On Kernel Safety and Speculative Execution

Work in progress

Davide Davoli[1,2] Tamara Rezk [2] Martin Avanzini[2]

[1]Université Côte d'Azur

[2]INRIA

3rd April 2024 – Annual Meeting of the WG "Formal Methods in Security"

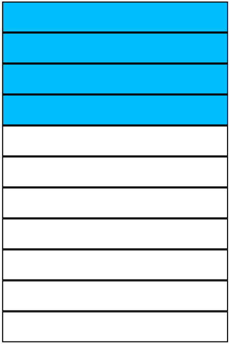
Address Space Layout Randomization

Layout randomization is a software mechanism to enforce *memory safety* and *control flow integrity*.

Address Space Layout Randomization

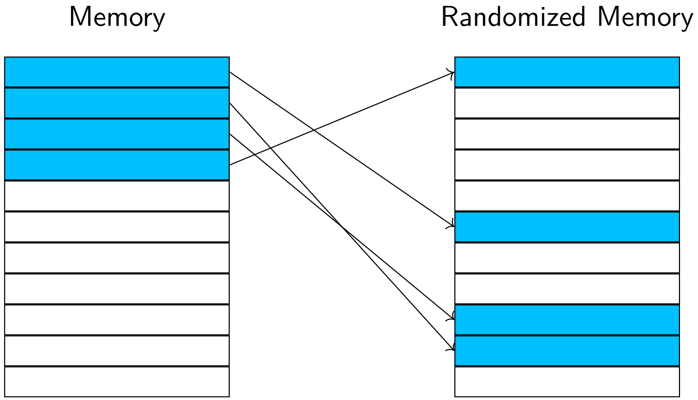
Layout randomization is a software mechanism to enforce *memory safety* and *control flow integrity*.

Memory



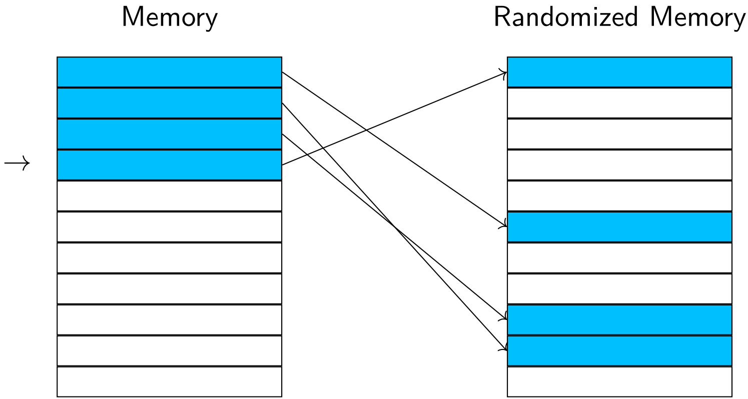
Address Space Layout Randomization

Layout randomization is a software mechanism to enforce *memory safety* and *control flow integrity*.



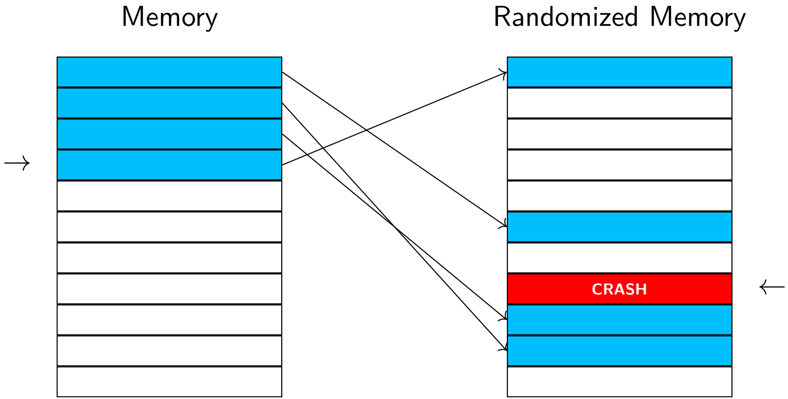
Address Space Layout Randomization

Layout randomization is a software mechanism to enforce *memory safety* and *control flow integrity*.



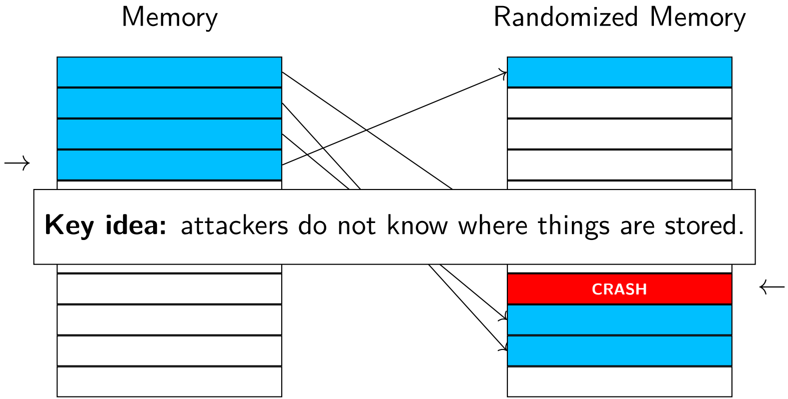
Address Space Layout Randomization

Layout randomization is a software mechanism to enforce *memory safety* and *control flow integrity*.



Address Space Layout Randomization

Layout randomization is a software mechanism to enforce *memory safety* and *control flow integrity*.



Key idea: attackers do not know where things are stored.

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on:

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels, speculative execution.

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels, speculative execution.

Research questions *and contributions*:

- Is KASLR effective without side-channels and speculative execution?

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels, speculative execution.

Research questions *and contributions*:

- Is KASLR effective without side-channels and speculative execution? *Yes.*

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels, speculative execution.

Research questions *and contributions*:

- Is KASLR effective without side-channels and speculative execution? *Yes.*
- Is it effective against side-channels and speculative execution?

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels, speculative execution.

Research questions *and contributions*:

- Is KASLR effective without side-channels and speculative execution? *Yes.*
- Is it effective against side-channels and speculative execution? *Not really...*

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels, speculative execution.

Research questions *and contributions*:

- Is KASLR effective without side-channels and speculative execution? *Yes.*
- Is it effective against side-channels and speculative execution? *Not really*...
- Can we do something else?

Open Problems, Research Questions and Contributions

Open Problems:

- No formal study of Layout Randomization for kernel memory (KASLR).
- Attacks to KASLR based on: side-channels, speculative execution.

Research questions *and contributions*:

- Is KASLR effective without side-channels and speculative execution? *Yes.*
- Is it effective against side-channels and speculative execution? *Not really*...
- Can we do something else? *Yes*.

Kernel's Execution Model

- Privilege level: user or kernel

Kernel's Execution Model

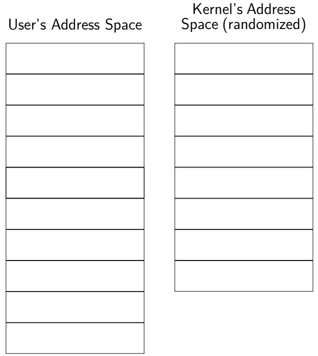
- Privilege level: user or kernel
- Syscalls change privilege level

Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces

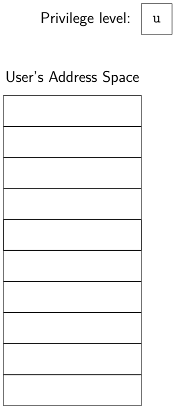
Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces



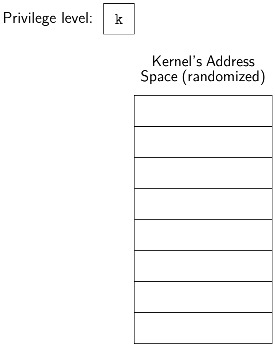
Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces



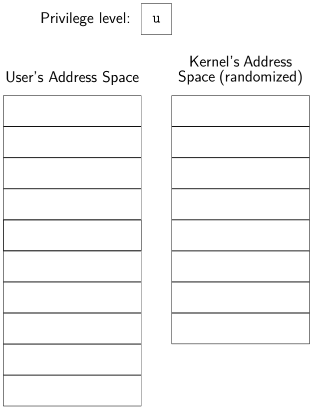
Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces



Kernel's Execution Model

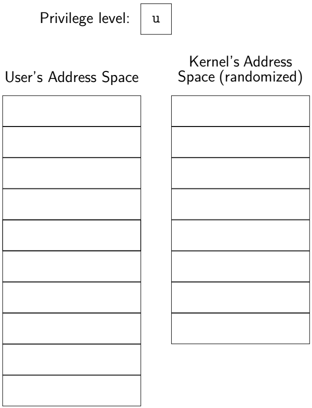
- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces
- Attacker: *user-space* program.



Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces
- Attacker: *user-space* program.

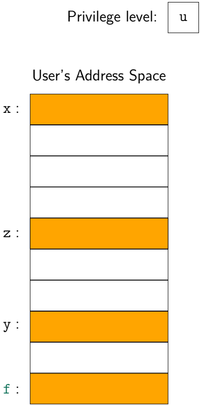
```
void A (void){
  x = get_uid();   //syscall
  y = f(x);        //ordinary call
  y = y + z;
  print(y);        //syscall
}
```



Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces
- Attacker: *user-space* program.

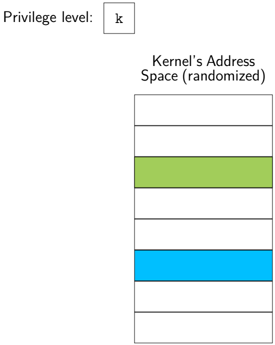
```
void A (void){
  x = get_uid();   //syscall
  y = f(x);        //ordinary call
  y = y + z;
  print(y);        //syscall
}
```



Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces
- Attacker: *user-space* program.

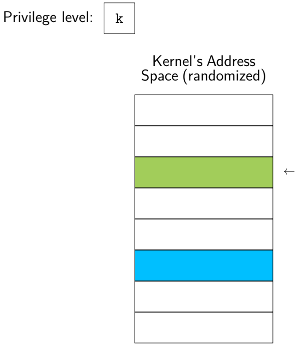
```
void A (void){
  x = get_uid()   //syscall
  y = f(x)        //ordinary call
  y = y + z
  print(y)        //syscall
}
```



Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces
- Attacker: *user-space* program.

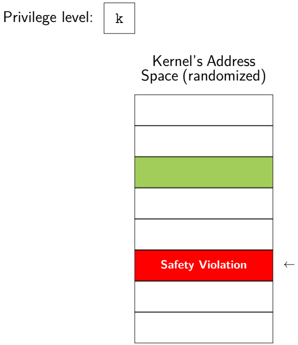
```
void A (void){
  x = get_uid()   //syscall
  y = f(x)        //ordinary call
  y = y + z
  print(y)        //syscall
}
```



Kernel's Execution Model

- Privilege level: user or kernel
- Syscalls change privilege level
- Disjoint address spaces
- Attacker: *user-space* program.

```
void A (void){
  x = get_uid()    //syscall
  y = f(x)         //ordinary call
  y = y + z
  print(y)         //syscall
}
```



Kernel Safety (no side-channels and speculative execution)

For every collection of system calls γ:

$$\text{KASLR} \wedge \text{LNI}(\gamma) \Rightarrow \text{probabilistic safety}$$

Kernel Safety (no side-channels and speculative execution)

For every collection of system calls γ:

$$\mathsf{KASLR} \wedge \mathsf{LNI}(\gamma) \Rightarrow \text{probabilistic safety}$$

$\mathsf{LNI}(\gamma) :=$ the semantics of the syscalls in γ must not depend on the layout

Kernel Safety (no side-channels and speculative execution)

For every collection of system calls γ:

$$\mathsf{KASLR} \wedge \mathsf{LNI}(\gamma) \Rightarrow \text{probabilistic safety}$$

$\mathsf{LNI}(\gamma) :=$ the semantics of the syscalls in γ must not depend on the layout

Victim	LNI
`return p` (*p* array pointer)	no

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

Victim	**Leaked information**

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

Victim	**Leaked information**
`*0xf1a34 = x;`	`data_op 0xf1a34`

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

Victim	**Leaked information**
`*0xf1a34 = x;`	`data_op 0xf1a34`
`x = *0xf1a34;`	`data_op 0xf1a34`

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

Victim	**Leaked information**
$*$`0xf1a34` = `x`;	`data_op 0xf1a34`
`x` = $*$`0xf1a34`;	`data_op 0xf1a34`
($*$`0xf1a34`)();	`jump 0xf1a34`

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

Victim	**Leaked information**
`*0xf1a34 = x;`	`data_op 0xf1a34`
`x = *0xf1a34;`	`data_op 0xf1a34`
`(*0xf1a34)();`	`jump 0xf1a34`

To restore the protection offered by KASLR, LNI needs to be strengthened:

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

Victim	**Leaked information**
`*0xf1a34 = x;`	`data_op 0xf1a34`
`x = *0xf1a34;`	`data_op 0xf1a34`
`(*0xf1a34)();`	`jump 0xf1a34`

To restore the protection offered by KASLR, LNI needs to be strengthened:

LNI(γ) := the semantics of the syscalls in γ must not depend on the layout

KASLR in the presence of *Side-Channel* Attackers

Instructions leak information on the layout.

Victim	**Leaked information**
`*0xf1a34 = x;`	`data_op 0xf1a34`
`x = *0xf1a34;`	`data_op 0xf1a34`
`(*0xf1a34)();`	`jump 0xf1a34`

To restore the protection offered by KASLR, LNI needs to be strengthened:

LNI(γ) := the semantics and the side-channel leaks of the syscalls in γ must not depend on the layout

KASLR in presence of *Speculative* Attackers

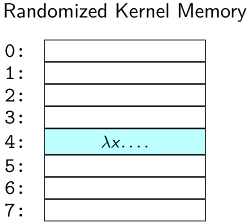
The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

```
void s (x, y){
  if(x)
    (*y)(x);
}
```

KASLR in presence of *Speculative* Attackers

The system call `s` is a threat in presence of speculative attackers (BlindSide), and we can model it.

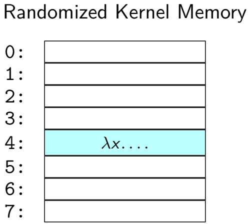
```
for(i = 0; i < 8; i++){
  force(true);
  s(false, i);
}
void s (x, y){
  if(x)
    (*y)(x);
}
```



KASLR in presence of *Speculative* Attackers

The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

```
for(i = 0; i < 8; i++){
  force(true);
  s(false, i);
}
void s (x, y){
  if(x)
    (*y)(x);
}
```

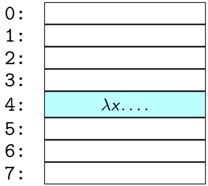


KASLR in presence of *Speculative* Attackers

The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

```
→   for(i = 0; i < 8; i++){
      force(true);
      s(false, i);
    }
    void s (x, y){
      if(x)
        (*y)(x);
    }
```

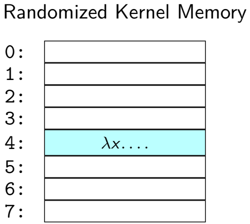
Randomized Kernel Memory



KASLR in presence of *Speculative* Attackers

The system call `s` is a threat in presence of speculative attackers (BlindSide), and we can model it.

```
    for(i = 0; i < 8; i++){
→     force(true);
      s(false, i);
    }
    void s (x, y){
      if(x)
        (*y)(x);
    }
```

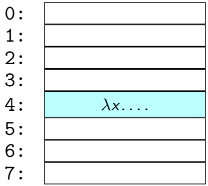


KASLR in presence of *Speculative* Attackers

The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

```
    for(i = 0; i < 8; i++){
        force(true);
        s(false, i);
    }
    void s (x, y){
→       if(x)
            (*y)(x);
    }
```

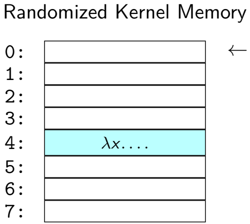
Randomized Kernel Memory



KASLR in presence of *Speculative* Attackers

The system call `s` is a threat in presence of speculative attackers (BlindSide), and we can model it.

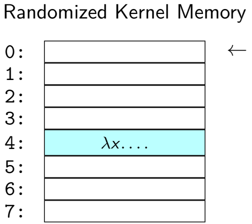
```
    for(i = 0; i < 8; i++){
      force(true);
      s(false, i);
    }
    void s (x, y){
→     if(x)
-->     (*y)(x);
    }
```



KASLR in presence of *Speculative* Attackers

The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

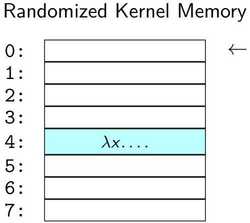
```
for(i = 0; i < 8; i++){
  force(true);
  s(false, i);
}
void s (x, y){
→   if(x)
      (*y)(x);
}
```



KASLR in presence of *Speculative* Attackers

The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

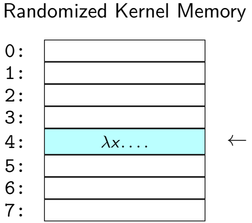
```
    for(i = 0; i < 8; i++){
      force(true);
      s(false, i);
    }
    void s (x, y){
      if(x)
        (*y)(x);
→   }
```



KASLR in presence of *Speculative* Attackers

The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

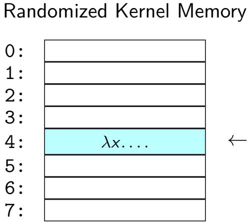
```
→   for(i = 0; i < 8; i++){
      force(true);
      s(false, i);
    }
    void s (x, y){
      if(x)
        (*y)(x);
    }
```



KASLR in presence of *Speculative* Attackers

The system call `s` is a threat in presence of speculative attackers (BlindSide), and we can model it.

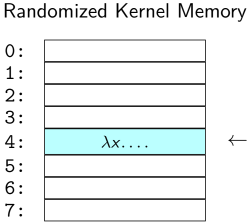
```
   for(i = 0; i < 8; i++){
→      force(true);
       s(false, i);
   }
   void s (x, y){
       if(x)
           (*y)(x);
   }
```



KASLR in presence of *Speculative* Attackers

The system call `s` is a threat in presence of speculative attackers (BlindSide), and we can model it.

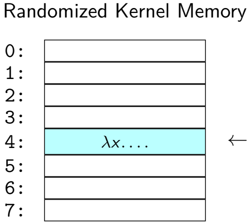
```
for(i = 0; i < 8; i++){
  force(true);
  s(false, i);
}
void s (x, y){
→   if(x)
      (*y)(x);
}
```



KASLR in presence of *Speculative* Attackers

The system call s is a threat in presence of speculative attackers (BlindSide), and we can model it.

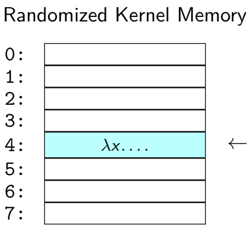
```
   for(i = 0; i < 8; i++){
     force(true);
     s(false, i);
   }
   void s (x, y){
→    if(x)
-->    (*y)(x);
   }
```



KASLR in presence of *Speculative* Attackers

The system call `s` is a threat in presence of speculative attackers (BlindSide), and we can model it.

```
for(i = 0; i < 8; i++){
  force(true);
  s(false, i);
}
void s (x, y){
  if(x)
    (*y)(x);
}
```

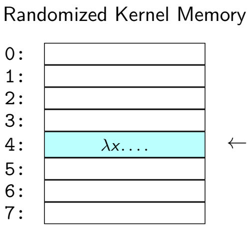


LNI(γ) := the semantics the side channel leaks of the syscalls in γ must not depend on the layout

KASLR in presence of *Speculative* Attackers

The system call `s` is a threat in presence of speculative attackers (BlindSide), and we can model it.

```
for(i = 0; i < 8; i++){
  force(true);
  s(false, i);
}
void s (x, y){
  if(x)
    (*y)(x);
}
```



SLNI(γ) := the speculative semantics and the side channel leaks of the syscalls in γ must not depend on the layout

KASLR with Speculative Attackers and Side-Channels

$$\mathsf{SLNI}(\gamma) \Rightarrow \mathsf{safety}$$

KASLR with Speculative Attackers and Side-Channels

$$\text{SLNI}(\gamma) \Rightarrow \text{safety}$$

$$\Downarrow$$

$$\text{KASLR} \wedge \text{SLNI}(\gamma) \Rightarrow \text{probabilistic safety}$$

KASLR with Speculative Attackers and Side-Channels

$$\text{SLNI}(\gamma) \Rightarrow \text{safety}$$
$$\Downarrow$$
$$\text{KASLR} \wedge \text{SLNI}(\gamma) \Rightarrow \text{probabilistic safety}$$

Victim SLNI

KASLR with Speculative Attackers and Side-Channels

$$\text{SLNI}(\gamma) \Rightarrow \text{safety}$$

$$\Downarrow$$

$$\text{KASLR} \wedge \text{SLNI}(\gamma) \Rightarrow \text{probabilistic safety}$$

Victim	SLNI
$\texttt{p}[0] = 42$ (p array pointer)	no

KASLR with Speculative Attackers and Side-Channels

$$\text{SLNI}(\gamma) \Rightarrow \text{safety}$$
$$\Downarrow$$
$$\text{KASLR} \wedge \text{SLNI}(\gamma) \Rightarrow \text{probabilistic safety}$$

Victim		SLNI
$\texttt{p}[0] = 42$	(p array pointer)	no
$\texttt{f}()$	($\texttt{f}$ procedure)	no

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (1/3)

γ is safe against speculative attackers

$\Downarrow$

γ is safe against ordinary attackers

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (1/3)

γ is safe against speculative attackers

$\Uparrow$?

γ is safe against ordinary attackers

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (1/3)

γ is safe against speculative attackers

$\not\Updownarrow$

γ is safe against ordinary attackers

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (1/3)

$$\gamma \text{ is safe against speculative attackers}$$
$$\not\Uparrow$$
$$\gamma \text{ is safe against ordinary attackers}$$

But maybe there is an instrumentation ζ such that:

$$\gamma \text{ is safe against ordinary attackers}$$
$$\Downarrow$$
$$\zeta(\gamma) \text{ is safe against } \textit{speculative} \text{ attackers}$$

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (2/3)

Theorem

If ζ*:*

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (2/3)

Theorem

If ζ*:*

- *preserves the semantics of the syscalls,*

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (2/3)

Theorem

If ζ*:*

- *preserves the semantics of the syscalls,*
- *prevents the transient execution of unsafe commands,*

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (2/3)

Theorem

If ζ*:*

- *preserves the semantics of the syscalls,*
- *prevents the transient execution of unsafe commands,*

and γ *is safe against ordinary attackers, then* $\zeta(\gamma)$ *is safe against speculative attackers.*

Kernel Safety ⇒ *Speculative* Kernel Safety (3/3)

Does such transformation exist?

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (3/3)

Does such transformation exist? Yes.

Kernel Safety $\Rightarrow$ *Speculative* Kernel Safety (3/3)

Does such transformation exist? Yes.

$$\zeta(\texttt{if(E) \{C\} else \{D\}}) \triangleq \texttt{if(E)}\ \{\zeta(\texttt{C})\}\ \texttt{else}\ \{\zeta(\texttt{D})\}$$

$$\zeta(\texttt{while(E) \{C\}}) \triangleq \texttt{while(E)}\ \{\zeta(\texttt{C})\}$$

$$\zeta(*\texttt{E} = \texttt{F}) \triangleq \texttt{lfence}; *\texttt{E} = \texttt{F}$$

$$\zeta(\texttt{E} = *\texttt{F}) \triangleq \texttt{lfence}; \texttt{E} = *\texttt{F}$$

$$\zeta((*\texttt{E})(\texttt{F}_1, \ldots, \texttt{F}_k)) \triangleq \texttt{lfence}; (*\texttt{E})(\texttt{F}_1, \ldots, \texttt{F}_k)$$

Conclusions

- Layout Randomization protects the kernel against *ordinary* attacks in presence of LNI,

Conclusions

- Layout Randomization protects the kernel against *ordinary* attacks in presence of LNI, but not against the *speculative* ones.

Conclusions

- Layout Randomization protects the kernel against *ordinary* attacks in presence of LNI, but not against the *speculative* ones.
- If a kernel is safe against ordinary attacks, it is possible to make it safe against speculative attacks.

Conclusions

- Layout Randomization protects the kernel against *ordinary* attacks in presence of LNI, but not against the *speculative* ones.
- If a kernel is safe against ordinary attacks, it is possible to make it safe against speculative attacks.

Future work:

Conclusions

- Layout Randomization protects the kernel against *ordinary* attacks in presence of LNI, but not against the *speculative* ones.
- If a kernel is safe against ordinary attacks, it is possible to make it safe against speculative attacks.

Future work:

- Model indirect branch speculation.

Conclusions

- Layout Randomization protects the kernel against *ordinary* attacks in presence of LNI, but not against the *speculative* ones.
- If a kernel is safe against ordinary attacks, it is possible to make it safe against speculative attacks.

Future work:

- Model indirect branch speculation.
- Evaluate the overhead of our instrumentation in practice.