of how to characterise and compare this intuitive *robustness* of system policies. Such a robustness metric would be complementary to a security property such as non-interference, as the possibility of secret leakage precludes any kind of robustness. A secure system might however be more or less robust, as stated above. The formal definition of a metric formalising the above intuition is left for future research.

## VII. Related Work

Regarding hardware/software contracts, the concept has been formalised by Guarnieri et al. [3], where they model several micro-architectural mechanisms in order to prove the security of programs even in the presence of speculation. Building on this concept, [19] proposes a technique to actually prove that a concrete implementation of a micro-processor respects a given contract. A similar approach is presented in [20], which uses symbolic execution on a micro-architectural model to verify Spectre countermeasures and led to the discovery of a flaw in the *GhostMinion* [21] countermeasure. The attacker model in these pioneering works is very abstract – for example exposing all memory addresses to an attacker – which makes it difficult to prove security of realistic programs. In contrast, we propose a more precise attacker model that requires that the attacker actually observes a different behavior depending on the secret. We also argue that including the OS allows for more fine-grained contracts and policies.

On the software side, numerous methods exist to verify constant-time execution with varying techniques such as abstract interpretation [6] or information flow tracking [4]. Constant-time properties have also been incorporated into the compilation process: Barthe et al. [5] show the preservation of constant-time execution down to the machine-code level. However – as for the other cited techniques on the software level – it does not yet take into account more recent micro-architectural attacks.

Finally, there is existing work related to the security of operating systems. In [22], the authors propose a set of OS policies implemented in a variant of the seL4 micro-kernel in order to provide *time protection*, i.e. isolating different processes and thereby removing timing leakage. While the goal is similar to our work, it is limited to a specific OS and its proposed protections, while we propose a more general framework. SeL4 is indeed an interesting target OS. A functional verification of the kernel is presented in [23], which does, however, not include timing behavior.

More specific to scheduling and the context switching responsibilities of the OS kernel, the approach of [24] consists of using a formal ISA definition to determine a context switch's security with respect to the available registers. While this approach answers many existing and potential issues, as many leakage paths involving the OS can be expressed in terms of the ISA definition, it does not take into account the subtleties of the micro-architecture and the interactions with constant-time programming and time-based side channel attacks.

## VIII. Conclusion and Future Work

In this paper, we have presented a framework that combines hardware/software contracts with OS policies in an execution platform contract, allowing for fine-grained security guarantees. The concept has been demonstrated with several attack scenarios under different OS policies regarding thread and memory management.

We thereby highlight the need for clear and formal guaranties in the OS's specification about its policies regarding its micro-architectural operations and its user data management, both in the memory system and in its thread scheduling and management. We also discussed the implications of those policies in terms of system robustness, even in the absence of direct secret data leakage.

In the future, we would like to examine further policies, covering additional primitives such as input/output system calls. In particular, we are interested in scenarios where secret data is transmitted via OS buffers, for example for file access. In this case, the policy must ensure that the OS itself does not leak secret information. Furthermore, we plan to explore robustness metrics beyond non-interference, as discussed in Section VI. Finally, we would like to apply unbounded verification techniques in order to show that an OS implementation respects a given abstract policy, or that a contract holds on a given execution platform.

[13] see for example CVE-2024-50302, CVE-2025-0662

## References


[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Symposium on Security and Privacy*, IEEE, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018.

[3] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-Software Contracts for Secure Speculation," in *IEEE Symposium on Security and Privacy (SP)*, pp. 1868–1883, IEEE, 2021.

[4] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level Non-interference for Constant-time Cryptography," in *Conference on Computer and Communications Security*, pp. 1267–1279, ACM, 2014.

[5] G. Barthe, B. Grégoire, and V. Laporte, "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"," in *Computer Security Foundations Symposium (CSF)*, pp. 328–343, 2018.

[6] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation," *Journal of Computer Security*, vol. 27, no. 1, pp. 137–163.

[7] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in *Engineering Secure Software and Systems*, vol. 10379 LNCS, pp. 161–176, Springer, 2017.

[8] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels." arXiv, 2018. 1806.07480.

[9] E. Torlak and R. Bodik, "Growing solver-aided languages with Rosette," in *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 135–152, ACM, 2013.

[10] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[11] *The RISC-V Instruction Set Manual: Volume II - Privileged Architecture*. RISC-V International, 2024. https://riscv.org/technical/specifications/.

[12] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical Fundamentals of Gate Level Information Flow Tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1128–1140, 2011.

[13] M. Waseem, *Assistance à l'abstraction de composants virtuels pour la vérification rapide de systèmes numériques*. PhD thesis, U. of Nice-Sophia-Antipolis, 2008. https://theses.hal.science/tel-00454617.

[14] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," in *Computer Aided Verification*, pp. 78–92, Springer, 2002.

[15] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *International Symposium on Static Analysis*, pp. 52–66, Springer, 1996.

[16] G. Stock, S. Hahn, and J. Reineke, "Cache persistence analysis: Finally exact," in *Real-Time Systems Symposium*, pp. 481–494, IEEE, 2019.

[17] F. Brandner and C. Noûs, "Precise and efficient analysis of context-sensitive cache conflict sets," in *International Conference on Real-Time Networks and Systems*, pp. 44–55, ACM, 2020.

[18] F. Brandner and C. Noûs, "Precise, efficient, and context-sensitive cache analysis," *Real-Time Systems Journal*, vol. 58, no. 1, pp. 36–84, 2022.

[19] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts," in *Conference on Computer and Communications Security*, pp. 2128–2142, ACM, 2023.

[20] Y. Yang, T. Bourgeat, S. Lau, and M. Yan, "Pensieve: Microarchitectural Modeling for Security Evaluation," in *Annual International Symposium on Computer Architecture*, pp. 1–15, ACM, 2023.

[21] S. Ainsworth, "GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation," in *Annual International Symposium on Microarchitecture*, pp. 592–606, ACM, 2021.

[22] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time Protection: The Missing OS Abstraction," in *EuroSys Conference*, pp. 1–17, ACM, 2019.

[23] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski, "seL4: Formal verification of an OS kernel," in *Symposium on Operating Systems Principles*, pp. 207—220, ACM, 2009.

[24] N. S. Kalani, T. Bourgeat, G. D. H. Hunt, and W. Ozga, "Automatic isa analysis for secure context switching," 2025. arXiv:2502.06609.