On Kernel's Safety in the Spectre Era (And KASLR is Formally Dead)

Davide Davoli
Inria, Université Côte d'Azur
Sophia Antipolis, France
davide.davoli@inria.fr

Martin Avanzini
Inria, Université Côte d'Azur
Sophia Antipolis, France
martin.avanzini@inria.fr

Tamara Rezk
Inria, Université Côte d'Azur
Sophia Antipolis, France
tamara.rezk@inria.fr

ABSTRACT

The efficacy of address space layout randomization has been formally demonstrated in a shared-memory model by Abadi et al., contingent on specific assumptions about victim programs. However, modern operating systems, implementing layout randomization in the kernel, diverge from these assumptions and operate on a separate memory model with communication through system calls. In this work, we relax Abadi et al.'s language assumptions while demonstrating that layout randomization offers a comparable safety guarantee in a system with memory separation. However, in practice, speculative execution and side-channels are recognized threats to layout randomization. We show that kernel safety cannot be restored for attackers capable of using side-channels and speculative execution and introduce a new condition, that allows us to formally prove kernel safety in the Spectre era. Our research demonstrates that under this condition, the system remains safe without relying on layout randomization. We also demonstrate that our condition can be sensibly weakened, leading to enforcement mechanisms that can guarantee kernel safety for safe system calls in the Spectre era.

CCS CONCEPTS

• **Security and privacy** → **Formal security models**; **Operating systems security**; *Side-channel analysis and countermeasures.*

KEYWORDS

Speculative execution; layout randomization; memory safety; control flow integrity; kernel safety

ACM Reference Format:
Davide Davoli, Martin Avanzini, and Tamara Rezk. 2024. On Kernel's Safety in the Spectre Era (And KASLR is Formally Dead). In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670332

1 INTRODUCTION

Memory safety violations on kernel memory can result in serious ramifications for security, such as e.g. arbitrary code execution, privilege escalation, or information leakage. In order to mitigate safety violations, operating systems — such as Linux — employ address space layout randomization [18, 19, 30, 42, 43, 47]. This protection measure can prevent attacks that depend on knowledge of specific data or function location, as it introduces randomization of these addresses.

On the one hand, the efficacy of layout randomization has been formally demonstrated in Abadi et al.'s line of work [2–4], as a protective measure within a *shared-memory model* between the attacker and the victim. These results, however, are contingent on specific assumptions regarding victim programs, notably the absence of pointer arithmetic, introspection, or indirect jumps. These precise constraints shaped a controlled environment where memory safety could be enforced effectively via layout randomization. However, operating systems employing layout randomization on kernel (a.k.a. KASLR in Linux e.g. [19]) diverge from these assumptions. Notably, they operate on a separate memory model, wherein, kernel code — acting as the victim — resides on kernel memory, while user code — acting as the potential attacker — resides in user space. The interaction between the two occurs through a limited set of functions provided via system calls [55]. In the operating system's realm, system calls may be written in C and assembly code, further deviating from the restricted conditions outlined by Abadi et al. This introduces a distinction not only in the expressiveness of victim code considered but also in the underlying memory model.

Hence, our first research question emerges: can we relax the language assumptions proposed by Abadi et al. [2–4] while concurrently demonstrating that layout randomization offers a comparable safety guarantee in a system with memory separation? We affirmatively respond to this question by showcasing that layout randomization probabilistically ensures kernel safety within a classic attacker model, where users of an operating system execute without privileges and victims can feature pointer arithmetic, introspection, and indirect jumps.

On the other hand, in the current state-of-the-art of security, often referred to as the Spectre era, speculative execution and side-channels are well known to be effective vectors for compromising layout randomization [24, 26, 35, 37–39]. Indeed, our first result neglects the impact of speculative execution and side-channels. Recognizing this limitation, our second research question arises: can we restore a similar safety result in the Spectre era?

In this regard, we formally acknowledge that by relying solely on layout randomization it is not possible to restore kernel safety. We then introduce a new condition, called *speculative layout non-interference* akin to speculative constant-time [14], which intuitively asserts that victims should not unintentionally leak information on the kernel's layout through side-channels. Our research formally demonstrates that under this assumption, the system is safe, and perhaps surprisingly, without the necessity of layout randomization. Later, we show that speculative layout non-interference is not a necessary requirement, and this motivates us to study how safety can be enforced without requiring that property.

Our third contribution is to show that kernels can be protected even without requiring *speculative layout non-interference*. Following other similar works [17, 57], we do so by relating safety in the classic execution model to the speculative one. We achieve this result by defining a program transformation by which we enforce safety against speculative attackers on a system that does not conform to this property. This transformation, in turn, requires the system to enjoy a notion of safety that cannot be provided by layout randomization, but that is sensibly weaker than the safety property it enforces. This marks the first formal step toward strengthening kernel safety in the presence of speculative and side-channel vulnerabilities, and the surpassing of layout randomization as a system level protection mechanism.

In summary, our contributions are:

- We formally demonstrate the effectiveness of layout randomization to provide kernel safety for a classic operating system scenario, with system calls offered as interfaces to attackers and different privilege execution modes, as well as kernel and user memory separation.
- We empower attackers in our first scenario to execute side-channel attacks and utilize speculative execution. Demonstrating that kernel safety is not maintained under this more potent attacker model, we subsequently present a sufficient condition to ensure kernel safety.
- We show that it is possible to enforce safety against speculative attackers on a system that enjoys weaker security guarantees by the application of a program transformation.

The paper is structured as follows: in Section 2 we give an overview of the contributions of this paper, motivated by some concrete examples. In Section 3, we introduce our execution model by giving its language and semantics; in Section 4, we establish threat models. Section 5 is devoted to showing that layout randomization is an effective protection measure for attacks that do not rely on speculative execution and side-channel observations. In Section 6 we first extend the model of Section 3 to encompass time-channel info leaks and speculative execution, then we show that layout randomization is not a viable protection mechanism in this scenario. In Section 7 we show that it is feasible to convert any system that is safe against classic attackers into an equivalent system that is safe against speculative attackers. Finally, we consider related work in Section 8, and we conclude in Section 9. An extended version of this paper with the complete semantics and the omitted proofs is available online [?].

2 MOTIVATION

Each year, dozens of vulnerabilities are found in commodity operating systems' kernels, and the majority of them are memory corruption vulnerabilities [53]. A kernel suffers a memory corruption vulnerability when an unprivileged attacker can trigger it to read or write its memory in an *unexpected* way, usually, by issuing a sequence of system calls with maliciously crafted arguments. In Figure 1, we show a pair of system calls of a hypothetical system

```
int buf[K+1][H];
int recv(socket* s, size_t idx) {
  if (valid(s, idx)) return buf[*s][idx];
  return 0;
}
void send(socket* s, size_t idx, int msg) {
  if (valid(s, idx)) {
    buf[*s][idx] = msg;
    if (buf[K][0] != NULL) (*buf[K][0])(s, idx);
  }
}
```

Figure 1: System Calls vulnerable to memory corruption

that are subject to this kind of vulnerability. The `recv` and `send` system calls are meant to implement a simple message passing protocol. The implementation supports up to `K` sockets, each socket storing up to `H` messages. A user can send messages by invoking the system call `send`, and read them with the system call `recv`. These system calls employ a shared buffer `buf` that stores messages, together with a hook for a customizable callback `buf[K][0]`. If specified, this callback is executed after a message is sent. Such a callback may, for instance, signal the receiver that a new message is available.

These system calls are meant to interact only with the memory containing the buffer, the code of the called functions and with the resources that these function in turn access. In the following, we will refer to the set of memory resources that a system call may access rightfully as the *capabilities* of that system call. Depending on the implementation of the `valid` function, these system calls can suffer from memory corruption vulnerabilities. For instance, if the `valid` function does not perform any bound checks on the value of `idx`, these two system calls can be used by the attacker to perform arbitrary read and write operations. In particular, if the attacker supplies an out-of-bounds value for `idx` to the `recv` system call, the system call can be used to perform an unrestricted memory read. Similarly, the `send` system call can be used to overwrite any value of kernel memory and, in particular, to overwrite the function pointer to the callback that is stored within the buffer. This means that the attacker can turn this memory-vulnerability into a control-flow vulnerability, as it can deviate the control flow from its intended paths. When this happens, we talk about violations of control flow integrity (CFI) [1].

However, if the system that implements these system calls is protected with layout randomization — like many commodity operating systems do [18, 19, 30, 42, 43, 47] — the exploitation of these vulnerabilities is not a straightforward operation. For instance, if an attacker wants to mount a privilege-escalation attack, one of the viable ways is to disable the SMEP protection by running the `native_write_cr4` function. When this protection is disabled, the attacker is allowed to run any *payload* stored in user-space. To this aim, the attacker can use the `send` system call twice: the first time to run the `native_write_cr4` function instead of the callback, and the second time to run the *payload*. However, in order to do so, the attacker has to first infer the address of `native_write_cr4`. In the absence of side channel info-leaks, an attacker has to effectively guess this address and, due to layout randomization, the probability of success is low.

This is what we show with our first result (Theorem 5.3): without side-channel leaks (and speculative execution), if a system is protected with layout randomization, the probability that an unprivileged attacker leads the system to perform an unsafe memory access is very low, provided the address space is sufficiently large. Of course, the precise probability depends on the concrete randomization scheme. We emphasize that this result is compatible with the large number of kernel attacks that break Linux's kernel layout randomization, e.g. by means of heap overflows ([52]). The distribution of Linux's heap addresses lack entropy [21], in consequence, the probability of mounting a successful attack are relatively high.

Although it was already well known that layout randomization can provide some security guarantees [2–4], the novelty of Theorem 5.3 lies in showing that these guarantees are valid even if victims can perform pointer arithmetic and indirect jumps.

Despite this positive result, the threat model considered in Theorem 5.3 is unrealistic nowadays. In particular, it does not take in account the ability of the attackers to access side-channel info-leaks and to steer speculative execution. There is evidence that, by leveraging similar features, the attackers can leak information on the kernel's layout [26, 29, 35, 38, 39] and compromise the security guarantees offered by layout randomization [24, 37].

In particular, if the system under consideration suffers from side-channel info-leaks that involve the layout, an attacker may break the protection offered by randomization. As an illustrative example, suppose the system contains the following system call:

```
int sc_leak(x){
  if ((void*) x  == (void*) native_write_cr4)
    for(int i = 0; i < K; i++);
  return 0;
}
```

By measuring the execution time of the system call, an attacker may deduce information on the location of `native_write_cr4`. If a call `sc_leak(a)` takes sufficiently long to execute, the attacker can deduce that the address a corresponds to that of `native_write_cr4`. Once deduced, the attacker will be effectively able to disable SMEP protection via the vulnerable system call send.

Similar attacks can be mounted by taking advantage of speculative execution: in our example from Figure 1, an attacker can make use of the `read` primitive to probe for readable data without crashing the system. This can be done by supplying to the system call arguments `s` and `idx` such that `valid(s, idx)` returns false — ideally, causing an out of bound access when the return value is fetched from memory. If the attacker manages in mis-training the branch predictor, the access to `buf[*s][idx]` is performed in transient execution. Depending on the allocation state of the address referenced by `buf[*s][idx]`, two cases arise. If that address does not store any readable data, the memory violation is not raised to the architectural state, because it occurred during transient execution. Most importantly, if that address stores writable data, this operation loads a new line in the system's cache and, as soon as the system detects the mis-prediction, the execution backtracks to the latest valid state. Although this operation does not affect the architectural state, the insertion of a new line in the cache can be detected from user-space. Thus, the attacker can infer that the address referenced by `buf[*s][idx]` contains readable data, and it can make use of the vulnerabilities of the send and the recv system calls to read or write the content of that memory address. This form of *speculative probing* is very similar to what happens, for instance, in the BlindSide attack [24] that effectively defeats Linux's KASLR.

$\mathsf{Expr} \ni \mathsf{E}, \mathsf{F} ::=$	v	*values*
	$\mid x$	*register*
	$\mid \mathsf{a}$	*array identifier*
	$\mid \mathsf{f}$	*procedure identifier*
	$\mid \mathsf{op}(\mathsf{E}_1, \ldots, \mathsf{E}_n)$	*operation*
$\mathsf{Instr} \ni \mathsf{I}, \mathsf{J} ::=$	skip	*no-op*
	$\mid x := \mathsf{E}$	*assignment*
	$\mid x := *\mathsf{E}$	*memory load*
	$\mid *\mathsf{E} := \mathsf{F}$	*memory store*
	$\mid \mathsf{call}\ \mathsf{F}(\mathsf{E}_1, \ldots, \mathsf{E}_n)$	*procedure call*
	$\mid \mathsf{syscall}\ \mathsf{s}(\mathsf{E}_1, \ldots, \mathsf{E}_n)$	*system call*
	$\mid \mathsf{if}\ \mathsf{E}\ \mathsf{then}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q}\ \mathsf{fi}$	*conditional*
	$\mid \mathsf{while}\ \mathsf{E}\ \mathsf{do}\ \mathsf{P}\ \mathsf{od}$	*while loop*
$\mathsf{Cmd} \ni \mathsf{P}, \mathsf{Q} ::=$	$\epsilon \mid \mathsf{I}; \mathsf{P}$	

Figure 2: Syntax of the language.

The reader may observe that these two attacks rely on the attacker's ability to reconstruct the kernel's memory layout by collecting side-channel info-leaks. For this reason, a natural question is whether these attacks can be prevented by imposing that no information of the layouts leaks to the architectural and the micro-architectural state during the execution of system calls. It turns out that this is the case, as we show in Theorem 6.4. In practice, this mitigation is of little help though, as it would effectively rule out all system calls that access memory at runtime.

However, we are able to show that any operating system can be pragmatically turned into another system that is architecturally equivalent to it, but that is not subject to vulnerabilities that are due to transient execution. With this approach, showing that a kernel is safe in the speculative execution model, reduces to showing that the kernel under consideration is safe in the classic execution model. Notably, this holds independently of the technique that is used to show safety in the classic model. Concretely, with this approach, the attack we showed above would be prevented by disallowing the transient execution of the unsafe load operation. In turn, this can be achieved by placing an instruction that stops transient execution before that operation. The efficacy of this technique is shown in Theorem 7.4.

3 THE LANGUAGE

In this section, we introduce the language that we employ throughout the following to study the effectiveness of kernel address space layout randomization.

Syntax and informal semantics. We are considering a simple imperative `while` language. The address space is explicit, and segregated into user and kernel space. The set $\mathtt{Cmd}$ of *commands* is given in Figure 2. Memories may store *procedures* and *arrays*, i.e., sequences of *values* $v \in \mathsf{Val}$ organized as contiguous regions. The set of values is left abstract, but we assume that it encompasses at least *Boolean values* $\mathsf{Bool} \triangleq \{\mathtt{true}, \mathtt{false}\}$, *(memory) addresses* Addr, and an *undefined value* $\mathtt{null}$. Within expressions, $x \in \mathsf{Reg}$ ranges over *registers*, $\mathtt{a} \in \mathsf{ArrId}$ and $\mathtt{f} \in \mathsf{FunId}$ over *array* and *procedure identifiers*, and $\mathtt{op} \in \mathsf{Ops}$ over *operations*. *Identifiers* $\mathsf{Id} \triangleq \mathsf{ArrId} \uplus \mathsf{FunId}$ are mapped to addresses at runtime, as governed by a layout randomization scheme. The *size* (length) of an array $\mathtt{a}$ is denoted by $\mathsf{size}(\mathtt{a})$ and is fixed for simplicity, i.e., we do not model dynamic allocation and deallocation.

A command $\mathtt{P} \in \mathtt{Cmd}$ is a sequence of instructions, evaluated in-order. The instruction $x := \mathtt{E}$ stores the result of evaluating $\mathtt{E}$ within register $x \in \mathsf{Reg}$. To keep the semantics brief, expressions neither read nor write to memory. Specifically, addresses are dereferenced explicitly. To this end, the instruction $x := *\mathtt{E}$ performs a memory read from the address given by $\mathtt{E}$, and stores the corresponding value in register x. Dually, the instruction $*\mathtt{E} := \mathtt{F}$ stores the value of $\mathtt{F}$ at the address given by $\mathtt{E}$. The instruction $\mathtt{call}\ \mathtt{F}(\mathtt{E}_1, \dots, \mathtt{E}_n)$ invokes the procedure residing at address $\mathtt{F}$ in memory, supplying arguments $\mathtt{E}_1, \dots, \mathtt{E}_n$. Likewise, $\mathtt{syscall}\ \mathtt{s}(\mathtt{E}_1, \dots, \mathtt{E}_n)$ invokes a system call $\mathtt{s} \in \mathsf{Sys}$ with given arguments. The execution of a system call engages the *privileged execution mode* and thereby the accessible address space changes. To this end, the address space Addr is partitioned into *kernel-space* addresses Addr_k, visible in privileged mode, and *user-space* addresses Addr_u, visible in unprivileged mode. The remaining constructs are standard.

Stores. Regarding the address space, we categorize identifiers Id into two distinct sets: *kernel-space identifiers* Id_k and *user-space identifiers* Id_u. This distinction signifies the intended location of the corresponding objects within the memory address space. We write $\mathsf{FunId}_\mathsf{k} \subseteq \mathsf{Id}_\mathsf{k}$ and $\mathsf{FunId}_\mathsf{u} \subseteq \mathsf{Id}_\mathsf{u}$ for the *kernel-space* and *user-space procedure idenitifiers*; similar for array identifiers. Let $\mathsf{ids}(\mathtt{P}) \subseteq \mathsf{Id}$ refer to the set of identifiers literally occurring in $\mathtt{P}$. A *store* is a (well-sorted) mapping $\tau : \mathsf{Id} \to \mathsf{Arr} \cup \mathtt{Cmd}$, mapping

(1) procedure identifiers $\mathtt{f}$ to their implementation $\tau(\mathtt{f}) \in \mathtt{Cmd}$; and
(2) array identifiers $\mathtt{a}$ to arrays $\tau(\mathtt{a})$ of size $|\tau(\mathtt{a})| = \mathsf{size}(\mathtt{a})$.

Here, $\mathsf{Arr}^{(n)}$ denotes the set of *arrays* of size n, i.e., finite sequence $\vec{v}$ of values of fixed length n, and Arr the set of arrays of arbitrary size. We write $\tau =_{Id} \tau'$ if τ and τ' coincide on $Id \subseteq \mathsf{Id}$.

Capabilities. To model our notion of safety, each system call $\mathtt{s}$ is associated at runtime with a fixed set of identifiers that it is meant to access. In the following, we call that set the *capabilities* of $\mathtt{s}$. This set identifies those memory areas that are safe to access, when a certain system call is running.

Systems. Let Sys denote a (finite) set of *system call identifiers*. A *system* for Sys is a tuple $\sigma = (\tau, \gamma, \xi)$, consisting of:

- a *store* $\tau : \mathsf{Id} \to \mathsf{Arr} \cup \mathtt{Cmd}$, relating identifiers to their initial value;
- a *system call map* $\gamma : \mathsf{Sys} \to \mathtt{Cmd}$ associating system calls to their implementation; and
- a *capability map* $\xi : \mathsf{Sys} \to \mathcal{P}\left(\mathsf{Id}_\mathsf{k}\right)$ associating system calls with their capabilities.

To prevent trivial memory safety violations, we impose the following two restrictions:

(i) the code $\tau(\mathtt{f})$ associated to user space identifiers $\mathtt{f} \in \mathsf{FunId}_\mathsf{u}$ is *unprivileged*, i.e. $\mathsf{ids}(\mathtt{P}) \subseteq \mathsf{Id}_\mathsf{u}$;
(ii) the capabilities $\xi(\mathtt{s})$ of a system call $\mathtt{s}$ contain at least those identifiers I that $\mathtt{s}$ refers to in code, taking procedure calls into account. Concretely, this means that we assume $I \subseteq \xi(\mathtt{s})$, where I is the least set containing identifiers occurring in the body of $\mathtt{s}$ ($\mathsf{ids}(\gamma(\mathtt{s})) \subseteq I$), and that is closed under procedure calls (i.e., if $\mathtt{f} \in I$ then $\mathsf{ids}(\tau(\mathtt{f})) \subseteq I$).

Observe that by dropping (i) a user-space function may return information on the kernel layout, such as the address of a kernel function to the attacker, making the protection offered by layout randomization vanish. Restriction (ii) relates the capabilities of a system calls to its code, thereby avoiding trivial mis-classifications.

For the sake of simplicity, we also assume that neither kernel-space procedures nor system calls perform system calls themselves. These assumptions, although not substantial, are commonly verified by commodity operating systems.

Memories and Layouts. A store τ determines the contents of a memory, but not its layout, which is governed by a function $w : \mathsf{Id} \to \mathsf{Addr}$ associating identifiers with their concrete memory addresses. *Memories* are modeled as functions $m : \mathsf{Addr} \to \mathsf{Val} \cup \mathtt{Cmd} \cup \{*\}$, associating addresses with their content. Arrays within τ will be laid out as continuous memory regions; $* \notin \mathsf{Val}$ marks that an address is unoccupied in memory. We denote by Mem the set of all memories. In the semantics, we will keep the distinction of procedures from values, thereby modeling a W^X memory protection policy, separating writable from executable memory space.

Let κ_u and κ_k denote the size of user-space and kernel-spaces addresses Addr_u and Addr_k. For simplicity, we assume that $\mathsf{Addr}_\mathsf{u} = \{0, \dots, \kappa_\mathsf{u} - 1\}$ and $\mathsf{Addr}_\mathsf{k} = \{\kappa_\mathsf{u}, \dots, \kappa_\mathsf{u} + \kappa_\mathsf{k} - 1\}$, i.e., that user and kernel address spaces are themselves continuous and consecutive regions in memory. Given a store τ, we will always assume that the address space is sufficiently large to hold τ; that is, $\kappa_b \geq \sum_{\mathtt{id} \in \mathsf{Id}_b} \mathsf{size}(\mathtt{id})$ ($b \in \{\mathsf{u}, \mathsf{k}\}$). Here, by convention $\mathsf{size}(\mathtt{f}) \triangleq 1$.

A *(memory) layout* is a function $w : \mathsf{Id} \to \mathsf{Addr}$ that describes where objects are placed in memory. As we mentioned, an array $\mathtt{a}$ is stored as continuous block at addresses $\underline{w}(\mathtt{a}) \triangleq \{w(\mathtt{a}), \dots, w(\mathtt{a}) + \mathsf{size}(\mathtt{a}) - 1\}$ within memory. For procedure identifiers $\mathtt{f}$, we set $\underline{w}(\mathtt{f}) \triangleq \{w(\mathtt{f})\}$. We overload this notation to arbitrary sets of identifiers in the obvious way. In particular, $\underline{w}(\mathsf{ArrId})$ and $\underline{w}(\mathsf{FunId})$ refer to the address-spaces of arrays and procedures, under the given layout. We regard only layouts that associate identifiers with non-overlapping blocks ($\underline{w}(\mathtt{id}_1) \cap \underline{w}(\mathtt{id}_2) = \emptyset$ for all $\mathtt{id}_1 \neq \mathtt{id}_2$) and that respect address space separation ($\underline{w}(\mathtt{id}) \subseteq \mathsf{Addr}_b$ for $\mathtt{id} \in \mathsf{Id}_b$, $b \in \{\mathsf{u}, \mathsf{k}\}$). The set of all such layouts is denoted by Lay. Note that, by the assumptions on κ_u and κ_k, Lay is non-empty.

The application of a *layout* to a *store* results in a memory and is defined in the standard way:

$$(w \diamond \tau)(p) \triangleq \begin{cases} \mathsf{P} & \text{if } \tau(w^{-1}(p)) = \mathsf{P} \in \mathsf{Cmd}, \\ t & \text{if } w(\mathsf{a}) + k = p,\ \tau(\mathsf{a}) = \vec{v} \in \mathsf{Arr} \text{ and } \vec{v}[k] = t \\ & \quad \text{for some } \mathsf{a} \in \mathsf{ArrId} \text{ and } 0 \le k < \mathrm{size}(\mathsf{a}), \\ * & \text{otherwise,} \end{cases}$$

where $\vec{v}[k]$ denotes the k-th element of the tuple $\vec{v}$, indexed starting from 0.

Abstracting from details, we model an address space randomization scheme through a probability distribution over layouts. A specific layout w is established prior to system execution by selecting a memory layout at random. For a given system $\sigma = (\tau, \gamma, \xi)$, this choice then dictates the initial memory configuration $w \diamond \tau$. Although the semantics is itself deterministic, computation can be viewed as a probabilistic process. Notably, as instructions are layout-sensitive—such as the result of a memory load operation at a specific address is contingent on whether w positions an object at that address—kernel's safety should be construed as a property that holds in a probabilistic sense.

Operational semantics. In the following, we endow our `while`-language with a small-step operational semantics. Due to the presence of (possible recursive) procedures, configurations make use of a stack of frames. Each such frame records the command under evaluation, the register contents and the execution mode. Formally, configurations are drawn from the following BNF:

$$\begin{array}{rll} b ::= & \mathsf{u} \mid \mathsf{k}_{\mathsf{s}} & \textit{execution mode} \\ F ::= & \varepsilon \mid \langle \mathsf{P}, \rho, b\rangle : F & \textit{frame stack} \\ C, D ::= & (F, m) \mid \mathsf{err} \mid \mathsf{unsafe} & \textit{configuration} \end{array}$$

A configuration is either of the form (F, m), where F is a (non-empty) frame stack and m the current memory. A top-frame $\langle \mathsf{P}, \rho, b\rangle$ indicates that P is executed with allocated registers, modeled as a mapping $\rho : \mathrm{Reg} \to \mathrm{Val}$, in mode b. In particular, $b = \mathsf{k}_{\mathsf{s}}$ indicates that execution proceeds in privileged kernel-mode, triggered by system call s. The annotation of the *kernel-mode* flag by a system call name facilitates the instrumentation of the semantics. Indeed, every time an access to the memory is made, the semantics checks whether that address is in the capabilities of the system call that is running (if any). If the address can be accessed, the execution proceeds regularly, otherwise it halts in the unsafe state. Finally, an error err signals abnormal termination (for instance, when dereferencing a kernel-space reference in user-space mode or vice versa).

The small step operational semantics takes now the form

$$w \vdash_\sigma C \to D,$$

indicating that, w.r.t. system σ, configuration C reduces to D in one step, under layout w. The reduction rules are defined in Figure 3. Rules [Load] implements a successful memory load x := $*\mathsf{E}$. Expression E is evaluated to an address $p = [\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w}$, and the register content of x is updated with the value $m(p)$ residing at address p in the memory m. Notice that, the semantics of an expression depends, besides register contents, on the layout w that resolves identifiers to memory addresses. The side-condition $p \in \underline{w}(\mathsf{ArrId}_b)$ enforces that p refers to a value accessible in the current execution mode b (by slight abuse of notation, we disregard the system call label in kernel-mode), otherwise the instruction leads to err (see rule [Load-Error]). As such, we are modeling unprivileged execution and SMAP protection, preventing respectively the access of kernel-space addresses when in user-mode, and vice versa. The final, boxed, side-condition refers to the safety instrumentation. In kernel-mode, triggered by system call s ($b = \mathsf{k}_{\mathsf{s}}$), the rule ensures that p refers to an object within the capabilities of s ($p \in \underline{w}(\xi(\mathsf{s}))$). When this condition is violated, unsafe execution is signaled (see rule [Load-Unsafe]). In a similar fashion, the rules for memory writes and procedure calls are defined.

Rule [Call] deals with procedure calls. It opens a new frame and, by convention, places the n evaluated arguments at registers $x_1, \ldots, x_n$ in an initial register environment ρ_0. System calls, modeled by rule [SystemCall], follow the same calling convention. Note that in the newly created frame, the execution flag is set to kernel-mode. Once a procedure or system call finished evaluation, rule [Pop] removes the introduced frame from the stack. Note how the rule permits return values through a designated register *ret*. The remaining rules are standard.

Let us denote by $w \vdash_\sigma C \to^* D$ that configuration C reduces in zero or more steps to configuration D, and by $w \vdash_\sigma C\!\uparrow$ that C *diverges*. In our semantics, under layout w, any non-diverging computation halts in a terminal configuration of the form $(\langle \epsilon, \rho, b\rangle, w \diamond \tau')$, or abnormally terminates through an error err or through a safety violation unsafe. This motivates the following definition of an evaluation function:

$$Eval_{\sigma,w}(\mathsf{P}, \rho, b, \tau) \triangleq \begin{cases} \Omega & \text{if } w \vdash_\sigma (\langle \mathsf{P}, \rho, b\rangle, w \diamond \tau)\!\uparrow, \\ (v, \tau') & \text{if } w \vdash_\sigma (\langle \mathsf{P}, \rho, b\rangle, w \diamond \tau) \\ & \qquad \to^* (\langle \epsilon, \rho[ret \mapsto v], b\rangle, w \diamond \tau'), \\ \mathsf{err} & \text{if } w \vdash_\sigma (\langle \mathsf{P}, \rho, b\rangle, w \diamond \tau) \to^* \mathsf{err}, \\ \mathsf{unsafe} & \text{if } w \vdash_\sigma (\langle \mathsf{P}, \rho, b\rangle, w \diamond \tau) \to^* \mathsf{unsafe}. \end{cases}$$

Note how, in the case of normal termination, we consider the return value as well as the memory contents of the result.

4 THREAT MODEL

In our threat model, attackers are unprivileged user-space programs that execute on a machine supporting two privilege rings: user-mode and kernel-mode. The victim is the host operating system which runs in kernel mode and has exclusive access to its private memory. In particular, the operating system exposes a set of procedures, the system calls, that can be invoked by the attacker and that have access to kernel's memory. The attacker's goal is to trigger a system call to perform an unsafe memory access.

In Section 5, attackers are ordinary programs that do not control speculative execution and do not have access to side-channel info-leaks. However, the target machine implements standard mitigations against this kind of attacks. In particular, it supports data execution protection mechanisms (DEP), SMAP [16] that prevents kernel-mode access to user-space data, and SMEP [20] that prevents the execution of user-space functions when running in kernel-mode. More precisely, the above-mentioned protection mechanisms are modeled in our semantics by the preconditions of the rules [Call],

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \in \underline{w}(\mathsf{ArrId}_b) \quad \boxed{b = \mathsf{k}_{\mathsf{s}} \Rightarrow p \in \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle x := *\mathsf{E};\mathsf{P},\rho,b\rangle : F, m) \longrightarrow (\langle \mathsf{P}, \rho[x \leftarrow m(p)], b\rangle : F, m)}\,[\textsc{Load}]$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \notin \underline{w}(\mathsf{ArrId}_b)}{w \vdash_\sigma (\langle x := *\mathsf{E};\mathsf{P},\rho,\mathsf{k}_{\mathsf{s}}\rangle : F, m) \longrightarrow \mathsf{err}}\,[\textsc{Load-Error}] \qquad \frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \in \underline{w}(\mathsf{ArrId}_{\mathsf{k}}) \quad \boxed{p \notin \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle x := *\mathsf{E};\mathsf{P},\rho,\mathsf{k}_{\mathsf{s}}\rangle : F, m) \longrightarrow \mathsf{unsafe}}\,[\textsc{Load-Unsafe}]$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \in \underline{w}(\mathsf{ArrId}_b) \quad \boxed{b = \mathsf{k}_{\mathsf{s}} \Rightarrow p \in \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle *\mathsf{E} := \mathsf{F};\mathsf{P},\rho,b\rangle : F, m) \longrightarrow (\langle \mathsf{P}, \rho, b\rangle : F, m[p \leftarrow [\![\mathsf{F}]\!]_{\rho,w}])}\,[\textsc{Store}]$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \notin \underline{w}(\mathsf{ArrId}_b)}{w \vdash_\sigma (\langle *\mathsf{E} := \mathsf{F};\mathsf{P},\rho,b\rangle : F, m) \longrightarrow \mathsf{err}}\,[\textsc{Store-Error}] \qquad \frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \in \underline{w}(\mathsf{ArrId}_{\mathsf{k}}) \quad \boxed{p \notin \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle *\mathsf{E} := \mathsf{F};\mathsf{P},\rho,\mathsf{k}_{\mathsf{s}}\rangle : F, m) \longrightarrow \mathsf{unsafe}}\,[\textsc{Store-Unsafe}]$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \in \underline{w}(\mathsf{FunId}_b) \quad \boxed{b = \mathsf{k}_{\mathsf{s}} \Rightarrow p \in \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle \mathtt{call}\ \mathsf{E}(\mathsf{F}_1,\dots,\mathsf{F}_n);\mathsf{P},\rho,b\rangle : F, m) \longrightarrow (\langle m(p), \rho_0[x_1 \leftarrow [\![\mathsf{F}_1]\!]_{\rho,w},\dots,x_n \leftarrow [\![\mathsf{F}_n]\!]_{\rho,w}], b\rangle : \langle \mathsf{P},\rho,b\rangle : F, m)}\,[\textsc{Call}]$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \notin \underline{w}(\mathsf{FunId}_b)}{w \vdash_\sigma (\langle \mathtt{call}\ \mathsf{E}(\mathsf{F}_1,\dots,\mathsf{F}_n);\mathsf{P},\rho,b\rangle : F, m) \longrightarrow \mathsf{err}}\,[\textsc{Call-Error}] \qquad \frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \in \underline{w}(\mathsf{FunId}_{\mathsf{k}}) \quad \boxed{p \notin \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle \mathtt{call}\ \mathsf{E}(\mathsf{F}_1,\dots,\mathsf{F}_n);\mathsf{P},\rho,\mathsf{k}_{\mathsf{s}}\rangle : F, m) \longrightarrow \mathsf{unsafe}}\,[\textsc{Call-Unsafe}]$$

$$\frac{}{w \vdash_\sigma (\langle \mathtt{syscall}\ \mathsf{s}(\mathsf{F}_1,\dots,\mathsf{F}_n);\mathsf{P},\rho,b\rangle : F, m) \longrightarrow (\langle \gamma(\mathsf{s}), \rho_0[x_1 \leftarrow [\![\mathsf{F}_1]\!]_{\rho,w},\dots,x_n \leftarrow [\![\mathsf{F}_n]\!]_{\rho,w}], \mathsf{k}_{\mathsf{s}}\rangle : \langle \mathsf{P},\rho,b\rangle : F, m)}\,[\textsc{SystemCall}]$$

$$\frac{}{w \vdash_\sigma (\langle \epsilon,\rho,b\rangle : \langle \mathsf{P},\rho',b'\rangle : F, m) \longrightarrow (\langle \mathsf{P}, \rho'[ret \leftarrow \rho(ret)], b'\rangle : F, m)}\,[\textsc{Pop}]$$

$$\frac{}{w \vdash_\sigma (\langle \mathtt{skip};\mathsf{P},\rho,b\rangle : F, m) \longrightarrow (\langle \mathsf{P},\rho,b\rangle : F, m)}\,[\textsc{Skip}] \qquad \frac{}{w \vdash_\sigma (\langle x := \mathsf{E};\mathsf{P},\rho,b\rangle : F, m) \longrightarrow (\langle \mathsf{P}, \rho[x \leftarrow [\![\mathsf{E}]\!]_{\rho,w}], b\rangle : F, m)}\,[\textsc{Op}]$$

$$\frac{}{w \vdash_\sigma (\langle \mathtt{if}\ \mathsf{E}\ \mathtt{then}\ \mathsf{P}_{\mathtt{true}}\ \mathtt{else}\ \mathsf{P}_{\mathtt{false}}\ \mathtt{fi};\mathsf{Q},\rho,b\rangle : F, m) \longrightarrow (\langle \mathsf{P}_{[\![\mathsf{E}]\!]^{\mathsf{Bool}}_{\rho,w}};\mathsf{Q},\rho,b\rangle : F, m)}\,[\textsc{If}]$$

$$\frac{C_{\mathtt{true}} = (\langle \mathsf{P};\mathtt{while}\ \mathsf{E}\ \mathtt{do}\ \mathsf{P}\ \mathtt{od};\mathsf{Q},\rho,b\rangle : F, m) \quad C_{\mathtt{false}} = (\langle \mathsf{Q},\rho,b\rangle : F, m)}{w \vdash_\sigma (\langle \mathtt{while}\ \mathsf{E}\ \mathtt{do}\ \mathsf{P}\ \mathtt{od};\mathsf{Q},\rho,b\rangle : F, m) \longrightarrow C_{[\![\mathsf{E}]\!]^{\mathsf{Bool}}_{\rho,w}}}\,[\textsc{While}]$$

Figure 3: Semantics w.r.t. system $\sigma = (\tau, \gamma, \xi)$.

[Load], [Store] that prevent the system from: (i) overwriting functions, (ii) execute values, (iii) accessing user-space data and functions when the system is in kernel-mode. Most importantly, the system adopts kernel address space layout randomization, that is modeled by sampling the memory layout from a probability distribution.

In Section 6, we then consider a stronger threat model where, in addition, attackers have access to side-channel observations and control PHT and STL predictions, related to Spectre v1 and v4 vulnerabilities [33]. In addition to the above-mentioned mechanisms against speculative attacks, the machine supports PTI [32] to prevent the speculative access of kernel-space memory from user-space; this is modeled by using the same preconditions of rules [Call], [Load], [Store] for their speculative counterparts, see Section 6.1.1.

5 CLASSIC THREAT MODEL

In this section we show how the result of Abadi et. al. [2–4] scales to the model introduced in Section 3. We formalize memory safety as follows:

Definition 5.1 (Kernel safety). We say that a system $\sigma = (\tau, \gamma, \xi)$ is *kernel safe*, if for every layout w, *unprivileged* attacker $\mathsf{P} \in \mathtt{Cmd}$, and registers ρ, we have:

$$\neg \left(w \vdash_\sigma (\langle \mathsf{P}, \rho, \mathsf{u}\rangle, w \diamond \tau) \longrightarrow^* \mathsf{unsafe}\right)$$

Thus, safety is broken if an attacker P, executing in unprivileged user mode, is able to trigger a system call in such a way that it accesses, or invokes, a kernel-space object outside its capabilities. The source of such a safety violation can be twofold:

1. **Scope extrusion:** An obvious reason why kernel-safety may fail is due to apparent communication channels, specifically through the memory and procedure returns. As an illustrative example, consider the two system calls `s1` and `s2` shown below:

```
void *v();
void s1() { v = &f; };
void s2() { (*v)(); };
```

A malicious program can use s1 to store the address of f at v, which is a shared capability. A consecutive call to s2 then breaks safety if f is not within the capabilities of s2.

2. **Probing:** Another counterexample is given by a system call accessing memory based on its input, such as

```
void s(a) { (*a)(); };
```

which directly invokes the procedure stored at the kernel-address a, that is supplied as argument. This system call can potentially be used as a gadget to invoke an arbitrary kernel-space function from user-space. Since an attacker lacks knowledge of the kernel-space layout, such an invocation needs to happen effectively through probing. As any probe of an unused memory address leads to an unrecoverable error,[1], the likelihood of an unsafe memory access is, albeit not zero, diminishingly small when the address-space is reasonably large.

To overcome Issue 1, we impose a form of (layout) *non-interference* on system calls.

Definition 5.2 (Layout non-interference). Given $\sigma = (\tau, \gamma, \xi)$, a system call s is *layout non-interfering*, if,

$$Eval_{\sigma,w_1}(\gamma(\mathsf{s}), \rho, \mathsf{k}_\mathsf{s}, \tau') \cong Eval_{\sigma,w_2}(\gamma(\mathsf{s}), \rho, \mathsf{k}_\mathsf{s}, \tau')$$

for all layouts w_1, w_2, registers ρ and stores $\tau' =_{\mathsf{FunId}} \tau$. Here, the equivalence $\cong$ extends equality by identifying the abnormal termination states err and unsafe. The system σ is non-interfering if all its system calls are.

In effect, layout non-interfering systems do not expose layout information, neither through the memory nor through return values. In particular, observe how non-interference rules out Issue 1, as witnessed by two layouts placing f at different addresses in kernel-memory.

Concerning Issue 2, it is well known that layout randomization provides in general safety not in an absolute sense, but *probabilistically* [4, 11, 56]. Indeed, the chance for a probe to be successful is proportional to the ratio between occupied and free (kernel) memory space. Following Abadi and Plotkin [4], let μ be a *probability distribution* of layouts, i.e., a function $\mu : \mathsf{Lay} \to [0, 1]$ assigning to each layout $w \in \mathsf{Lay}$ a probability $\mu(w)$ (where $\sum_{w \in \mathsf{Lay}} \mu(w) = 1$). Without loss of generality, we assume that the layout of public, i.e. user-space, addresses is fixed. That is, we require for each w_1, w_2 with non-zero probability in μ, that $w_1(\mathsf{id}) = w_2(\mathsf{id})$ for all $\mathsf{id} \in \mathsf{Id}_\mathsf{u}$. For a system call $\mathsf{s} \in \mathsf{Sys}$, let $\mathsf{id}^\mathsf{s}_1, \ldots, \mathsf{id}^\mathsf{s}_k$ be an enumeration of its capabilities $\xi(\mathsf{s})$. The following probability δ_μ quantifies the chance that accessing an address $p \in \mathsf{Addr}_\mathsf{k}$ causes an error (rather than an unsafe access), given that capabilities $\mathsf{id}^\mathsf{s}_1, \ldots, \mathsf{id}^\mathsf{s}_h$ are stored at addresses $p_1, \ldots, p_h$ respectively, and that p does not refer to any of the objects within the capabilities of s.

$$\begin{aligned}\delta_\mu \triangleq \min\Big\{ &\Pr_{w \leftarrow \mu}[p \notin \underline{w}(\mathsf{Id}) \mid w(\mathsf{id}^\mathsf{s}_i) = p_i,\ \text{for } 1 \le i \le h] \\ &\mid \mathsf{s} \in \mathsf{Sys}, p, p_1, \ldots, p_h \in \mathsf{Addr}_\mathsf{k} \wedge \\ &\quad p \notin \{p_i, \ldots, p_i + \mathsf{size}(\mathsf{id}^\mathsf{s}_i) - 1\},\ \text{for } 1 \le i \le h\Big\}.\end{aligned}$$

More concretely, δ_μ is the probability that, during the execution of a system call s, a fixed kernel address p is not allocated, given that it does not store any object that is the capabilities of that system call. Notably, if an attacker controls the value of p, this is a lower bound to the probability that its guess fails. We arrive now at the main result of this section:

Theorem 5.3. *Let $\sigma = (\tau, \gamma, \xi)$ be layout non-interfering. Then*

$$\mathbb{P}_{w \leftarrow \mu}\left[w \vdash_\sigma ((\mathsf{P}, \rho, \mathsf{u}), w \diamond \tau) \to^* \mathsf{unsafe}\right] \le 1 - \delta_\mu,$$

for any unprivileged *attacker* $\mathsf{P} \in \mathtt{Cmd}$, *and registers* ρ.

Theorem 5.3 extends the results of [2–4] by showing that layout randomization guarantees *kernel safety* probabilistically to operating systems; in contrast with [2–4], this holds even when victim's code contains unsafe programming constructs such as arbitrary pointer arithmetic and indirect jumps. This is achieved by replacing Abadi and Plotkin [4]'s restrictions on the syntax of the victims with a *weaker* dynamic property: *layout non-interference.* Notice that the strength of the security guarantee provided by Theorem 5.3 depends on the distribution of the layouts μ. Therefore, in practice, it is important to determine a randomization scheme that provides a good bound. This can be done quite easily: for instance, if we assume that (i) $\kappa_\mathsf{k} \gg \sum_{\mathsf{id} \in \mathsf{Id}_\mathsf{k}} \mathsf{size}(\mathsf{id})$ and that (ii) $\theta \triangleq \max_{\mathsf{id} \in \mathsf{Id}_\mathsf{k}}(\mathsf{size}(\mathsf{id}))$ divides κ_k, we can think of the kernel space address range as divided in $\frac{\kappa_\mathsf{k}}{\theta}$ slots, each storing a single object. In this setting, we can define the distribution ν as the uniform distribution of all the layouts that store each *memory object* within a *slot* starting from the beginning of that slot. For this simple scheme, we can approximate the bound δ_ν as the ratio between unallocated slots and all the slots that do not store an object that is referenced by the capabilities of a system call:

$$\delta_\nu \ge \min_{\mathsf{s} \in \mathsf{Sys}} \frac{\kappa_\mathsf{k}/\theta - \mathsf{size}(\mathsf{Id}_\mathsf{k})}{\kappa_\mathsf{k}/\theta - \mathsf{size}(\xi(\mathsf{s}))}.$$

In particular, the fraction in the right-hand side is the probability that by choosing a slot that is not storing any object referenced by s, we end up with a fully unoccupied slot. Observe that this lower-bound approaches 1 when κ_k goes to infinity.

Now that we have established that Layout Randomization provides *kernel safety* in a probabilistic sense, it is worth mentioning how this property relates with other desirable safety properties. In particular, *kernel safety* encompasses some form of *spatial memory safety* and of *control flow integrity*, that are maybe the most sought-after security properties for operating systems, as witnessed by the large number of measures that, together with layout randomization, have been developed for their enforcement [22, 36, 45, 48, 50, 58].

Spatial Memory Safety. Although it is difficult to find a common definition of *spatial memory safety*, many of these definitions associate a software component (a program, an instruction, or even a variable) with a fixed memory area, that this component can access rightfully [7, 10, 44, 46]. In this realm, any load or store operation

[1]This is not always the case for *user-space* software protected with layout randomization, as some programs (e.g. web servers) may automatically restart after a crash to ensure availability. This behavior can be exploited by attackers to probe the entire memory space of the victim program, thus compromising the protection offered by layout randomization [54].

that does not fall within this area is considered a violation of *spatial memory safety*. Our notion of *kernel safety* encompasses a form of spatial memory safety: if a system enjoys *kernel safety*, then no system call can access a memory region that does not appear within its capabilities.

Control Flow Integrity. This property requires that the control transfer operations performed by a program can reach only specific targets that are determined statically [1]. If a system enjoys *kernel safety* then it also enjoys a weak form of *control flow integrity*. Indeed, our semantics prevents the execution of a function if its address does not belong to the set of capabilities of the current system call. This means that if a system $\sigma = (\tau, \gamma, \xi)$ is *kernel safe*, we are certain that by executing that system call, the control flow will flow across the procedures that belong to $\xi(\mathsf{s})$.

6 SPECULATIVE THREAT MODEL

In this section we establish to which extent a system enjoys *kernel safety* in presence of speculative attackers. To this aim, in Section 6.1, we extend the model of Section 3 for this new scenario. More precisely, we endow the semantics of Section 3 with speculative execution and side-channel observations that reveal the accessed addresses and the value of conditional branches [10, 14, 27, 28]. In Section 6.2, we refine the notion of *kernel safety* for this model, by defining *speculative kernel safety*.

6.1 The Speculative Execution Model

A substantial difference between our model and previous models [10, 14, 27, 28] lies in the possibility to explicitly model attackers. More precisely, an attacker is not given as a mere sequence of microarchitectural directives, but becomes a fully-fledged program that can directly interact with the system. This permits us to naturally extend the notion of *kernel safety* to the new scenario. Besides, we believe that modeling an attack explicitly can be interesting on its own. The feasibility of an attack is witnessed explicitly through a program. In this setting, for instance, assumptions on the attacker's computational capabilities can be imposed seamlessly.

6.1.1 Victim Language and Semantics. The victims' language remains identical to the classic model. To permit attackers to influence the speculative execution of specific instructions, we assume load and branch instructions are tagged by unique labels $\ell \in \mathsf{Lbl}$, We enrich the language with a *fence instruction* found in modern CPUs [31]:

$$\mathsf{Instr} \ni \mathtt{I} ::= \cdots \mid \mathtt{fence}.$$

Architecturally, this instruction is a no-op, on the microarchitecture level it commits all buffered writes to memory. Following Barthe et al. [10], the speculative semantics is instrumented through *directives*, modeling the choice made by prediction units of the processor. Directives take the form

$$d \ni \mathsf{Dir} ::= \mathsf{br}_\ell\, b \mid \mathsf{ld}_\ell\, i \mid \mathsf{bt} \mid \mathsf{st},$$

where $i \in \mathbb{N}$ and $b \in \mathsf{Bool}$. The $\mathsf{br}_\ell\, b$ directive causes a branch instruction to be evaluated as if the guard resolved to b. The $\mathsf{ld}_\ell\, i$ causes the load instruction to load the i-th most recent value that is associated to an address in a (buffered) memory. The bt directive is used to direct speculations, either backtracking the most recent mis-speculation or committing the microarchitectural state. Finally, the st directive evaluates an instruction without engaging into speculation, in correspondence to the semantics we have given in Section 3.

The semantics is also instrumented with observations to model timing side-channel leakage:

$$o, q \ni \mathsf{Obs} ::= \circ \mid \mathsf{br}\, b \mid \mathsf{mem}\, p \mid \mathsf{jmp}\, p \mid \mathsf{bt}\, b,$$

where $n \in \mathbb{N}$ $b \in \mathsf{Bool}$, and $p \in \mathsf{Addr}$. We use $\circ$ to label transitions that do not leak observations. The $\mathsf{br}\, b$ observation is caused by branching instructions, with b reflecting the taken branch. The $\mathsf{mem}\, p$ observation is caused by memory access, through loads or stores, and contains the address of the accessed location, thus modeling instruction-cache leaks. Likewise, the $\mathsf{jmp}\, p$ observation is caused by calls to procedures residing at address p in memory. Finally, the $\mathsf{bt}\, b$ observation signals a backtracking step during speculative execution. Notice that we leak full addresses on memory accesses, and the value of the branching instructions, i.e. we adopt the *baseline leakage model* that is widely employed in the literature to model side-channel info-leaks [5, 9, 10, 14].

A reduction step now takes the form

$$w \vdash_\sigma S \xrightarrow[d]{o} S',$$

indicating that for a given system σ, under layout $w \in \mathsf{Lay}$, the system evolves from state S with directive $d \in \mathsf{Dir}$ to S' in one step, producing the side-channel observation o. The state of a system S is now modeled as a stack of backtrackable configurations. Specifically, configurations follow the following BNF:

$$C, D ::= (F, \mu m, b_{ms}) \mid (\mathsf{err}, b_{ms}) \mid \mathsf{unsafe}$$

In a configuration $(F, \mu m, b_{ms})$, F is a call-stack as in Section 3, μm is a memory equipped with a *write buffer* μ, and b_{ms} the *mis-speculation* flag. Buffered memories μm permit out-of-order, speculative memory operations. Specifically, writing a value v at address p results in a delayed write $[p \mapsto v]\mu m$, and $(\mu m)^k(p)$ yields the kth-last buffered entry v at address p, together with a boolean flag f that it $\bot$ if and only if v is the most recent one associated to address p. This operation is formally described by the following function:

$$\begin{aligned}
([]m)^k(p) &\triangleq m(a), \bot \\
([p \mapsto n] : \mu m)^0(p) &\triangleq n, \bot \\
([p \mapsto n] : \mu m)^{i+1}(p) &\triangleq n', \top && \text{if } (\mu m)^i(p) = n', b \\
([p' \mapsto n] : \mu m)^i(p) &\triangleq (\mu m)^i(p) && \text{if } p \neq p'.
\end{aligned}$$

In a configuration, the mis-speculation flag b_{ms} records whether a past step of computation led to a mis-speculation. It is employed when backtracking from a speculative state. As errors are recoverable under mis-speculation, error configurations err carry also a mis-speculation flag. Finally, as in Section 3, unsafe indicates a safety violation.

Some illustrative rules of the semantics are given in Figure 4, The rules for a load instruction are very similar to the ones we give in Section 3, but attackers can take advantage of the store-to-load dependency speculation by issuing a $\mathsf{ld}_\ell\, i$ directive. When this happens, the i-th most recent value associated to the address

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad (\mu m)^i(p) = (v, f) \quad p \in \underline{w}(\mathsf{ArrId}_b) \quad \boxed{b = \mathsf{k_s} \Rightarrow p \in \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle x \ :=_\ell *\mathsf{E}; \mathsf{P}, \rho, b\rangle : F, \mu m, b_{ms}) : S \xrightarrow[\mathsf{ld}_\ell\, i]{\mathsf{mem}\, p} (\langle \mathsf{P}, \rho[x \leftarrow v], b\rangle : F, \mu m, b_{ms} \vee f) : (\langle x \ :=_\ell *\mathsf{E}; \mathsf{P}, \rho, b\rangle : F, \mu m, b_{ms}) : S} \text{[SLoad-Load]}$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad p \notin \underline{w}(\mathsf{ArrId}_b)}{w \vdash_\sigma (\langle x \ :=_\ell *\mathsf{E}; \mathsf{P}, \rho, b\rangle : F, \mu m, b_{ms}) : S \xrightarrow[\mathsf{ld}_\ell\, i]{\circ} (\mathsf{err}, b_{ms}) : S} \text{[SLoad-Error]}$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \in \underline{w}(\mathsf{ArrId_k}) \quad \boxed{p \notin \underline{w}(\xi(\mathsf{s}))} \quad d \in \{\mathsf{st}, \mathsf{ld}_\ell\, i\}}{w \vdash_\sigma (\langle x \ :=_\ell *\mathsf{E}; \mathsf{P}, \rho, \mathsf{k_s}\rangle : F, \mu m, b_{ms}) : S \xrightarrow[d]{\mathsf{mem}\, p} \mathsf{unsafe}} \text{[SLoad-Unsafe]}$$

$$\frac{C = (\langle \mathtt{if}_\ell\ \mathsf{E}\ \mathtt{then}\ \mathsf{Q}_{\mathtt{true}}\ \mathtt{else}\ \mathsf{Q}_{\mathtt{false}}\ \mathtt{fi}; \mathsf{Q}, \rho, b\rangle : F, \mu m, b_{ms})}{w \vdash_\sigma C : S \xrightarrow[\mathsf{br}_\ell\, d]{\mathsf{br}\, d} (\langle \mathsf{Q}_d; \mathsf{Q}, \rho, b\rangle : F, \mu m, b_{ms} \vee d \neq [\![\mathsf{E}]\!]^{\mathsf{Bool}}_{\rho,w}) : C : S} \text{[If-Branch]}$$

$$\frac{[\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w} = p \quad (\mu m)^0(p) = v, \bot \quad p \in \underline{w}(\mathsf{ArrId}_b) \quad \boxed{b = \mathsf{k_s} \Rightarrow p \in \underline{w}(\xi(\mathsf{s}))}}{w \vdash_\sigma (\langle x \ :=_\ell *\mathsf{E}; \mathsf{P}, \rho, b\rangle : F, \mu m, b_{ms}) : S \xrightarrow[\mathsf{st}]{\mathsf{mem}\, p} (\langle \mathsf{P}, \rho[x \leftarrow v], b\rangle : F, \mu m, b_{ms}) : S} \text{[SLoad-Step]}$$

$$\frac{C = (F, \mu m, \top) \vee C = (\mathsf{err}, \top)}{w \vdash_\sigma C : S \xrightarrow[\mathsf{bt}]{\mathsf{bt}\, \top} S} \text{[Bt}_\top\text{]} \qquad \frac{C = (F, \mu m, \bot) \vee C = (\mathsf{err}, \bot) \quad S \neq \epsilon}{w \vdash_\sigma C : S \xrightarrow[\mathsf{bt}]{\mathsf{bt}\, \bot} C : \epsilon} \text{[Bt}_\bot\text{]}$$

$$\frac{}{w \vdash_\sigma (\langle \mathtt{fence}; \mathsf{P}, \rho, b\rangle : F, \mu m, \bot) : S \xrightarrow[\mathsf{st}]{\circ} (\langle \mathsf{P}, \rho, b\rangle : F, \overline{\mu m}, \bot) : S} \text{[Fence]}$$

Figure 4: Speculative semantics, excerpt.

$p = [\![\mathsf{E}]\!]^{\mathsf{Addr}}_{\rho,w}$ is retrieved from the buffered memory, as described by rule [SLoad-Load]. This value may not correspond to that of the most recent store to the address p, and this is signaled by the flag f that is returned after the buffer lookup. If $f = \top$, this operation may be engaging mis-speculation and, for this reason, the semantics keeps track of the current configuration in the stack. A successful load produces the observation $\mathsf{mem}\, p$ that leaks the address to the attacker. The rules for erroneous and unsafe loads are [SLoad-Error] and [SLoad-Unsafe] and they are analogous to their non-speculative counterparts.

Even branch instructions can be executed speculatively by issuing the directive $\mathsf{br}\, d$ by means of the rule [If-Branch]. This causes the evaluation to continue as if the guard resolved to d. This operation leaks which branch is being executed by means of the observation $\mathsf{br}\, d$. Even in this case, the rule may be mis-speculating; for this reason, the current configuration is book-kept in the stack and the mis-speculation flag is updated.

In our semantics, every command supports the st directive, which evaluates the configuration without speculating. For instance, the [SLoad-Step] rule evaluates the $x\ :=\ *\mathsf{E}$ command by fetching the most recent value from the write buffer, instead of an arbitrary one.

When the topmost configuration of a stack carries the mis-speculation flag $\top$, the configuration can be is discarded with the rules [Bt$_\top$]. If it is $\bot$, the current state is not mis-speculating, so the whole stack of book-kept configurations can be discarded with the rule [Bt$_\bot$].

The [Fence] rule commits all the entries in the write buffer to the memory. Precisely, this operation is defined as follows:

$$\overline{[\,]m} \triangleq m \qquad \overline{[p \mapsto v] : \mu m} \triangleq \overline{\mu m}[p \leftarrow v],$$

where, by $m[p \leftarrow v]$, we denote the memory obtained by updating the value at address p with v. In particular, for consistency, a potentially mis-speculative state must be resolved. This is why this rule requires the mis-speculation flag to be $\bot$. This means that, if this configuration is reached when the flag is $\top$, the semantics must backtrack with the rule [Bt$_\top$].

6.1.2 *Attacker's Language and Semantics.*

To give a definition of kernel safety w.r.t. speculative semantics, we endow an attacker with the ability to engage in speculative executions, by issuing directives, and by the ability to read side-channel information. To this end, we extend the instructions from Section 3 as follows:

$$\begin{array}{lll}
\mathsf{Instr} \ni \mathsf{I} ::= \ldots & & \\
\quad \mid \mathtt{spec\ on}\ \mathsf{P} & & \textit{speculation on victim } \mathsf{P} \\
\quad \mid \mathtt{poison}(d) & & \textit{speculative poisoning} \\
\quad \mid x := \mathtt{observe}() & & \textit{side-channel observation} \\
\mathsf{SpAdv} \ni \mathsf{A} ::= \epsilon \mid \mathsf{I}; \mathsf{A} & &
\end{array}$$

The instruction $\mathtt{spec\ on}\ \mathsf{P}$ is used to execute a victim code P, w.r.t. the speculative semantics defined just above. By using the instruction $\mathtt{poison}(d)$, the attacker is able to to mistrain microarchitectural predictors and to control the speculative execution. Issued directives control the evaluation of victim code under speculative semantics. Dual, the instruction $x := \mathtt{observe}()$ is used to extract side-channel info-leaks, collected during speculative execution of

$$\frac{}{w \vdash_\sigma (\langle \mathtt{poison}(d); \mathsf{A}, \rho, b\rangle : F, m, D, O) \mathrel{-\!\triangleright} (\langle \mathsf{A}, \rho, b\rangle : F, m, d : D, O)}\;[\textsc{Poison}]$$

$$\frac{}{w \vdash_\sigma (\langle x := \mathtt{observe}(); \mathsf{A}, \rho, b\rangle : F, m, D, o : O) \mathrel{-\!\triangleright} (\langle \mathsf{A}, \rho[x \leftarrow o], b\rangle : F, m, D, O)}\;[\textsc{Observe}]$$

$$\frac{}{w \vdash_\sigma (\langle \mathtt{spec\ on}\ \mathsf{P}; \mathsf{A}, \rho, b\rangle : F, m, D, O) \mathrel{-\!\triangleright} (\langle \mathsf{P}, \rho, b\rangle, []m, \bot) \mid (\langle \mathsf{A}, \rho, b\rangle : F, D, O)}\;[\textsc{Spec-Init}]$$

$$\frac{w \vdash_\sigma S \xrightarrow[d]{o} S'}{w \vdash_\sigma S \mid (F, d{:}D, O) \mathrel{-\!\triangleright} S' \mid (F, D, o{:}O)}\;[\textsc{Spec-D}] \qquad \frac{w \vdash_\sigma S{\downarrow_D} \quad w \vdash_\sigma S \xrightarrow[\mathsf{st}]{o} S'}{w \vdash_\sigma S \mid (F, D, O) \mathrel{-\!\triangleright} S' \mid (F, D, o{:}O)}\;[\textsc{Spec-S}] \qquad \frac{w \vdash_\sigma S{\downarrow_D} \quad w \vdash_\sigma S{\downarrow_{\mathsf{st}}} \quad w \vdash_\sigma S \xrightarrow[\mathsf{bt}]{o} S'}{w \vdash_\sigma S \mid (F, D, O) \mathrel{-\!\triangleright} S' \mid (F, D, o{:}O)}\;[\textsc{Spec-BT}]$$

$$\frac{}{w \vdash_\sigma (\langle \epsilon, \rho, b\rangle, \mu m, \bot) \mid (\langle \mathsf{A}, \rho', b'\rangle : F, D, O) \mathrel{-\!\triangleright} (\langle \mathsf{A}, \rho', b'\rangle : F, \overline{\mu m}, D, O)}\;[\textsc{Spec-Term}]$$

$$\frac{}{w \vdash_\sigma (\mathsf{err}, \bot) \mid (F, D, O) \mathrel{-\!\triangleright} \mathsf{err}}\;[\textsc{Spec-Error}] \qquad \frac{}{w \vdash_\sigma \mathsf{unsafe} \mid (F, D, O) \mathrel{-\!\triangleright} \mathsf{unsafe}}\;[\textsc{Spec-Unsafe}]$$

Figure 5: Semantics for speculative attackers, excerpt.

the victim's code. To model this operation, in the following, we assume Obs ⊆ Val. As an example, the snippet

```
poison(br_ℓ ⊤);                                (†)
spec on if_ℓ E then syscall s(p) fi;
x := observe()
```

forces the mis-speculative execution of `syscall s(p)`, independently of the value of E. The register x will hold the final observation leaked through executing the system call.

The attacker's semantics is defined in terms of a relation

$$w \vdash_\sigma C \mathrel{-\!\triangleright} C'.$$

In essence, the attacker executes under the standard semantics given in Section 3, the speculative semantics defined above play a role only when execution of the victim is triggered by the directive `spec on` P. Consequently, configurations are identical in structure to the ones underlying the standard semantics, but carry however additionally stacks D and O of directives and observations, in order to model the new constructs. In addition, hybrid configurations $S \mid (F, D, O)$ are used to model the system when executing the victim under speculative semantics. Here S is a stack of speculative configurations concerning the victim, and F the attacker's call stack up to the invocation of speculation. Again, D gives the directives (to be processed) and O the observations (collected from executing victim's code). In summary, configurations are drawn from the following BNF:

$$C ::= (F, m, D, O) \mid S \mid (F, D, O) \mid \mathsf{err} \mid \mathsf{unsafe}$$

Figure 5 shows the evaluation rules for the new constructs. Rules [Poison] and [Observe] define the semantics for poisoning and side-channel observations, by pushing and redacting elements of the corresponding stacks. Rule [Spec-Init] deals with the initialization `spec on` P of speculative execution, starting from the corresponding initial configuration of the victim P in an empty speculation context. A frame for the continuation of the attacker A is pushed on the call stack S. This frame is used to resume execution of the attacker, once the victim has been fully evaluated. The victim itself is evaluated via the speculative semantics through rules [Spec-D]–[Spec-BT]. Note how execution of the victim is directed through the directive stack D (rule [Spec-D]). Should the current directive be inapplicable, a non-speculative rewrite step (rule [Spec-S]) or backtracking (rule [Spec-BT]) is performed. Here, the premise $w \vdash_\sigma S{\downarrow_d}$ signifies that S is irreducible w.r.t. the directive d. Likewise, $w \vdash_\sigma S{\downarrow_D}$ means that S is irreducible w.r.t. the topmost directive of D, or that D is empty. Note also how side-channel leakage, modeled through observations, is collected in the configuration via these rules. Upon normal termination, resuming of evaluation of the attacker is governed by rule [Spec-Term] in the case of normal termination. Finally, rules [Spec-Error] and [Spec-Unsafe] deal with abnormal termination.

We write $\mathrel{-\!\triangleright}^*$ for the multistep reduction relation induced by $\mathrel{-\!\triangleright}$, i.e, $S \xrightarrow[\epsilon]{\epsilon} S$ and $S \xrightarrow[d:D]{o:O}{}^* S'$ if $S \xrightarrow[d]{o} \cdot \xrightarrow[D]{O}{}^* S'$.

6.2 Speculative Kernel Safety

We are now ready to extend the definition of kernel safety (Definition 5.1) to the speculative semantics.

Definition 6.1 (Speculative kernel safety). We say that a system $\sigma = (\tau, \gamma, \xi)$ is *speculative kernel safe* if for every unprivileged attacker A ∈ SpAdv, every layout w, and register map ρ, we have:

$$\neg \left(w \vdash_\sigma (\langle \mathsf{A}, \rho, \mathsf{u}\rangle, w \diamond \tau, \epsilon, \epsilon) \mathrel{-\!\triangleright}^* \mathsf{unsafe} \right).$$

It is important to note that this safety notion captures violations that occur during transient execution. This is in line with what happens, for instance, for Spectre and Meltdown [33, 37], both exploiting unsafe memory access under transient execution in order to reveal confidential information.

6.3 The Demise of Layout Randomization in the Spectre Era

A direct consequence of Definition 6.1 is that every system that is *speculative kernel safe* is also *kernel safe*. The inverse, of course, does not hold in general. Most importantly, the probabilistic form of safety provided by layout randomization in Section 5 does not scale to this extended threat model. This happens because Definition 5.1 does not take side-channel leakage into account. As a simple example, gadgets like

$$\texttt{if f} = p \texttt{ then P else Q fi},$$

can be exploited by an attacker to infer information about the address of a kernel-space procedure f, through side-channel leakage distinguishing the execution of P and Q. In our model, this is reflected as executing this instruction allows the attacker to observe br b, with b being true precisely when f resides at address p. Secondly, speculative execution undermines a fundamental premise crucially leveraged in Theorem 5.3 and, more widely, in the majority of studies demonstrating the efficacy of layout randomization as a defense against attacks (e.g., [2–4]): the notion that an unsuccessful memory probe leads to abnormal termination, thus thwarting the attack. Indeed, within transient executions, memory access violations are recoverable.

This happens, for instance, if the system call s of (†) tries to load the content of the address p form the memory to a register. If p is not allocated, the system performs a memory access violation under transient execution, that does not terminate the execution. Conversely, if p is allocated, its content is loaded into the cache, producing the observation mem p before the execution of the branch and the system call are backtracked. By reading side-channel observations, the attacker can thus distinguish allocated kernel-addresses from those that are not allocated. This last example, in particular, is not at all fictitious: the BlindSide attack [24] uses the same idea to break Linux's KASLR and locate the position of kernel's executable code and data.

6.4 Speculative Layout Non-Interference

As this revised model significantly enhances the attackers' strength, we will need to implement more stringent countermeasures in order to restore kernel safety. To counter side-channel info-leaks, we can impose a form of side-channel non-interference that is in line with the notion of *speculative constant-time* from [14].

Definition 6.2 (Speculative layout non-interference). Given $\sigma = (\tau, \gamma, \xi)$, a system call s is *speculative layout non-interfering*, if,

$$w_1 \vdash_\sigma (\langle \gamma(\mathsf{s}), \rho, \mathsf{k}_\mathsf{s} \rangle, \mu(w_1 \diamond \tau'), b_{ms}) \xrightarrow[D]{O}\!\!\!\triangleright^* S_1$$

implies

$$w_2 \vdash_\sigma (\langle \gamma(\mathsf{s}), \rho, \mathsf{k}_\mathsf{s} \rangle, \mu(w_2 \diamond \tau'), b_{ms}) \xrightarrow[D]{O}\!\!\!\triangleright^* S_2,$$

for all layouts w_1, w_2, configurations over stores $\tau' =_{\mathsf{FunId}} \tau$ coinciding on procedures ($\tau'(\mathsf{f}) = \tau(\mathsf{f})$ for all $\mathsf{f} \in \mathsf{FunId}$), directives D, observations O and register map ρ.

Speculative layout non-interference effectively prevents side-channel-related attacks, even during transient executions. Importantly, it ensures the non-leakage of layout information throughout the side-channels by requiring the identity of the two sequences of observations produced by the two reductions. This, however, implies severe restrictions on memory interactions — effectively prohibiting the use of random memory layouts! Unsurprisingly, this form of non-interference directly establishes kernel safety of system calls:

Lemma 6.3. *For every system $\sigma = (\tau, \gamma, \xi)$, if*

$$\kappa_\mathsf{k} \geq \sum_{\mathsf{id} \in \mathsf{Id}_\mathsf{k}} \mathsf{size}(\mathsf{id}) + 2 \cdot \max_{\mathsf{id} \in \mathsf{Id}_\mathsf{k}} \mathsf{size}(\mathsf{id}),$$

and if s is speculative layout non-interfering, then

$$\neg \left(w \vdash_\sigma (\langle \gamma(\mathsf{s}), \rho, \mathsf{k}_\mathsf{s} \rangle, \mu(w \diamond \tau'), b_{ms}) \xrightarrow[D]{O}\!\!\!\triangleright^* \mathsf{unsafe} \right)$$

for all layouts w and initial configurations over stores τ' coinciding with τ on FunId.

Intuitively, this statement holds because if an invocation of a system call s performs an unsafe memory access when executing under a layout w, the address p of the accessed resource is leaked; but the same address cannot leak if the resource is moved to another location — this is why we impose the condition on the size of the memory. Thus, if a system call is speculative layout non-interfering, it cannot be speculative non-interferent because different memory layouts produce different observations.

In general, it is not always the case that a non-interference property has as consequence memory safety. For instance, being *non-interferent* with respect to a set of secrets does not prevent a victim program from breaking memory safety. In our case, this property holds because the layouts are not only used as the inputs for a computation, but they also determine *where* objects are placed in memory.

Theorem 6.4. *Under the assumption $\kappa_\mathsf{k} \geq \sum_{\mathsf{id} \in \mathsf{Id}_\mathsf{k}} \mathsf{size}(\mathsf{id}) + 2 \cdot \max_{\mathsf{id} \in \mathsf{Id}_\mathsf{k}} \mathsf{size}(\mathsf{id})$, if a system is speculative layout non-interfering, then it is also* speculative kernel safe.

Observe that the safety guarantee provided by *speculative layout non-interference* is not probabilistic. Although its effectiveness, layout randomization is unlikely to be restored at the software level without imposing *speculative layout non-interference*, in presence of this assumption, layout randomization is a redundant protection measure. Also notice that *speculative layout non-interference* is not a necessary condition for *speculative kernel safety*. For instance, we can take in consideration the following system:

```
void f() { skip; };
void s() { f(); };
```

where s is a system call and f is a kernel function. This system does not enjoy *speculative non-interference*, because by executing s, the address of f leaks, and this address depends on the layouts. However, this system is speculatively safe if we assume that f belongs to the capabilities of s.

7 ENFORCEMENT OF SPECULATIVE KERNEL SAFETY

Although by requiring *speculative layout non-interference*, we would be able to restore *speculative kernel safety*, this would impose important limitations on the system. For this reason, we believe it is worth investigating whether *speculative kernel safety* can be enforced without imposing *speculative layout non-interference.*

Nevertheless, directly enforcing *speculative kernel safety* is non-trivial, because it requires the developers to constantly take in account a large variety of microarchitectural behaviors that their system may run into. On the other hand, in the last decades, plenty of effort has been put in developing safe code in the classic model, [22, 36, 45, 48, 50, 58]. So, our last question is to determine to which extent we can establish a link between kernel safety and *speculative* kernel safety. Following other works in this direction [17, 57], our main idea is to nullify the gap between *kernel safety* and *speculative kernel* safety, by making the latter property a consequence of the former.

This goal can be achieved by finding a transformation ζ that turns any *kernel safe* system σ into another system $\zeta(\sigma)$ which is architecturally equivalent to σ but enjoys *speculatively kernel safety.* The semantic requirement on the transformation ζ is expressed by Definition 7.1, by which we ask that no user-space program may show different behaviors by executing in the two systems.

Definition 7.1 (Semantics preservation). A system transformation ζ is *user-space semantics preserving* if, for any system $\sigma = (\tau, \gamma, \xi)$,

$$Eval_{\zeta(\sigma),w}(\mathsf{P}, \rho, \mathsf{u}, \tau') \simeq Eval_{\sigma,w}(\mathsf{P}, \rho, \mathsf{u}, \tau)$$

for every layout w, unprivileged command P, and registers ρ. Here, τ' is the store underlying $\zeta(\sigma)$. The equivalence is given by $(v, \tau_1) \simeq (v, \tau_2)$ if $\tau_1 =_{\mathsf{Id_u}} \tau_2$, and coincides with equality otherwise.

Notice that in the previous definition we require $\tau_1 =_{\mathsf{Id_u}} \tau_2$ instead of $\tau_1 = \tau_2$ in order to allow the transformation ζ to modify kernel-space procedures.

Thanks to *semantics preservation*, the second requirement on ζ can be fulfilled by asking that the system $\zeta(\sigma)$ can violate *speculative kernel safety* only if it violates *kernel safety*, as captured by Definition 7.2 below.

Definition 7.2. We say that ζ *imposes speculative kernel safety* if, for every system σ such that $\zeta(\sigma) = (\tau, \gamma, \xi)$, every buffer μ with $\mathrm{dom}(\mu) \subseteq \underline{w}(\mathsf{ArrId})$ and store $\tau' =_{\mathsf{Fun}} \tau$, if

$$w \vdash_{\zeta(\sigma)} ((\gamma(\mathsf{s}), \rho, \mathsf{k_s}), \mu(w \diamond \tau'), b_{ms}) \xrightarrow[D]{O}\!\!\triangleright^{*} \mathsf{unsafe},$$

then

$$w \vdash_{\zeta(\sigma)} ((\gamma(\mathsf{s}), \rho, \mathsf{k_s}), \overline{\mu(w \diamond \tau')}) \to^{*} \mathsf{unsafe}.$$

Observe that, by means of this property, we can easily show that if a system $\zeta(\sigma)$ is *kernel safe*, then it is also *speculative kernel safe.* Finally, by combining Definition 7.2 and Definition 7.1, we obtain the following conclusion:

Proposition 7.3. *If a system σ is* kernel-safe*, and the transformation ζ (i)* imposes speculative kernel safety *and (ii) is* user-space semantics preserving*, then (i) $\zeta(\sigma)$ is* speculative kernel safe*, and (ii) $\zeta(\sigma)$ is semantically equivalent to σ.*

This result states that every *kernel safe* system can be transformed into another system that is equivalent to it from the user's perspective and enjoys stronger security guarantees. Notice that *kernel safety* cannot be provided solely by the adoption of layout randomization: by Theorem 5.3, we know that layout randomization provides *kernel safety* only modulo a small probability of failure.

Just as an example, we observe that a simple transformation that satisfies the requirements of Proposition 7.3 can be implemented by placing a `fence` instruction before *all* the potentially unsafe operations. This instrumentation stops any ongoing speculation before executing potentially unsafe operations, and prevents their transient execution, yet leaving the program's semantics unaltered at the architectural level. This *fencing* transformation is expressed by $\eta : \mathsf{Instr} \longrightarrow \mathsf{P}$ on the level of instructions where, in particular:

$$\begin{aligned}
\eta({*}\mathsf{E} := \mathsf{F}) &\triangleq \mathtt{fence}; {*}\mathsf{E} := \mathsf{F} \\
\eta(x \ := {*}\mathsf{E}) &\triangleq \mathtt{fence}; x \ := {*}\mathsf{E} \\
\eta(\mathtt{call}\ \mathsf{E}(\mathsf{F}_1, \ldots, \mathsf{F}_k)) &\triangleq \mathtt{fence}; \mathtt{call}\ \mathsf{E}(\mathsf{F}_1, \ldots, \mathsf{F}_k) \\
\eta(\mathtt{while}\ \mathsf{E}\ \mathtt{do}\ \mathsf{P}\ \mathtt{od}) &\triangleq \mathtt{while}\ \mathsf{E}\ \mathtt{do}\ \eta(\mathsf{P})\ \mathtt{od} \\
\eta(\mathtt{if}\ \mathsf{E}\ \mathtt{then}\ \mathsf{P}\ \mathtt{else}\ \mathsf{Q}\ \mathtt{fi}) &\triangleq \mathtt{if}\ \mathsf{E}\ \mathtt{then}\ \eta(\mathsf{P})\ \mathtt{else}\ \eta(\mathsf{Q})\ \mathtt{fi},
\end{aligned}$$

and it is the identity on the remaining instructions. Here, the transformation is extended homomorphically to a transformation $\eta : \mathsf{P} \longrightarrow \mathsf{P}$. It is lifted to a system σ by systematically applying it to system calls and procedures $\tau(\mathsf{f})$ in kernel-memory ($\mathsf{f} \in \mathsf{FunId_k}$).

Notice that the transformation η, does not stop completely speculation as, for instance, speculation on conditional instructions is still allowed. This is not in contrast with Definition 7.2 because even in transient execution, a conditional instruction cannot perform any safety violation. However, as their branches can contain unsafe operations, the transformation visits them.

By observing that η enjoys both the properties in Definitions 7.1 and 7.2, we can draw the following conclusion:

Theorem 7.4. *If a system σ is* kernel-safe*, then $\eta(\sigma)$ is* speculative kernel safe*, and $\eta(\sigma)$ is semantically equivalent to σ.*

In addition to η, other program transformations that fit the requirements of Proposition 7.3 can be identified: for instance, the variation of η' that places a single `fence` instruction before sequences of loads — and of stores — is a good choice. Similarly, `fence` instructions can be omitted before direct calls. Finally, in presence of an external proof that shows that a system call s enjoys *speculative kernel safety*, the instrumentation may decide to leave that system call unchanged, yet preserving Definition 7.2.

8 RELATED WORK

On Layout Randomization. The first work that provided a formal account of layout randomization was by Abadi and Plotkin [4], later extended in [2, 3]. In these works, the authors show that layout randomization prevents, with high probability, malicious programs from accessing the memory of a victim in an execution context with shared address space. We have already discussed this in the body of the paper how these results do not model speculative execution or side-channel observations.

Spatial Memory Safety and Non-Interference. Spatial memory safety is typically defined by associating a software component with a memory area and requiring that, at runtime, it only accesses that area [10, 44, 46]. Azevedo de Amorim et al. [7] demonstrated that memory safety can be expressed in terms of non-interference; this property, in turn, stipulates that the final output of a computation is not influenced by secret data that a program must keep confidential [23]. Both of these properties have been extended to the speculative model. The definition of *speculative memory safety* from [10] closely aligns with ours, while *speculative non-interference* was initially introduced in the context of the SPECTECTOR symbolic analyzer [27]. SPECTECTOR's property captures information flows to side-channels that occur with speculative execution but not in sequential execution. In contrast to SPECTECTOR's approach, our definition aligns with *speculative constant-time* [14], as it specifically targets information leaks that occur with speculative semantics.

Formal Analysis of Security Properties of Privileged Execution Environments. Barthe et al. [9] deploy a model with side-channel leaks and privileged execution mode, without specualtive execution. In particular, they are interested in studying the preservation of constant-time in virtualization platforms. They also model privilege-raising procedures *hypercalls*, similar to our system calls. They show that if one of the hosts is constant-time then the system enjoys a form of non-interference with respect to that host's secret memory. For this reason, although the two models are similar, the purposes of Barthe et al. [9] and our work are different: in [9] the victim and the attacker have the same levels of privilege and the role of the hypervisor is to ensure their separation whilst, in our work, the privileged code base is itself the victim.

Attacks to Kernel Layout Randomization. Attacks that aim at leaking information on the kernel's layout are very popular and can rely on implementation bugs that reveal information the kernel's layout [15, 34, 40] or on side-channel info-leaks [26, 35, 38, 38, 39]. In particular attacks such as EchoLoad, TagBleed and EntryBleed [12, 35, 39] are successful even in presence of state-of-art mitigations such as Intel's Page Table Isolation (PTI) [32]. These attacks motivate our decision to take into account side-channel info-leaks. Due to address-space separation between kernel and user space programs, an attacker cannot easily use a pointer to a kernel address to access the victim's memory. So, in general, if the attacker does not control the value of a pointer that is used by the victim, this kind of leak is not harmful.

The Meltdown attack [37] uses speculative execution to overcome this limitation on operating systems running on Intel processors that do not adopt KAISER [25] or PTI [32]. In particular, the hardware can speculatively access an address before checking its permissions. The attack uses this small time window to access kernel memory content and leak it by using a side-channel info-leak gadget. These attacks can also be used to leak information on the layout: by dereferencing pointers under transient execution, the whole kernel's address space can be brute-forced without crashing the system. Due to the adoption of PTI [32], this kind of attack is mitigated by removing most of the kernel-space addresses from the page tables of user-space programs. The BlindSide attack [24] overcomes this issue by probing directly from kernel-space. Similar attacks can be mounted by triggering different forms of mispredictions [8, 41]. Arm's Pointer Authentication [51] is a technique that prevents forging pointers by extending them with an authentication code and raising an error if the code is violated. This can be used to deploy protections similar to layout randomization, but Ravichandran et al. [49] showed that by leveraging speculative execution, it is possible to brute-force the authentication codes.

Relation Between Security in the Speculative- and Classic-model. Blade [57] is a protection mechanism which is aimed at preventing speculative data-flows by selectively stopping speculations. The authors show that, with this mechanism, all those program that are constant-time in the sequential model, are constant-time in the speculative model too. This is similar to what we do in Section 7, by imposing *speculative kernel safety* on a system that enjoys *kernel safety*. PROSPECT [17] is an open-source RISC-V processor that ensures a similar guarantee: each program that is constant-time in the classical model remains *constant-time* even when executed on that processor. This protection relies on taint-tracking and requires explicit annotations on the security level of programs' data.

Protections against Speculative Data Leaks. Commonly, speculative attacks are aimed at leaking its victim's secret data [10, 14, 17, 27, 28, 33, 37, 57]. As a consequence, many of the conventional mitigations against speculative attacks are aimed at preventing secret data from leaking during speculative execution. For instance, Speculative Load Hardening [13] is a software protection measure which, in its simplest form, sets each value that is loaded from memory during transient execution to a constant value. By doing so, this mechanism does not prevent these value to be loaded — i.e. its application to the kernel would not prevent the attacker from breaking speculative kernel safety. Together with the above-mentioned PROSPECT, other hardware-level taint-tracking based mechanisms have been deployed to prevent speculative leaks [59, 60]. These mechanisms limit the speculative execution of load instructions with different levels of strictness, ranging from completely preventing the execution of these instructions ([59], strict propagation and load restriction mode), from just prohibiting the propagation of the loaded value [60]. Although this approach is promising, the above-mentioned mechanisms do not impose limitations on the speculative execution of indirect branches that may be used by attackers in practice to break speculative kernel safety.

9 CONCLUSION

We have formally demonstrated that kernel's layout randomization ensures kernel safety modulo a small probability of failure in a classic threat model. Specifically, in this model attackers cannot compromise the system via speculative execution or side-channels, and users of an operating system execute without privileges, but victims can feature pointer arithmetic, introspection, and indirect jumps.

We have also shown that the protection offered by layout randomization does not naturally scale against attackers that can control speculative execution and side-channels, and stipulate a sufficient condition to enforce kernel safety in the Spectre era. Furthermore, we propose mechanisms based on program transformations that provably enforce speculative kernel safety on a system, provided that this system already enjoys kernel safety in the classic model. To the best of our knowledge, our work is the first to formally investigate and provide ways to achieve kernel safety in the presence of speculative and side-channel vulnerabilities.

This work prepares the ground for future developments such as modeling more expressive attacker models, e.g. attackers speculating on the branch target buffer, related to Spectre v2 [33], optimizing the instrumentation we presented in Section 7, and assessing its overhead on a real operating system.

An orthogonal research direction is to study more fine-grained safety properties instead of *(speculative) kernel safety* — e.g. by distinguishing violations of CFI from violations of spatial safety, and data integrity from confidentiality, akin to what happens in [6]. In this direction, it would also be interesting to model a call stack in order to determine whether the safety of kernel's stack can be granted under other, possibly weaker, conditions.

Finally, a valuable future development is to extend our execution model with other features that are often used to undermine operating systems' security, such as dynamic memory allocation and dynamic module loading, with the aim to establish their impact on system's safety in presence of speculative attackers.

ACKNOWLEDGMENTS

We are grateful to Gilles Barthe, Márton Bognár, Ugo Dal Lago, Lesly-Ann Daniel, Benjamin Grégoire, Jean-Pierre Lozi, and Frank Piessens for their comments on an early draft of the paper. This work was partially supported through the projects PPS ANR-19-C48-0014 and UCA DS4H ANR-17-EURE-0004, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Martín Abadi and Jérémy Planul. 2013. On Layout Randomization for Arrays and Functions. In *Principles of Security and Trust*, David Basin and John C. Mitchell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–185.

[3] Martín Abadi, Jérémy Planul, and Gordon D. Plotkin. 2014. *Layout Randomization and Nondeterminism*. Springer International Publishing, Cham, 1–39. https://doi.org/10.1007/978-3-319-06880-0_1

[4] Martín Abadi and Gordon D. Plotkin. 2012. On Protection by Layout Randomization. *ACM Trans. Inf. Syst. Secur.* 15, 2, Article 8 (jul 2012), 29 pages. https://doi.org/10.1145/2240276.2240279

[5] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. Enforcing Fine-grained Constant-time Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 83–96. https://doi.org/10.1145/3548606.3560689

[6] Sean Noble Anderson, Roberto Blanco, Leonidas Lampropoulos, Benjamin C. Pierce, and Andrew Tolmach. 2023. Formalizing Stack Safety as a Security Property. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE, New York, NY, USA, 356–371. https://doi.org/10.1109/CSF57540.2023.00037

[7] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 79–105.

[8] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 971–988. https://www.usenix.org/conference/usenixsecurity22/presentation/barberis

[9] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-Level Non-Interference for Constant-Time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1267–1279. https://doi.org/10.1145/2660267.2660283

[10] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 1884–1901. https://doi.org/10.1109/SP40001.2021.00046

[11] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[12] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan) *(ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 481–493. https://doi.org/10.1145/3320269.3384747

[13] Chandler Carruth. 2018. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html

[14] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 913–926. https://doi.org/10.1145/3385412.3385970

[15] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A Systematic Study of Elastic Objects in Kernel Exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1165–1184. https://doi.org/10.1145/3372297.3423353

[16] Jonathan Corbet. 2012. Supervisor mode access prevention. https://lwn.net/Articles/517475/

[17] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7161–7178. https://www.usenix.org/conference/usenixsecurity23/presentation/daniel

[18] Theo de Raadt. 2017. OpenBSD 6.3. https://www.openbsd.org/33.html

[19] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/

[20] Stephen Fischer. 2011. Supervisor Mode Execution Protection. https://www.ncsi.com/nsatc11/presentations/wednesday/emerging_technologies/fischer.pdf

[21] Thomas Garnier. 2016. Randomizing the Linux kernel heap freelists. https://mxatone.medium.com/randomizing-the-linux-kernel-heap-freelists-b899bb99c767

[22] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, New York, NY, USA, 179–194. https://doi.org/10.1109/EuroSP.2016.24

[23] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, New York, NY, USA, 11 pages. https://doi.org/10.1109/SP.1982.10014

[24] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1871–1885. https://doi.org/10.1145/3372297.3417289

[25] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham, 161–176.

[26] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 368–379. https://doi.org/10.1145/2976749.2978356

[27] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 1–19. https://doi.org/10.1109/SP40000.2020.00011

[28] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

[29] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, New York, NY, USA, 191–205. https://doi.org/10.1109/SP.2013.23

[30] Apple Inc. 2011. Mac OS X has you Covered. http://www.apple.com/macosx/security/

[31] Intel Corporation 2023. *Intel ®64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

[32] The kernel development community. 2023. Page Table Isolation (PTI). https://www.kernel.org/doc/html/next/x86/pti.html

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, New York, NY, USA, 1–19. https://doi.org/10.1109/SP.2019.00002

[34] Jakob Koschel, Pietro Borrello, Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. 2023. Uncontained: Uncovering Container Confusion in the Linux Kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5055–5072. https://www.usenix.org/conference/usenixsecurity23/presentation/koschel

[35] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, New York, NY, USA, 309–321. https://doi.org/10.1109/EuroSP48549.2020.00027

[36] Jinku Li, Zhi Wang, Tyler Bletsch, Deepa Srinivasan, Michael Grace, and Xuxian Jiang. 2011. Comprehensive and Efficient Protection of Kernel Control Data. *IEEE Transactions on Information Forensics and Security* 6, 4 (2011), 1404–1417. https://doi.org/10.1109/TIFS.2011.2159712

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[39] William Liu, Joseph Ravichandran, and Mengjia Yan. 2023. EntryBleed: A Universal KASLR Bypass against KPTI on Linux. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy* (<conf-loc>, <city>Toronto</city>, <country>Canada</country>, </conf-loc>) *(HASP '23)*. Association for Computing Machinery, New York, NY, USA, 10–18. https://doi.org/10.1145/3623652.3623669

[40] Ziqin Liu, Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Yalong Zou, Dongliang Mu, and Xinyu Xing. 2023. Towards Unveiling Exploitation Potential With Multiple Error Behaviors for Kernel Bugs. *IEEE Transactions on Dependable and Secure Computing* 21, 1 (2023), 1–18. https://doi.org/10.1109/TDSC.2023.3246170

[41] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus. 2021. Bypassing memory safety mechanisms through speculative control flow hijacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 633–649. https://doi.org/10.1109/EuroSP51992.2021.00048

[42] Tarjei Mandt. 2013. Attacking the iOS Kernel: A Look at 'evasi0n'. https://papers.put.as/papers/ios/2013/NISlecture201303.pdf

[43] Ed Maste. 2023. Address Space Layout Randomization (ASLR). https://wiki.freebsd.org/AddressSpaceLayoutRandomization

[44] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code, In Proceedings of the 50th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. *Proc. ACM Program. Lang.* 7, Article 15, 30 pages. https://doi.org/10.1145/3571208

[45] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. DROP THE ROP Fine-grained Control-flow Integrity for the Linux Kernel.

[46] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. *SIGPLAN Not.* 44, 6 (jun 2009), 245–258. https://doi.org/10.1145/1543135.1542504

[47] Android Open Source Project. 2022. Kernel Hardening. https://source.android.com/docs/core/architecture/kernel/hardening

[48] Liam Proven. 2022. Linux 6.1: Rust to hit mainline kernel. https://www.theregister.com/2022/10/05/rust_kernel_pull_request_pulled/

[49] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 685–698. https://doi.org/10.1145/3470496.3527429

[50] Elena Reshetova, Hans Liljestrand, Andrew Paverd, and N Asokan. 2018. Toward Linux kernel memory safety. *Software: Practice and Experience* 48, 12 (2018), 2237–2256.

[51] Mark Rutland. 2017. ARMv8. 3 Pointer Authentication.

[52] Michael S and Vitaly Nikolenko. 2022. Linux kernel heap feng shui in 2022. https://duasynt.com/blog/linux-kernel-heap-feng-shui-2022

[53] SecurityScorecard. 2022. Threat overview for Linux Kernel. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html

[54] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington DC, USA) *(CCS '04)*. Association for Computing Machinery, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[55] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall Press, USA.

[56] PaX Team. 2003. Documentation for the PaX project. https://pax.grsecurity.net/docs/

[57] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL, Article 49 (jan 2021), 30 pages. https://doi.org/10.1145/3434330

[58] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *2010 IEEE Symposium on Security and Privacy*. IEEE, New York, NY, USA, 380–395. https://doi.org/10.1109/SP.2010.30

[59] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 572–586. https://doi.org/10.1145/3352460.3358306

[60] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 954–968. https://doi.org/10.1145/3352460.3358274