On Kernel's Safety in the Spectre Era

(And KASLR is Formally Dead)

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

[1]Université Côte d'Azur

[2]Inria

15th October 2024 – ACM Conference on Computer and Communications Security

Contribution of the talk

There is hope of protecting kernels against speculative attacks.

Memory corruption and layout randomization

Layout randomization is meant to contrast *memory corruption*, i.e., when memory can be modified against the programmer's expectations.

Memory corruption and layout randomization

Layout randomization is meant to contrast *memory corruption*, i.e., when memory can be modified against the programmer's expectations.

```
void s(){
    ...;
    (*fp)();
}
```

The attacker 🕵 controls `fp`.

Memory corruption and layout randomization

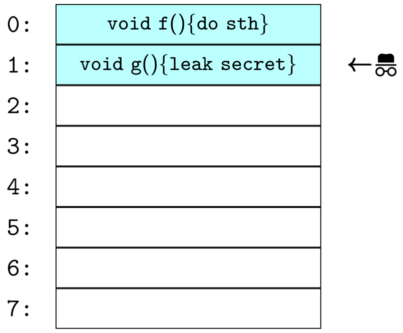
Layout randomization is meant to contrast *memory corruption*, i.e., when memory can be modified against the programmer's expectations.

```
void s(){
    ...;
    (*fp)();
}
```

The attacker 🕵 controls `fp`.

Attack: `fp` $= 1; s()$.

With Deterministic Layout



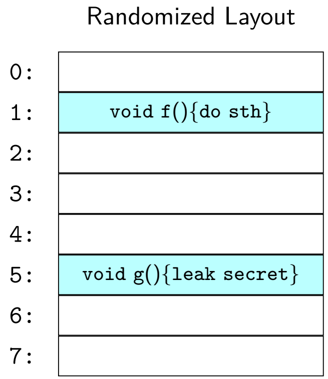
Memory corruption and layout randomization

Layout randomization is meant to contrast *memory corruption*, i.e., when memory can be modified against the programmer's expectations.

```
void s(){
   ...;
   (*fp)();
}
```

The attacker controls fp.

Attack: $\texttt{fp} = ?; s()$.



Memory corruption and layout randomization

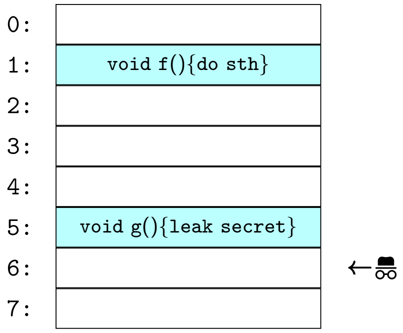
Layout randomization is meant to contrast *memory corruption*, i.e., when memory can be modified against the programmer's expectations.

```
void s(){
    ...;
    (*fp)();
}
```

The attacker 🕵 controls fp.

Attack: fp $= 6$; $s()$.

Randomized Layout



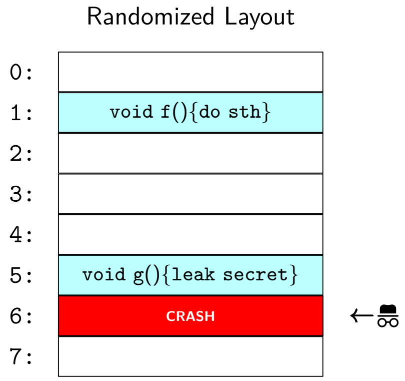
Memory corruption and layout randomization

Layout randomization is meant to contrast *memory corruption*, i.e., when memory can be modified against the programmer's expectations.

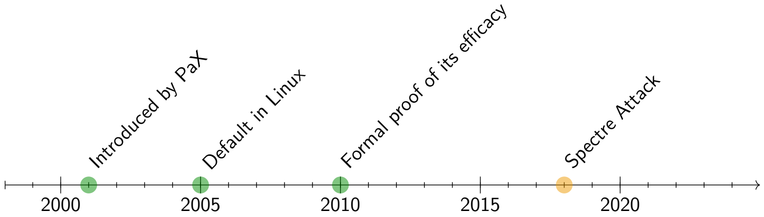
```
void s(){
    ...;
   (*fp)();
}
```

The attacker controls `fp`.

Attack: `fp` $= 6$; $s()$.

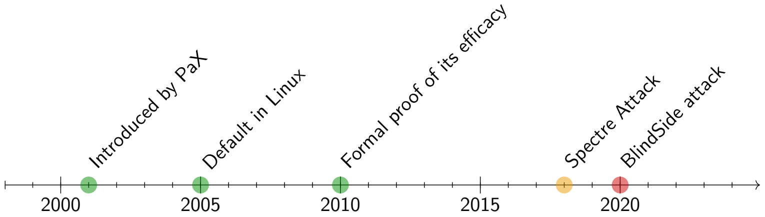


The demise of layout randomization



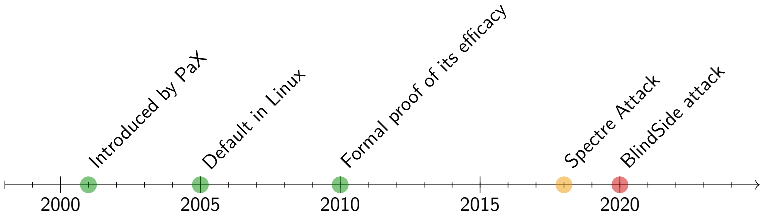
- (Abadi and Plotkin, 2010): the probability of memory corruption is low.

The demise of layout randomization



- (Abadi and Plotkin, 2010): the probability of memory corruption is low.
- (Göktaș et. al., 2020): with speculative execution, layout randomization is broken.

The demise of layout randomization



- (Abadi and Plotkin, 2010): the probability of memory corruption is low.
- (Göktaş et. al., 2020): with speculative execution, layout randomization is broken.

Can we prevent speculative attacks on kernels?

Main contributions

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

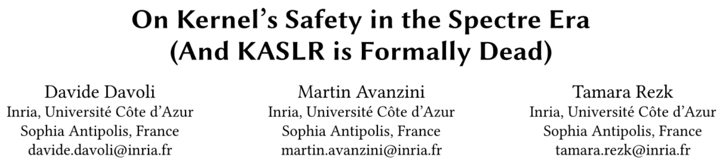
Davide Davoli
Inria, Université Côte d'Azur
Sophia Antipolis, France
davide.davoli@inria.fr

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

- We devise a semantics where side-channel and speculative attacks to kernel's layout randomization can be expressed as programs.

Main contributions



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

- We devise a semantics where side-channel and speculative attacks to kernel's layout randomization can be expressed as programs.
- **If a kernel is safe against *ordinary attacks*, it is possible to protect it against *speculative attacks*, systematically.**

Threat model

Victim:

- Kernel exposing functionalities to user space programs via system calls.
- *Kernel space* memory is not accessible to *user space* programs.

Threat model

Victim:

- Kernel exposing functionalities to user space programs via system calls.
- *Kernel space* memory is not accessible to *user space* programs.

Attacker:

- User space program, interacts with the victim via system calls.

Threat model

Victim:

- Kernel exposing functionalities to user space programs via system calls.
- *Kernel space* memory is not accessible to *user space* programs.

Attacker:

- User space program, interacts with the victim via system calls.
- Access to side-channel leaks.

Threat model

Victim:

- Kernel exposing functionalities to user space programs via system calls.
- *Kernel space* memory is not accessible to *user space* programs.

Attacker:

- User space program, interacts with the victim via system calls.
- Access to side-channel leaks.
- Controls direct branch speculation and store-to-load forwarding.

Threat model

Victim:

- Kernel exposing functionalities to user space programs via system calls.
- *Kernel space* memory is not accessible to *user space* programs.

Attacker:

- User space program, interacts with the victim via system calls.
- Access to side-channel leaks.
- Controls direct branch speculation and store-to-load forwarding.

Attacker's Goal:

- Trigger a system call to execute code or access data that it is not authorized to access.

Speculative code execution, despite layout randomization

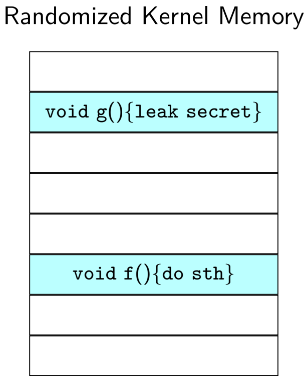
Speculation introduces new vulnerabilities.

Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
void s(fp){             //victim syscall
  if (fp points to f){
    ...;
    (*fp)();
}}
for fp in Addresses { //attack (user space)
  predict(branch true);
  s(fp);
}
```

not corrupted

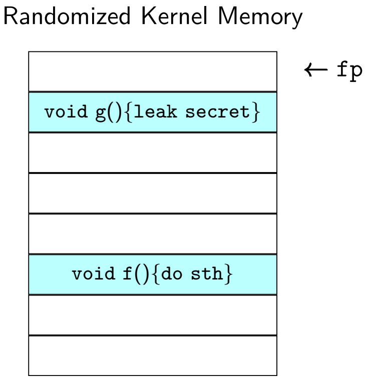


Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
  void s(fp){              //victim syscall
    if (fp points to f){
      ...;
      (*fp)();
  }}
→ for fp in Addresses { //attack (user space)
    predict(branch true);
    s(fp);
  }
```

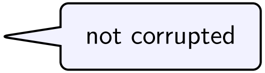
not corrupted

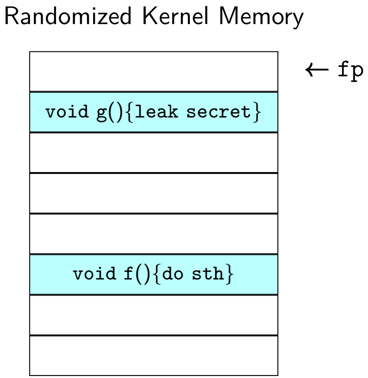


Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
  void s(fp){              //victim syscall
    if (fp points to f){
      ...;
      (*fp)();
  }}
  for fp in Addresses { //attack (user space)
→   predict(branch true);
    s(fp);
  }
```

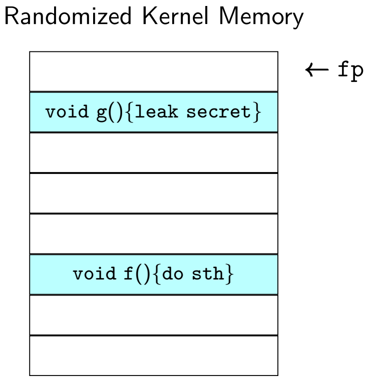




Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
→   void s(fp){              //victim syscall
      if (fp points to f){
        ...;
        (*fp)();   not corrupted
    }}
    for fp in Addresses { //attack (user space)
      predict(branch true);
      s(fp);
    }
```

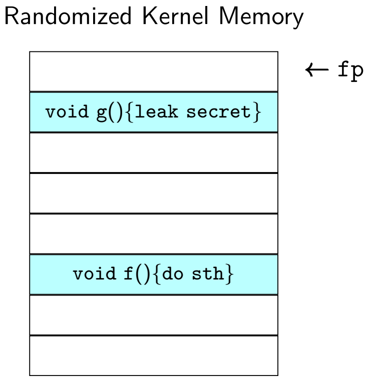


Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
→     void s(fp){              //victim syscall
        if (fp points to f){
          ...;
-->       (*fp)();
      }}
      for fp in Addresses { //attack (user space)
        predict(branch true);
        s(fp);
      }
```

speculatively corrupted

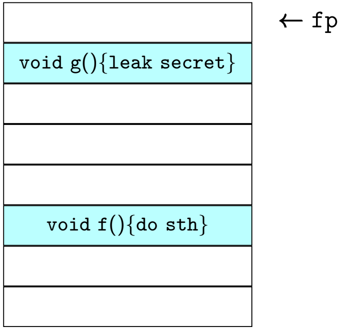


Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
        void s(fp){               //victim syscall
→         if (fp points to f){
            ...;
            (*fp)();
        }}
        for fp in Addresses { //attack (user space)
          predict(branch true);
          s(fp);
        }
```

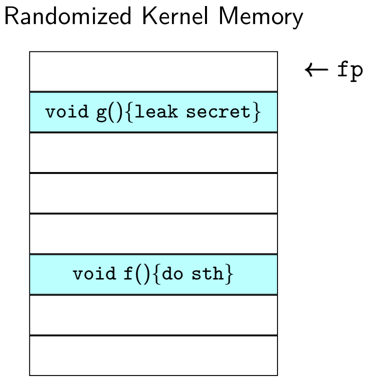
Randomized Kernel Memory



Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
     void s(fp){            //victim syscall
       if (fp points to f){
         ...;
         (*fp)();    not corrupted
→    }}
     for fp in Addresses { //attack (user space)
       predict(branch true);
       s(fp);
     }
```

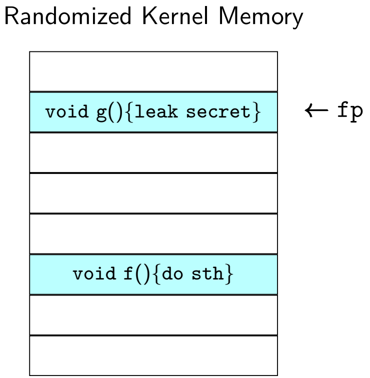


Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
    void s(fp){              //victim syscall
      if (fp points to f){
        ...;
        (*fp)();
    }}
→   for fp in Addresses { //attack (user space)
      predict(branch true);
      s(fp);
    }
```

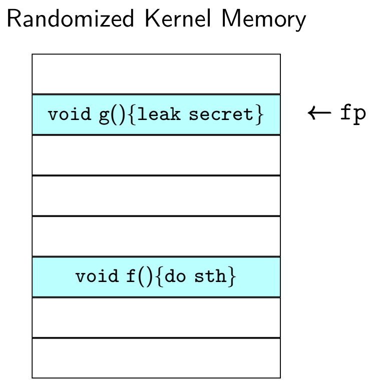
not corrupted



Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

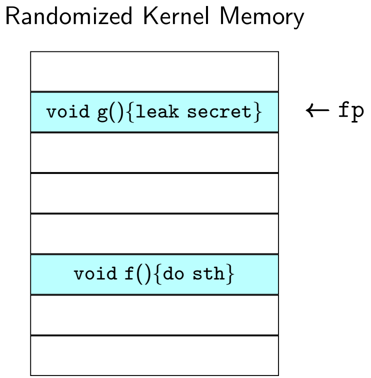
```
  void s(fp){              //victim syscall
    if (fp points to f){
      ...;
      (*fp)();   not corrupted
  }}
  for fp in Addresses { //attack (user space)
→   predict(branch true);
    s(fp);
  }
```



Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
   void s(fp){              //victim syscall
→    if (fp points to f){
        ...;
        (*fp)();     not corrupted
   }}
   for fp in Addresses { //attack (user space)
     predict(branch true);
     s(fp);
   }
```

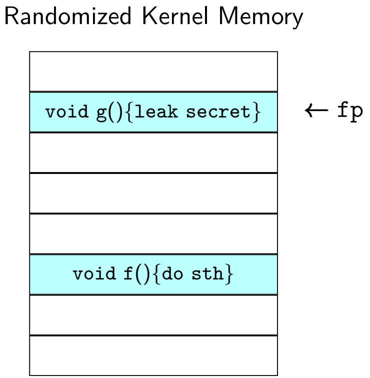


Speculative code execution, despite layout randomization

Speculation introduces new vulnerabilities.

```
→     void s(fp){            //victim syscall
        if (fp points to f){
          ...;
-->       (*fp)();
      }}
      for fp in Addresses { //attack (user space)
        predict(branch true);
        s(fp);
      }
```

speculatively corrupted



Protecting kernels in the presence of speculative execution (1/2)

Speculative execution introduces new behaviors, so *reasoning about speculative attacks is harder*, but:

Protecting kernels in the presence of speculative execution (1/2)

Speculative execution introduces new behaviors, so *reasoning about speculative attacks is harder*, but:

Theorem (Main result)

If a kernel is safe for ordinary attacks,

Protecting kernels in the presence of speculative execution (1/2)

Speculative execution introduces new behaviors, so *reasoning about speculative attacks is harder*, but:

Theorem (Main result)

If a kernel is safe for ordinary attacks, and we transform it in such a way that if a speculative attack can exploit a system call an ordinary attack can also,

Protecting kernels in the presence of speculative execution (1/2)

Speculative execution introduces new behaviors, so *reasoning about speculative attacks is harder*, but:

Theorem (Main result)

If a kernel is safe for ordinary attacks, and we transform it in such a way that if a speculative attack can exploit a system call an ordinary attack can also, then the transformed kernel is safe for speculative attacks.

Protecting kernels in the presence of speculative execution (1/2)

Speculative execution introduces new behaviors, so *reasoning about speculative attacks is harder*, but:

Theorem (Main result)

If a kernel is safe for ordinary attacks, and we transform it in such a way that if a speculative attack can exploit a system call an ordinary attack can also, then the transformed kernel is safe for speculative attacks.

So layout randomization + transformation = no *speculative* attacks?

Protecting kernels in the presence of speculative execution (1/2)

Speculative execution introduces new behaviors, so *reasoning about speculative attacks is harder*, but:

Theorem (Main result)

If a kernel is safe for ordinary attacks, and we transform it in such a way that if a speculative attack can exploit a system call an ordinary attack can also, then the transformed kernel is safe for speculative attacks.

So layout randomization + transformation = no *speculative* attacks? **No, see the paper.**

Protecting kernels in the presence of speculative execution (2/2)

Does such transformation exist?

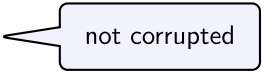
Protecting kernels in the presence of speculative execution (2/2)

Does such transformation exist? **Yes:** *without indirect branch speculation*, it is enough to place an `lfence` before instructions that interact with the memory.

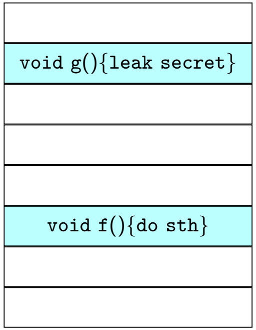
Protecting kernels in the presence of speculative execution (2/2)

Does such transformation exist? **Yes:** *without indirect branch speculation*, it is enough to place an `lfence` before instructions that interact with the memory.

```
void s(fp){          //victim syscall
  if (fp points to f){
    ...;
    lfence;
    (*fp)();
}}
for fp in Addresses { //attack (user space)
  predict(true);
  s(fp);
}
```



Randomized Kernel Memory

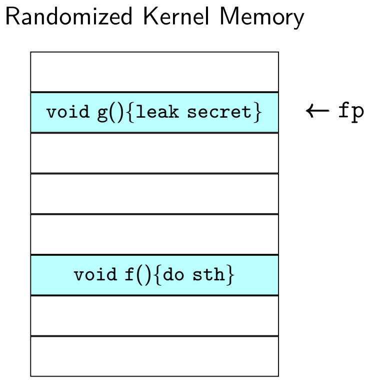


Protecting kernels in the presence of speculative execution (2/2)

Does such transformation exist? **Yes:** *without indirect branch speculation*, it is enough to place an `lfence` before instructions that interact with the memory.

```
→    void s(fp){              //victim syscall
       if (fp points to f){
         ...;
         lfence;
         (*fp)();
     }}
     for fp in Addresses { //attack (user space)
       predict(true);
       s(fp);
     }
```

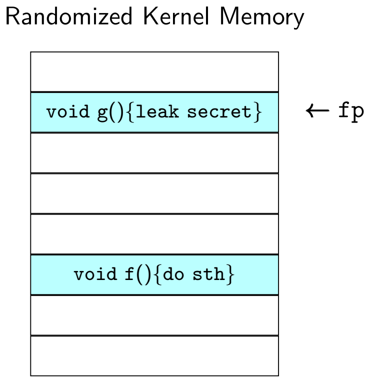
not corrupted



Protecting kernels in the presence of speculative execution (2/2)

Does such transformation exist? **Yes:** *without indirect branch speculation*, it is enough to place an `lfence` before instructions that interact with the memory.

```
  void s(fp){              //victim syscall
→   if (fp points to f){
      ...;
      lfence;
      (*fp)();      not corrupted
  }}
  for fp in Addresses { //attack (user space)
    predict(true);
    s(fp);
  }
```

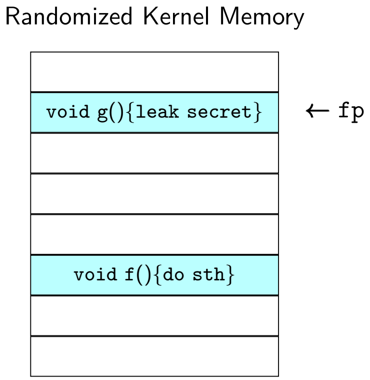


Protecting kernels in the presence of speculative execution (2/2)

Does such transformation exist? **Yes:** *without indirect branch speculation*, it is enough to place an `lfence` before instructions that interact with the memory.

```
      void s(fp){              //victim syscall
→       if (fp points to f){
          ...;
-->       lfence;
          (*fp)();
      }}
      for fp in Addresses { //attack (user space)
        predict(true);
        s(fp);
      }
```

not corrupted



Conclusions and ongoing work

Conclusions:

- There is hope of protecting kernels against speculative attacks.

Conclusions and ongoing work

Conclusions:

- There is hope of protecting kernels against speculative attacks.
- In the paper:
    - semantics modeling speculative attacks,

Conclusions and ongoing work

Conclusions:

- There is hope of protecting kernels against speculative attacks.
- In the paper:
  - semantics modeling speculative attacks,
  - why side-channels are a threat for layout randomization,

Conclusions and ongoing work

Conclusions:

- There is hope of protecting kernels against speculative attacks.
- In the paper:
  - semantics modeling speculative attacks,
  - why side-channels are a threat for layout randomization,
  - code transformation to protect kernels, ...

Conclusions and ongoing work

Conclusions:

- There is hope of protecting kernels against speculative attacks.
- In the paper:
  - semantics modeling speculative attacks,
  - why side-channels are a threat for layout randomization,
  - code transformation to protect kernels, . . .

Ongoing work:

- Model indirect branch speculation.

Conclusions and ongoing work

Conclusions:

- There is hope of protecting kernels against speculative attacks.
- In the paper:
  - semantics modeling speculative attacks,
  - why side-channels are a threat for layout randomization,
  - code transformation to protect kernels, ...

Ongoing work:

- Model indirect branch speculation.
- Implementation of suitable program transformations.

Conclusions and ongoing work

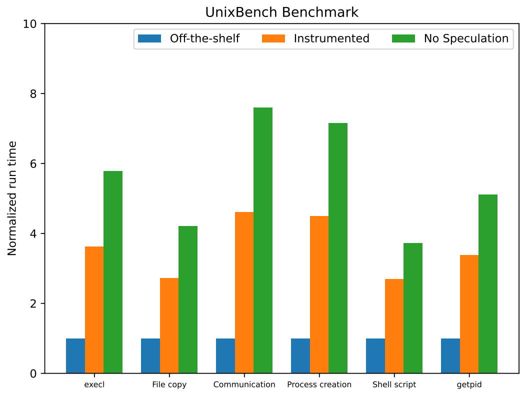
Conclusions:

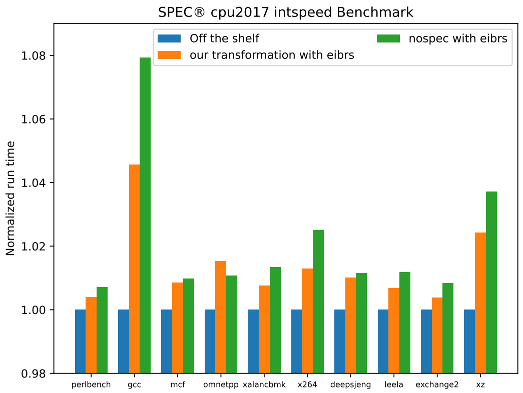
- There is hope of protecting kernels against speculative attacks.
- In the paper:
  - semantics modeling speculative attacks,
  - why side-channels are a threat for layout randomization,
  - code transformation to protect kernels, . . .

Ongoing work:

- Model indirect branch speculation.
- Implementation of suitable program transformations.
- Performance overhead evaluation.

Ongoing work


UnixBench Benchmark
Normalized run time
Off-the-shelf
Instrumented
No Speculation
execl
File copy
Communication
Process creation
Shell script
getpid


SPEC® cpu2017 intspeed Benchmark
Normalized run time
Off the shelf
our transformation with eibrs
nospec with eibrs
perlbench
gcc
mcf
omnetpp
xalancbmk
x264
deepsjeng
leela
exchange2
xz