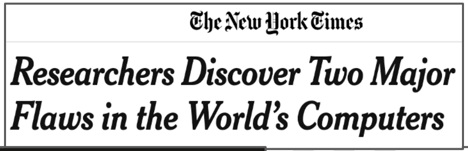
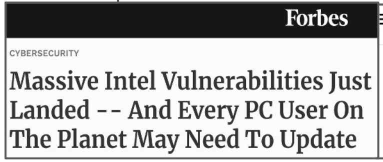
KASPER: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel

Brian Johannesmeyer*, Jakob Koschel*,
Kaveh Razavi, Herbert Bos, Cristiano Giuffrida

*Joint first authors

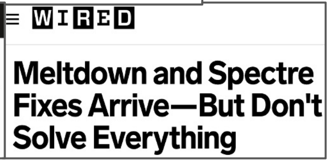

The New York Times

Researchers Discover Two Major Flaws in the World's Computers

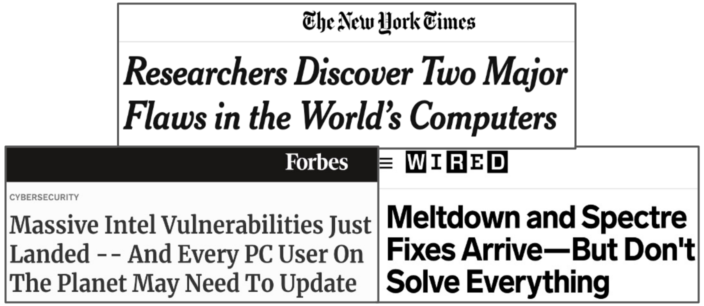

Forbes

CYBERSECURITY

Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update


WIRED

Meltdown and Spectre Fixes Arrive—But Don't Solve Everything



The New York Times

Researchers Discover Two Major Flaws in the World's Computers

Forbes

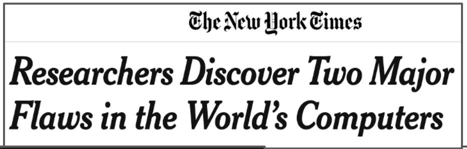
CYBERSECURITY

Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update

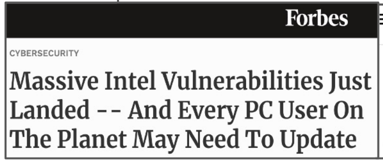
WIRED

Meltdown and Spectre Fixes Arrive—But Don't Solve Everything

SPECTRE

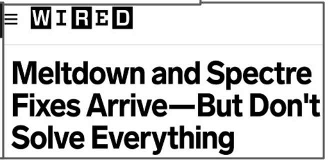

The New York Times

Researchers Discover Two Major Flaws in the World's Computers

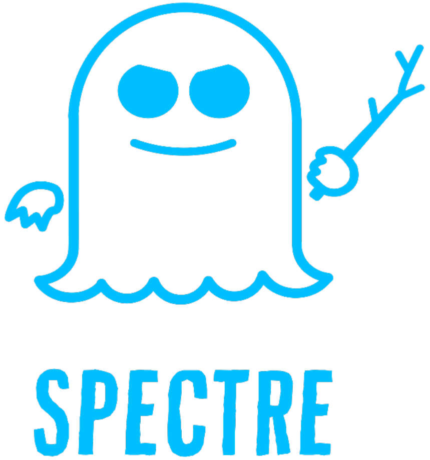

Forbes

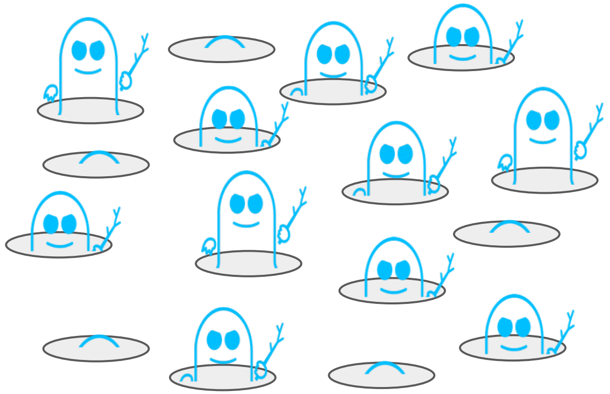
CYBERSECURITY

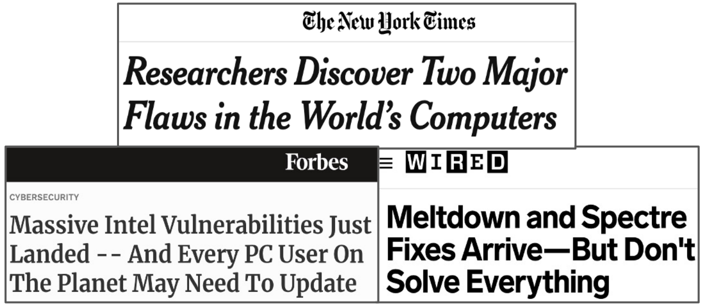
Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update


WIRED

Meltdown and Spectre Fixes Arrive—But Don't Solve Everything






The New York Times

Researchers Discover Two Major Flaws in the World's Computers

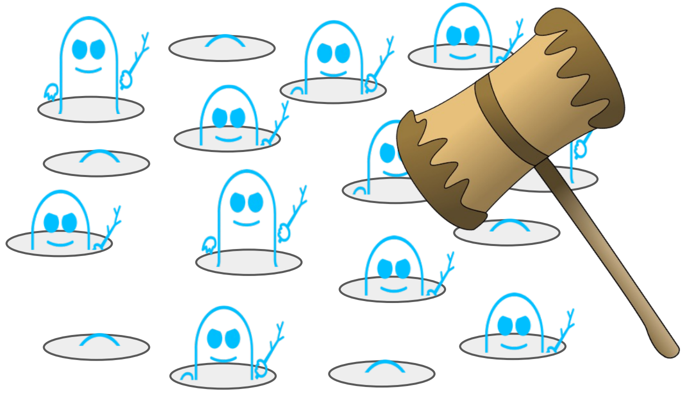
Forbes

CYBERSECURITY

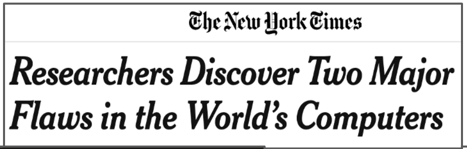
Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update

WIRED

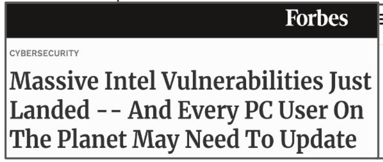
Meltdown and Spectre Fixes Arrive—But Don't Solve Everything



SPECTRE

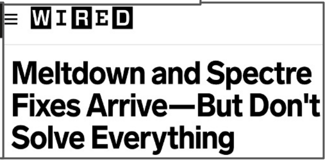

The New York Times

Researchers Discover Two Major Flaws in the World's Computers

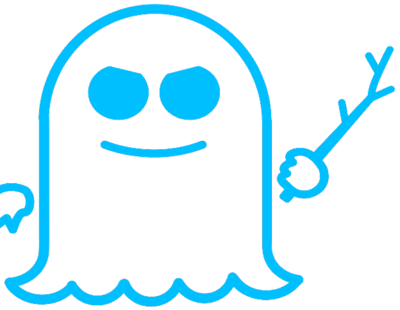

Forbes

CYBERSECURITY

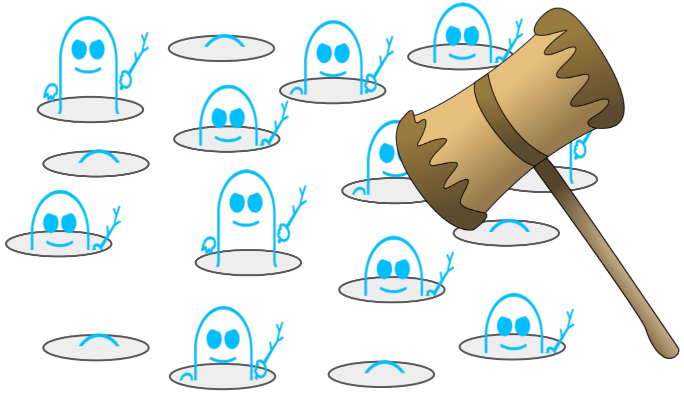
Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update

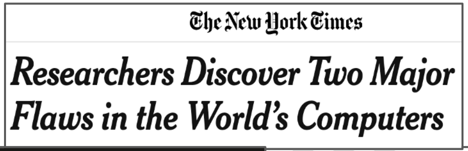

WIRED

Meltdown and Spectre Fixes Arrive—But Don't Solve Everything



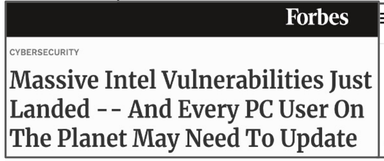
SPECTRE





The New York Times

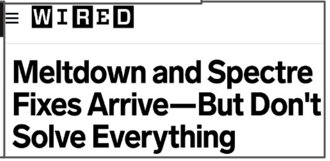
Researchers Discover Two Major Flaws in the World's Computers



Forbes

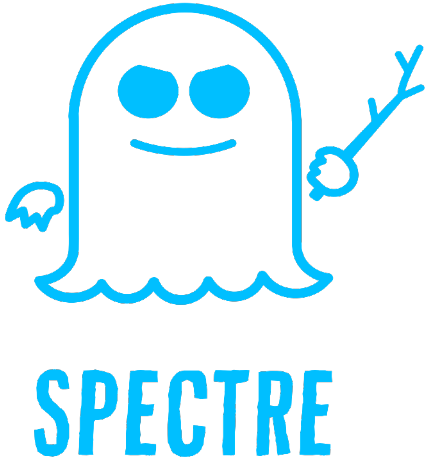
CYBERSECURITY

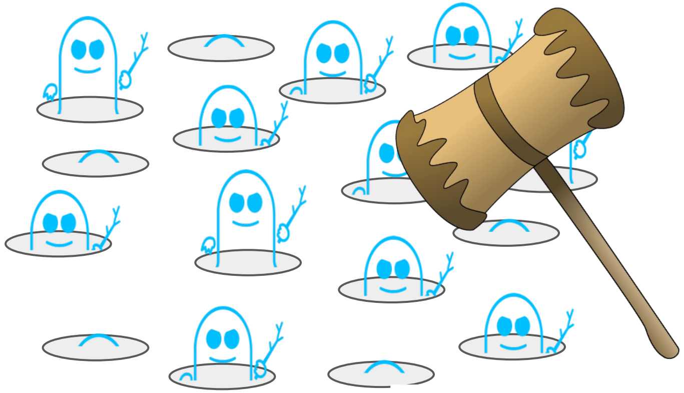
Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update

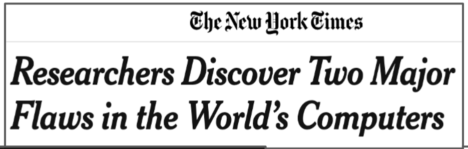


WIRED

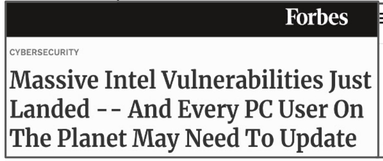
Meltdown and Spectre Fixes Arrive—But Don't Solve Everything





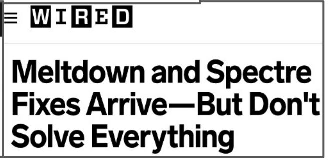

The New York Times

Researchers Discover Two Major Flaws in the World's Computers

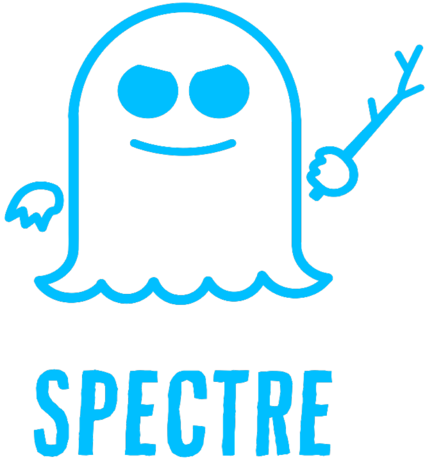

Forbes

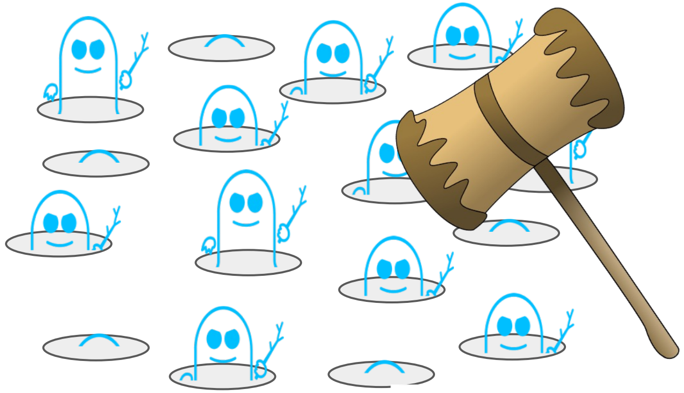
CYBERSECURITY

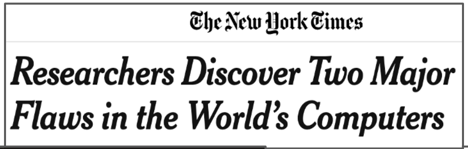
Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update


WIRED

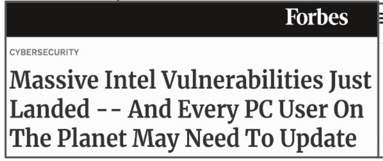
Meltdown and Spectre Fixes Arrive—But Don't Solve Everything





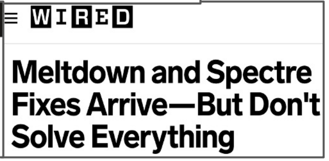

The New York Times

Researchers Discover Two Major Flaws in the World's Computers

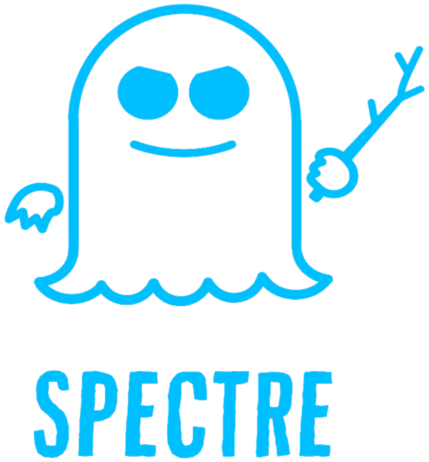

Forbes

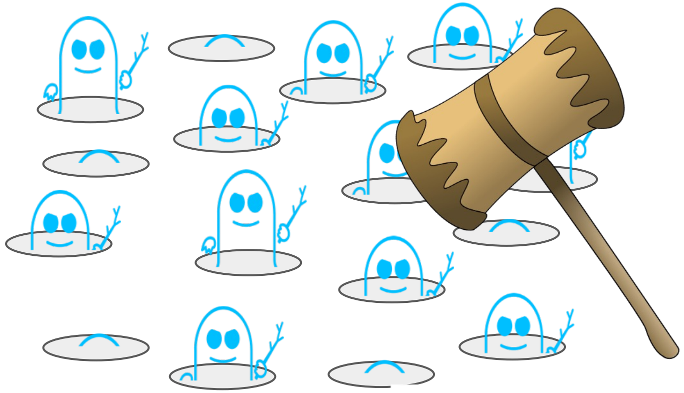
CYBERSECURITY

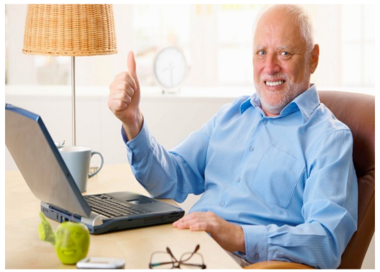
Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update

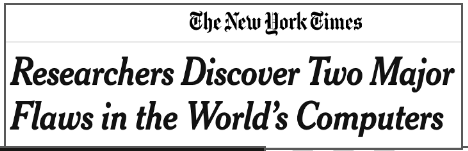

WIRED

Meltdown and Spectre Fixes Arrive—But Don't Solve Everything

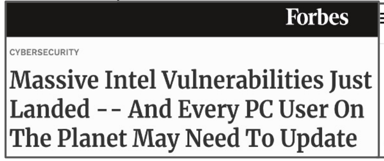






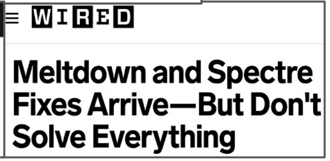

The New York Times

Researchers Discover Two Major Flaws in the World's Computers

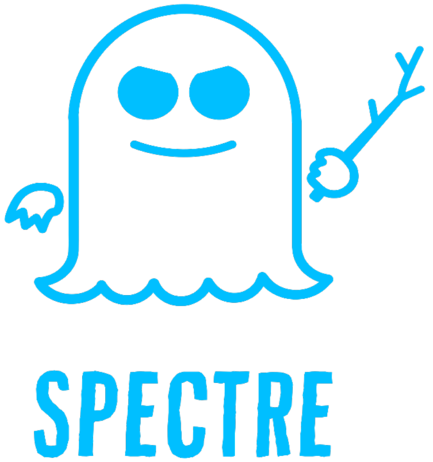

Forbes

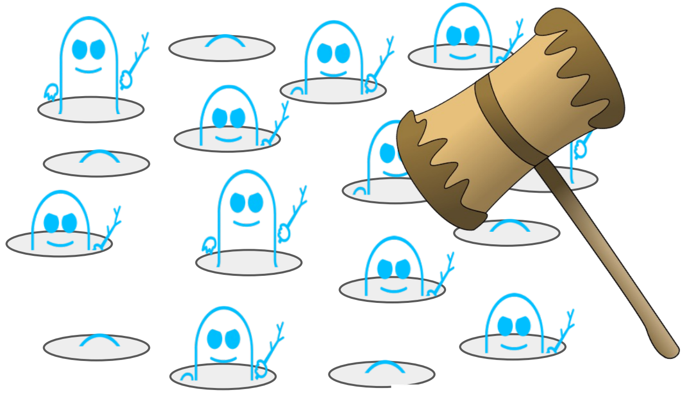
CYBERSECURITY

Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update


≡ WIRED

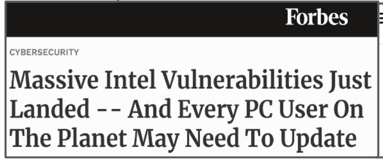
Meltdown and Spectre Fixes Arrive—But Don't Solve Everything





The New York Times

Researchers Discover Two Major Flaws in the World's Computers



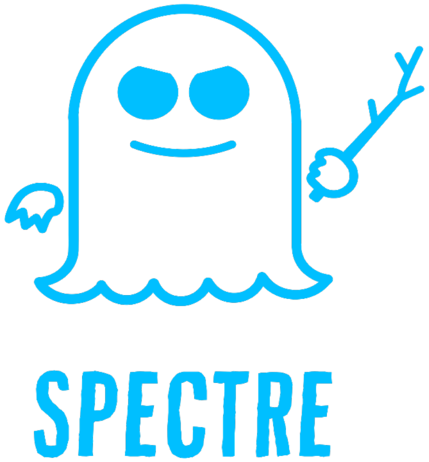
Forbes

CYBERSECURITY

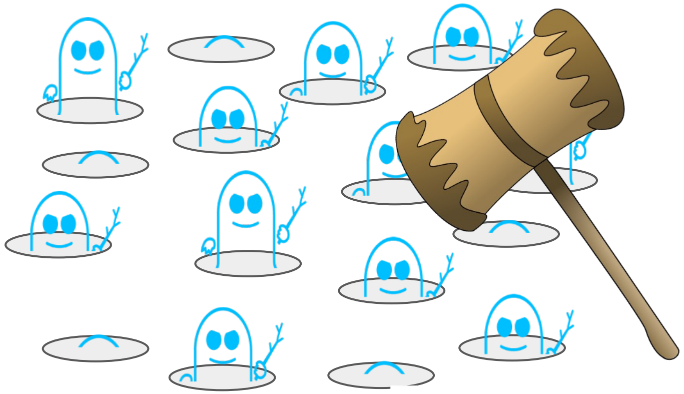
Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update

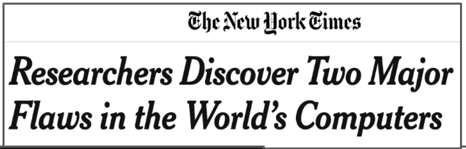
WIRED

Meltdown and Spectre Fixes Arrive—But Don't Solve Everything



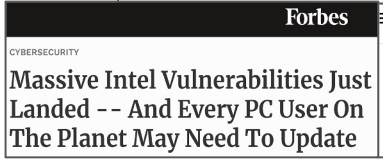
1379 gadgets





The New York Times

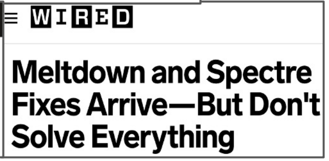
Researchers Discover Two Major Flaws in the World's Computers



Forbes

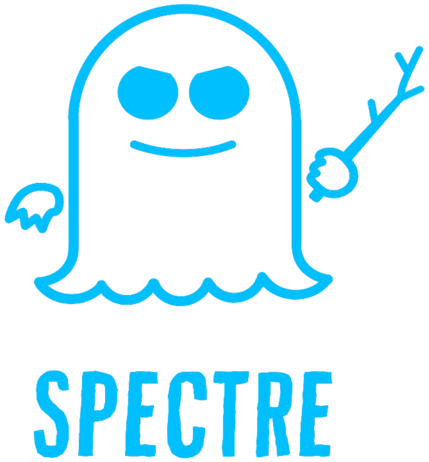
CYBERSECURITY

Massive Intel Vulnerabilities Just Landed -- And Every PC User On The Planet May Need To Update

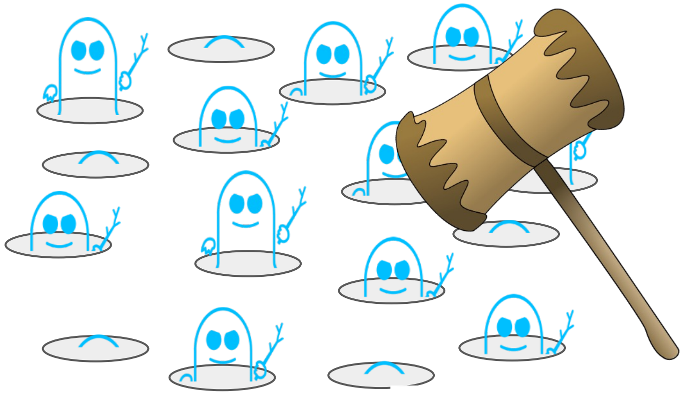


WIRED

Meltdown and Spectre Fixes Arrive—But Don't Solve Everything



1379 gadgets



What is a Spectre gadget?

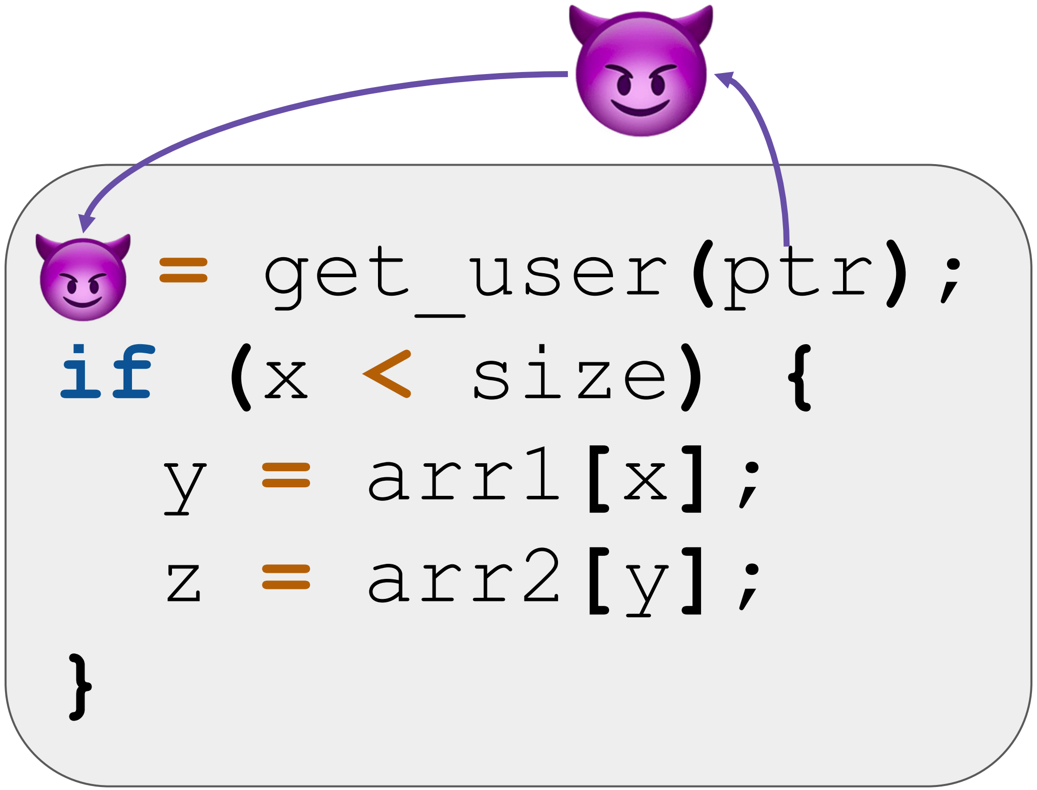
What is a Spectre gadget?

```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

What is a Spectre gadget?

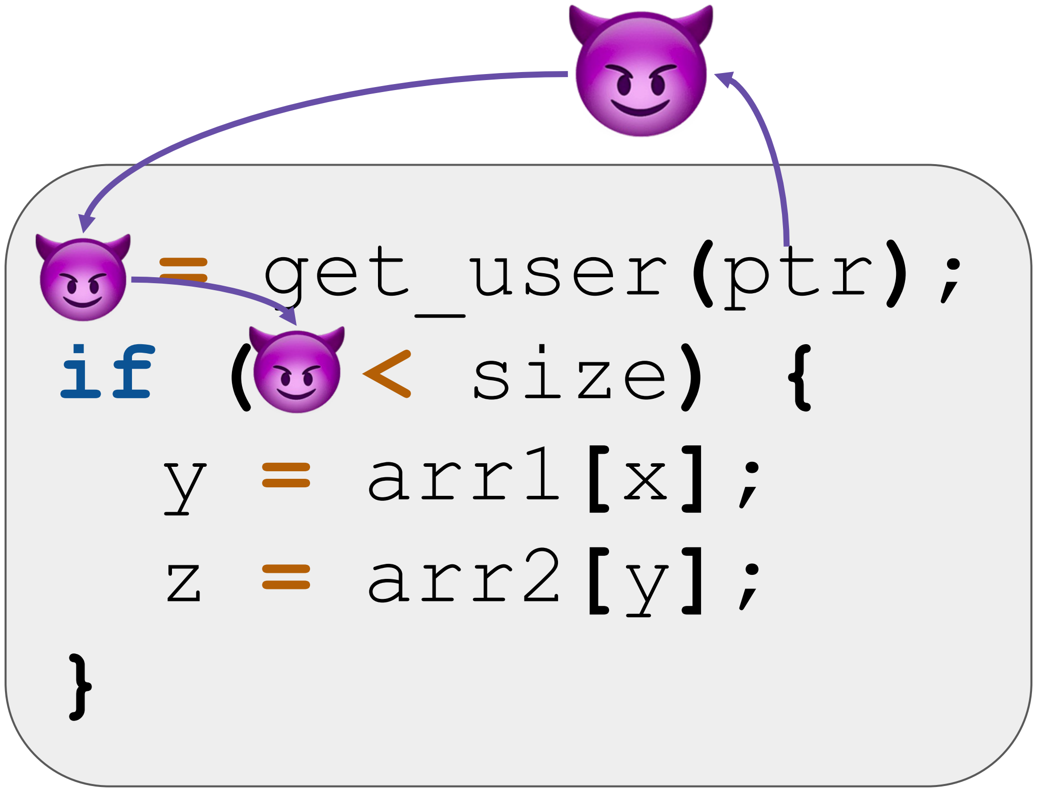
```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

What is a Spectre gadget?



```
😈 = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

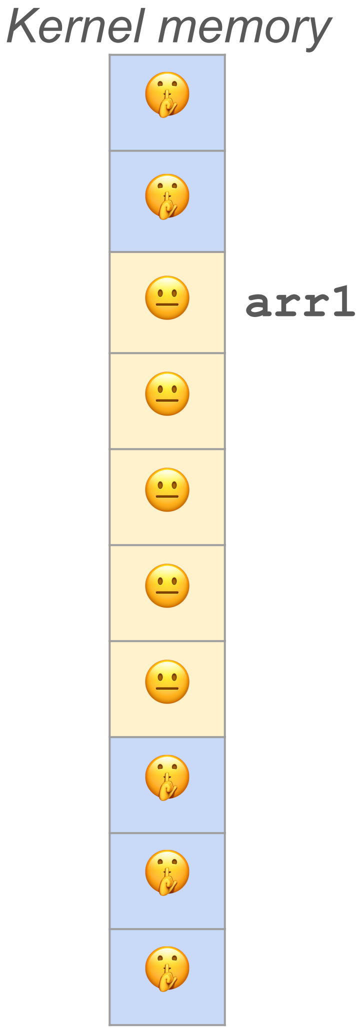
What is a Spectre gadget?



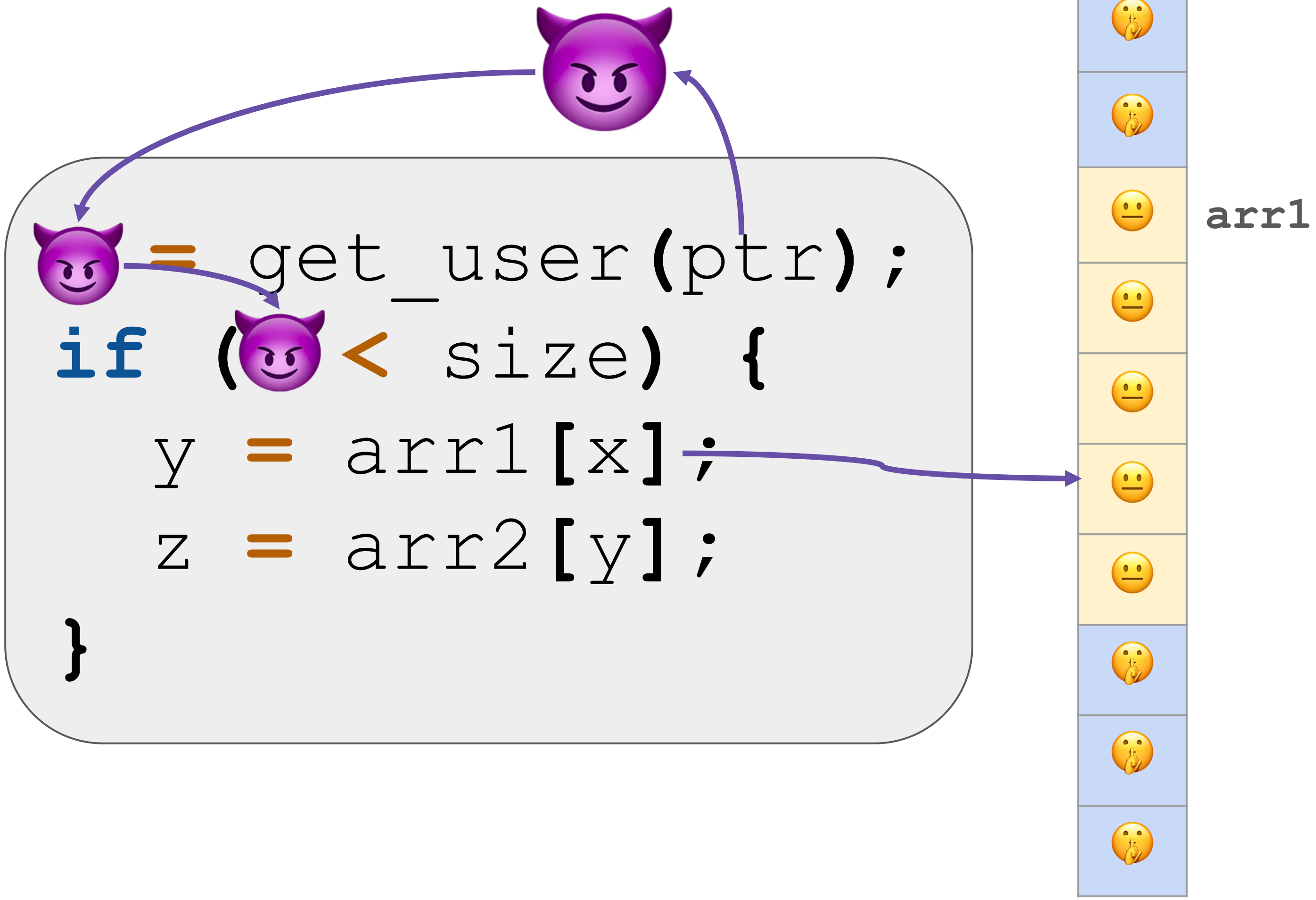
```
😈 = get_user(ptr);
if (😈 < size) {
   y = arr1[x];
   z = arr2[y];
}
```

What is a Spectre gadget?

```
😈 = get_user(ptr);
if (😈 < size) {
    y = arr1[x];
    z = arr2[y];
}
```

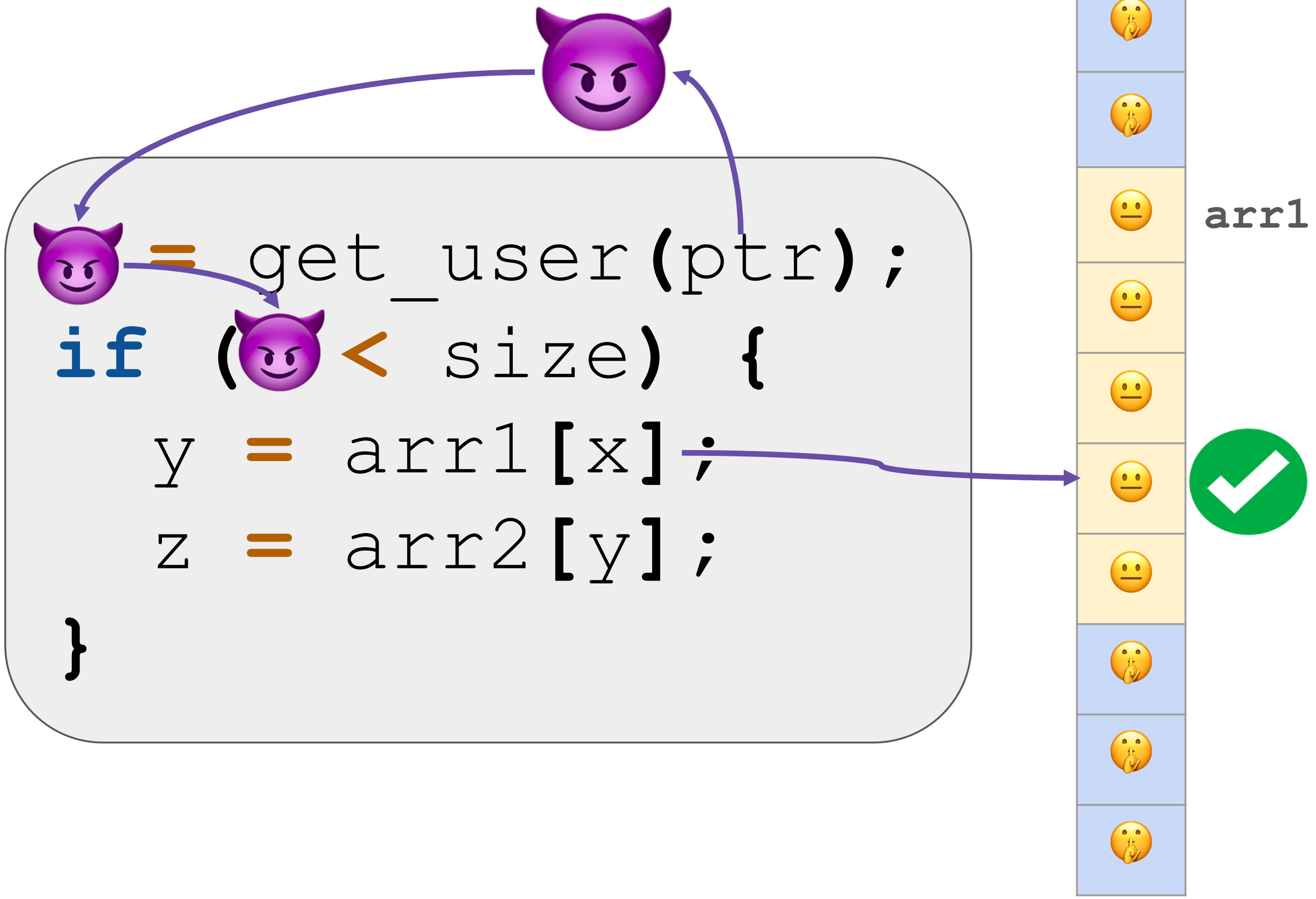


What is a Spectre gadget?



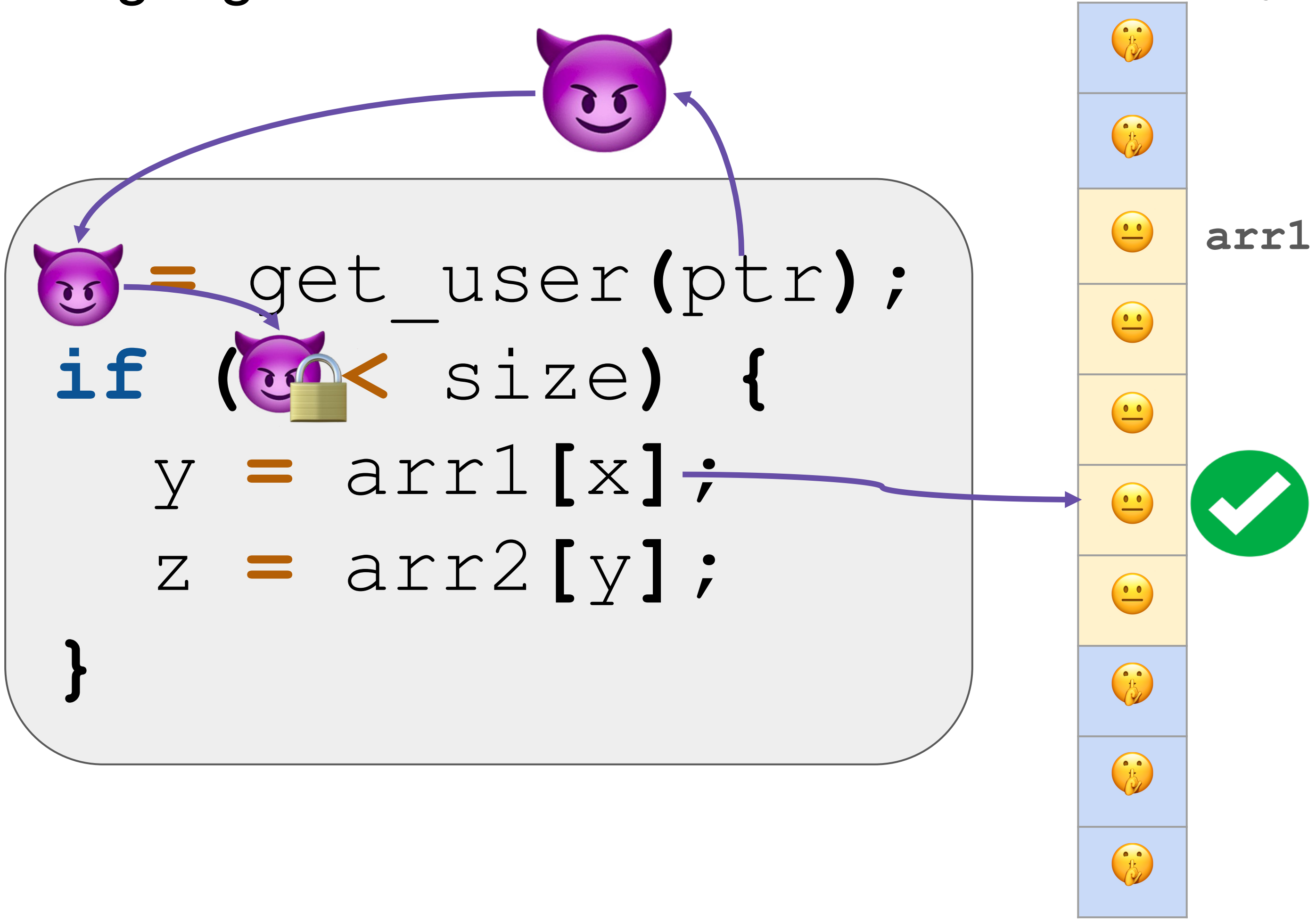
```
😈 = get_user(ptr);
if (😈 < size) {
  y = arr1[x];
  z = arr2[y];
}
```

What is a Spectre gadget?



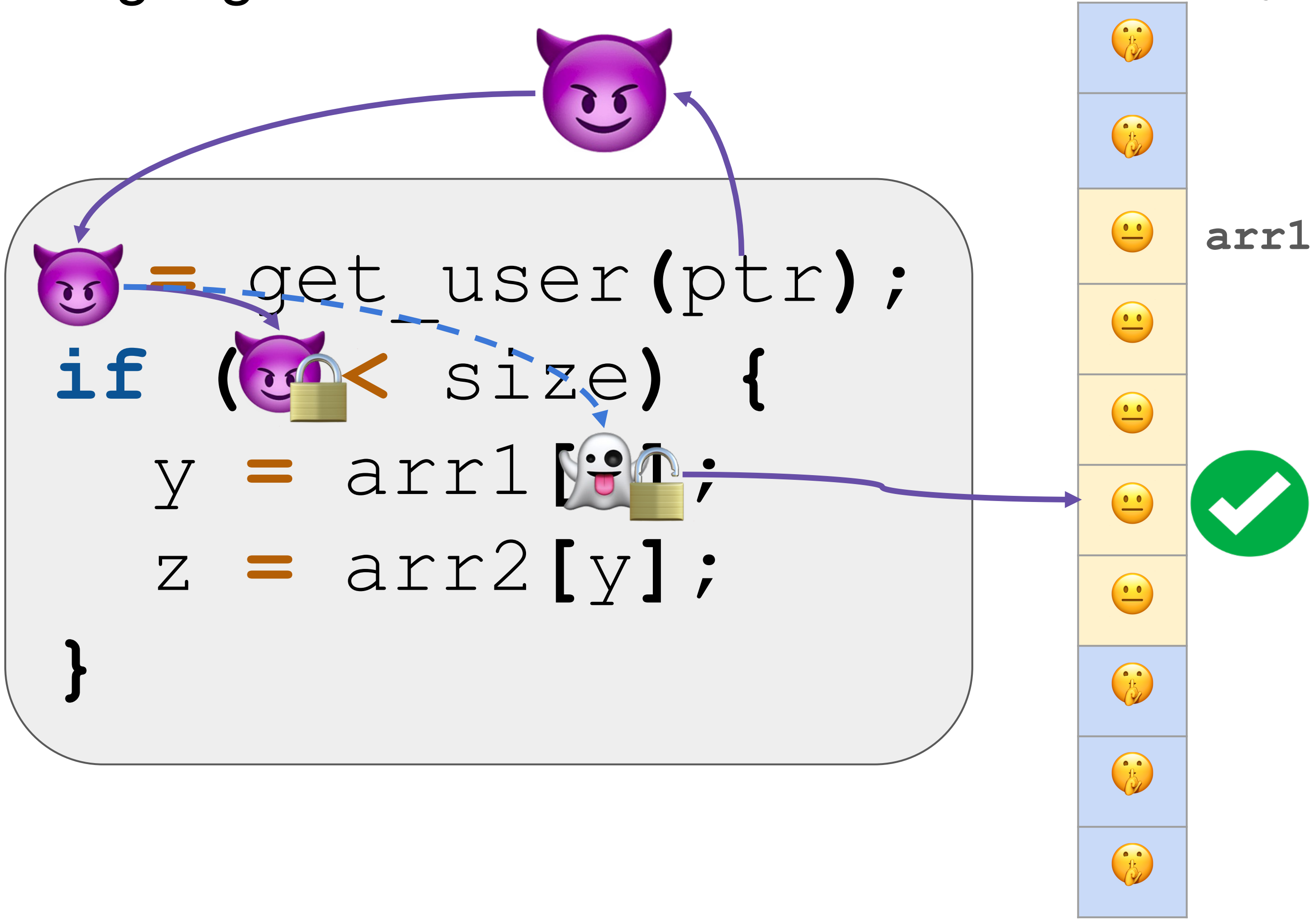
```
😈 = get_user(ptr);
if (😈 < size) {
    y = arr1[x];
    z = arr2[y];
}
```

What is a Spectre gadget?

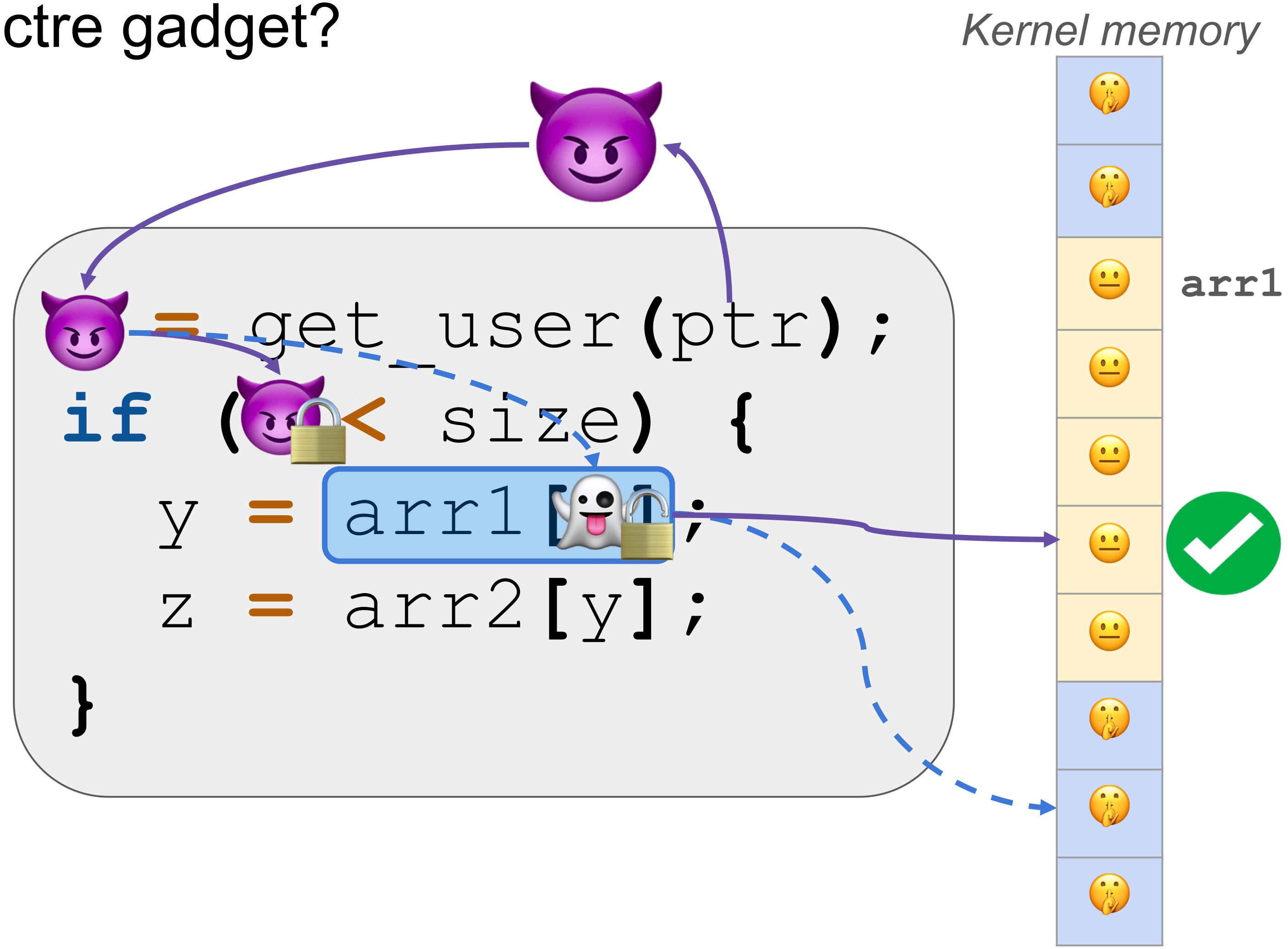


```
😈 = get_user(ptr);
if (😈🔒< size) {
   y = arr1[x];
   z = arr2[y];
}
```

What is a Spectre gadget?

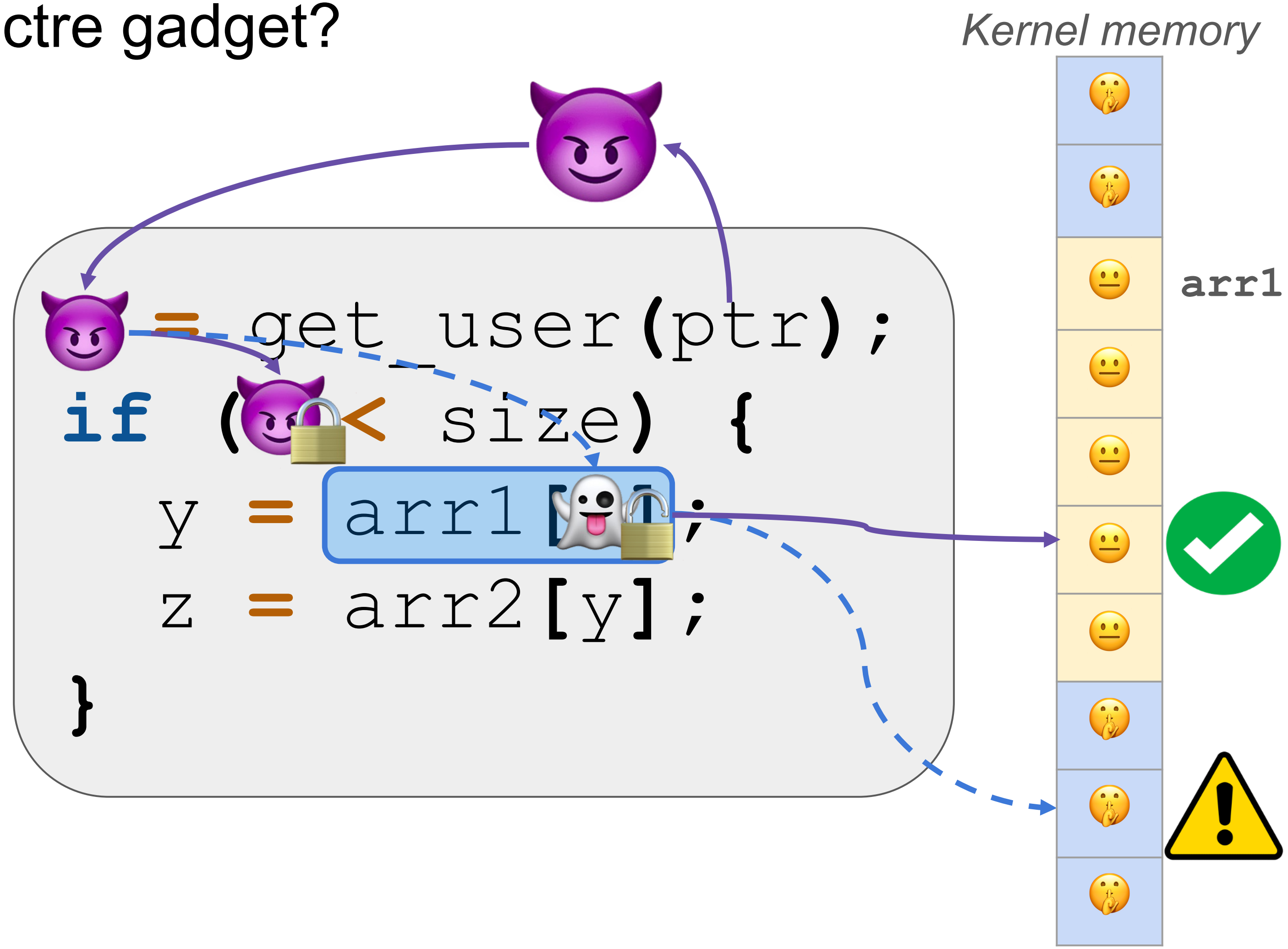


What is a Spectre gadget?



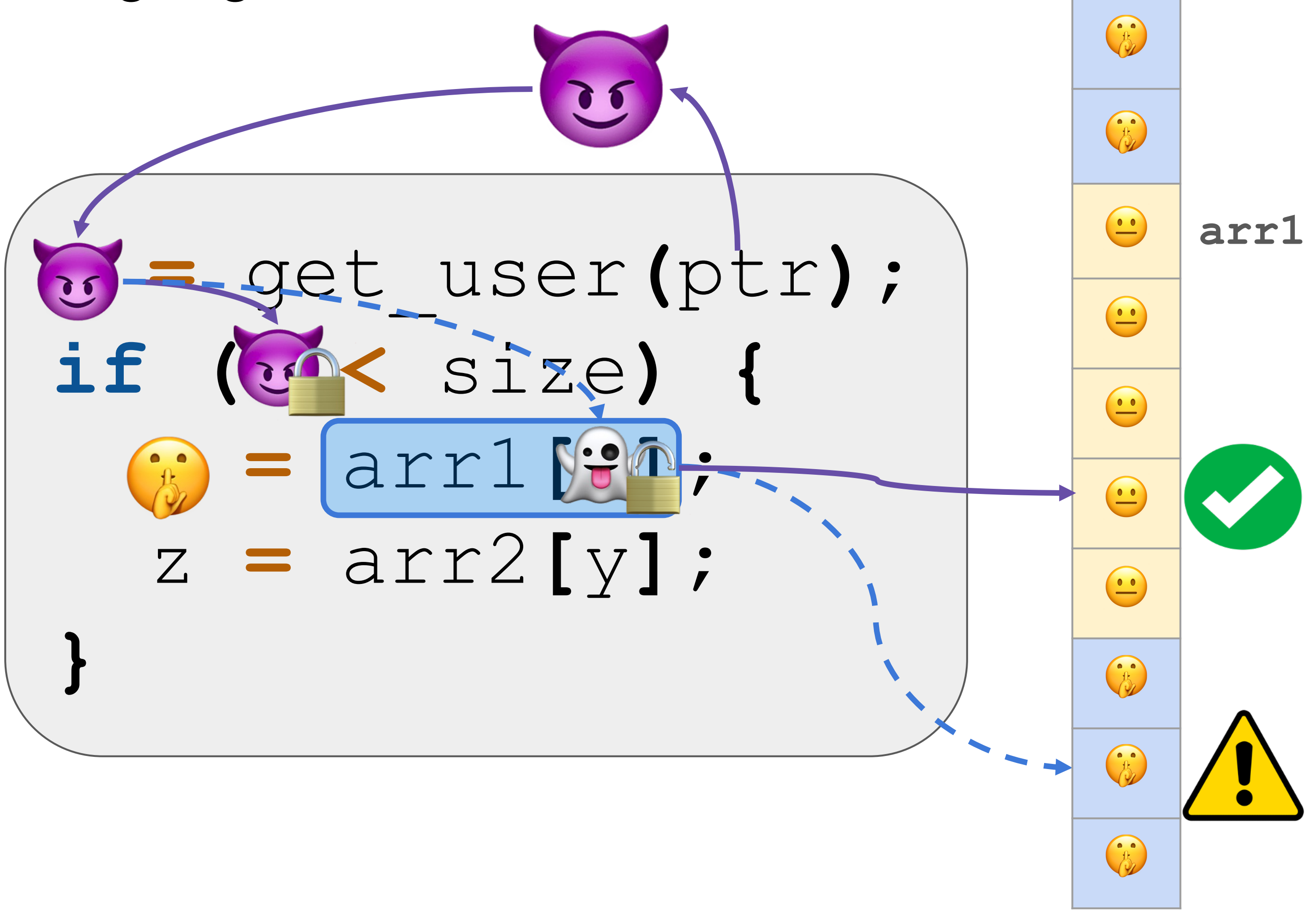
```
😈 = get_user(ptr);
if (😈🔒< size) {
    y = arr1[👻🔒];
    z = arr2[y];
}
```

What is a Spectre gadget?

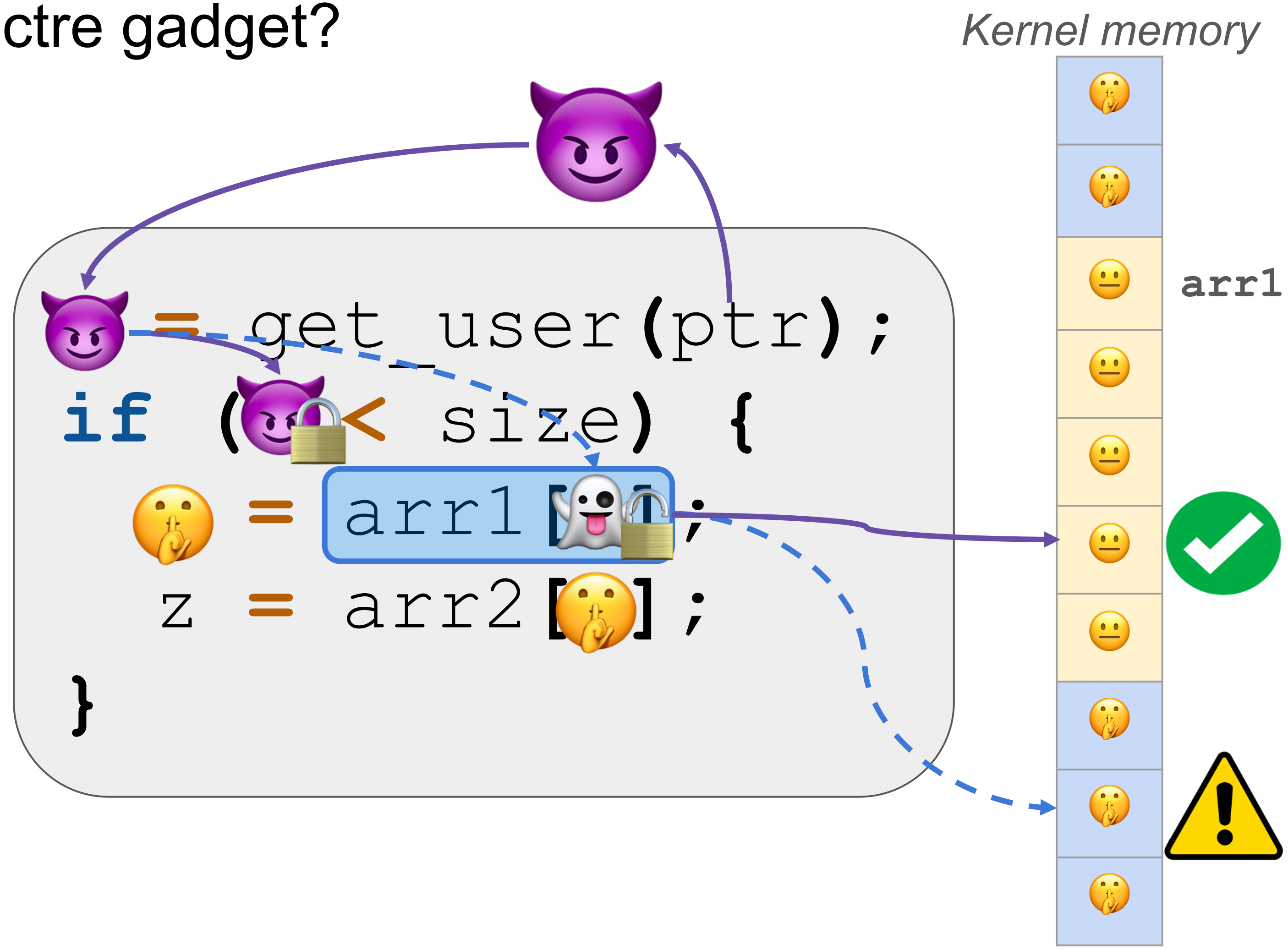


```
😈 = get_user(ptr);
if (😈🔒< size) {
    y = arr1[👻🔒];
    z = arr2[y];
}
```

What is a Spectre gadget?

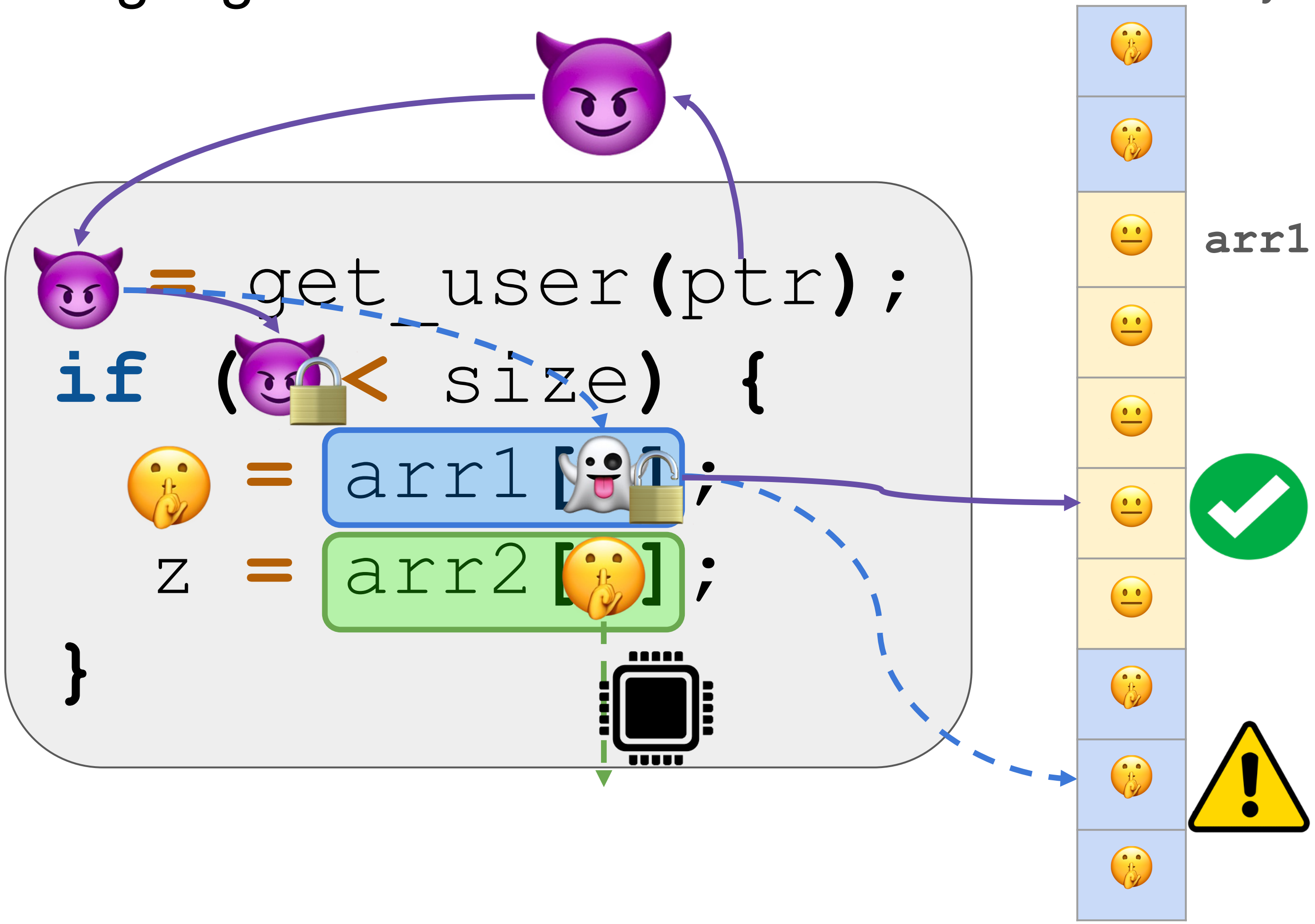


What is a Spectre gadget?



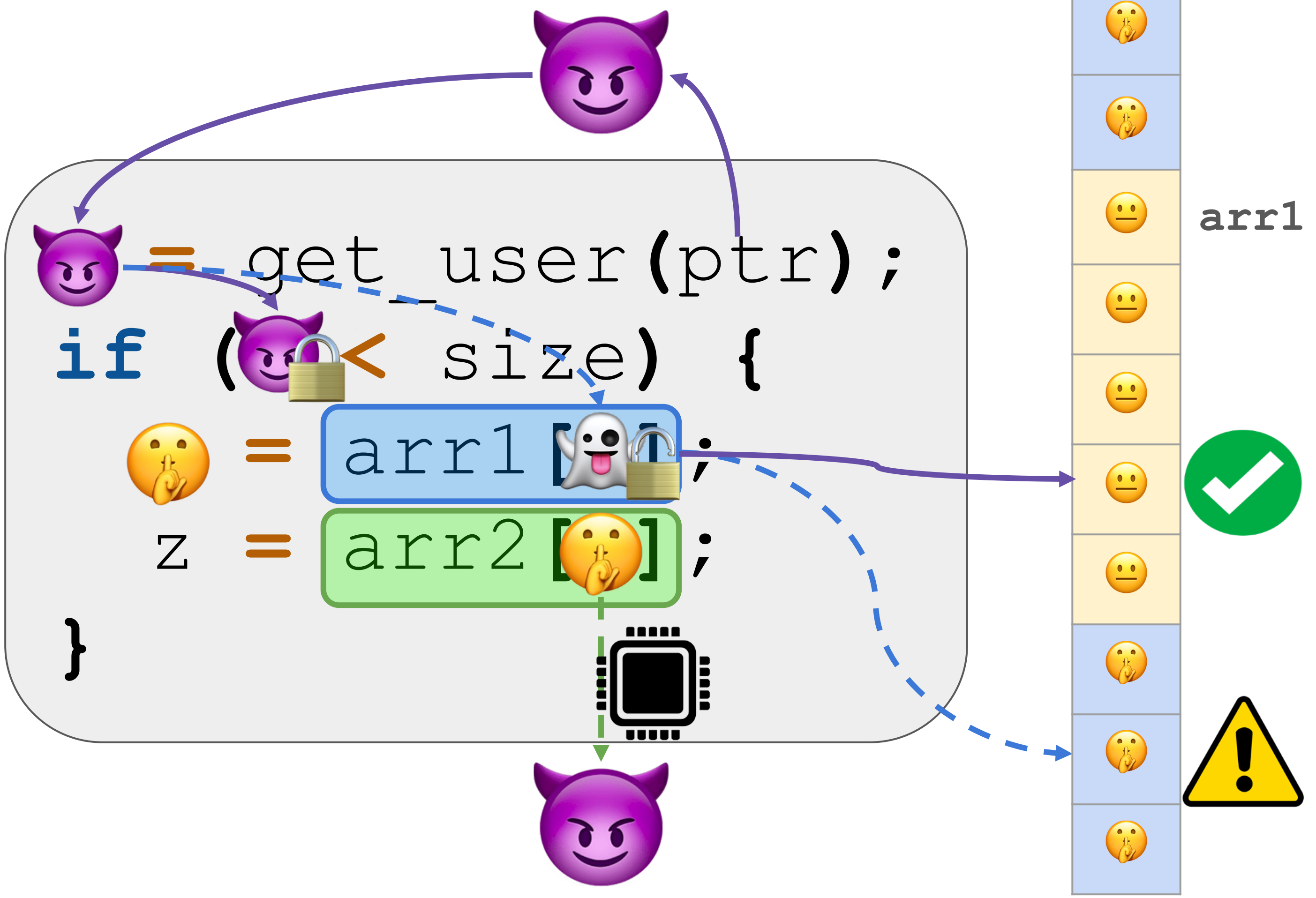
```
😈 = get_user(ptr);
if (😈🔒< size) {
  🤫 = arr1[👻🔒];
  z = arr2[🤫];
}
```

What is a Spectre gadget?



```
😈 = get_user(ptr);
if (😈🔒< size) {
  🤫 = arr1[👻🔓];
  z = arr2[🤫];
}
```

What is a Spectre gadget?



How do existing gadget scanners work?

```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

How do existing gadget scanners work?

Existing scanners are **pattern-driven**.

```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

How do existing gadget scanners work?

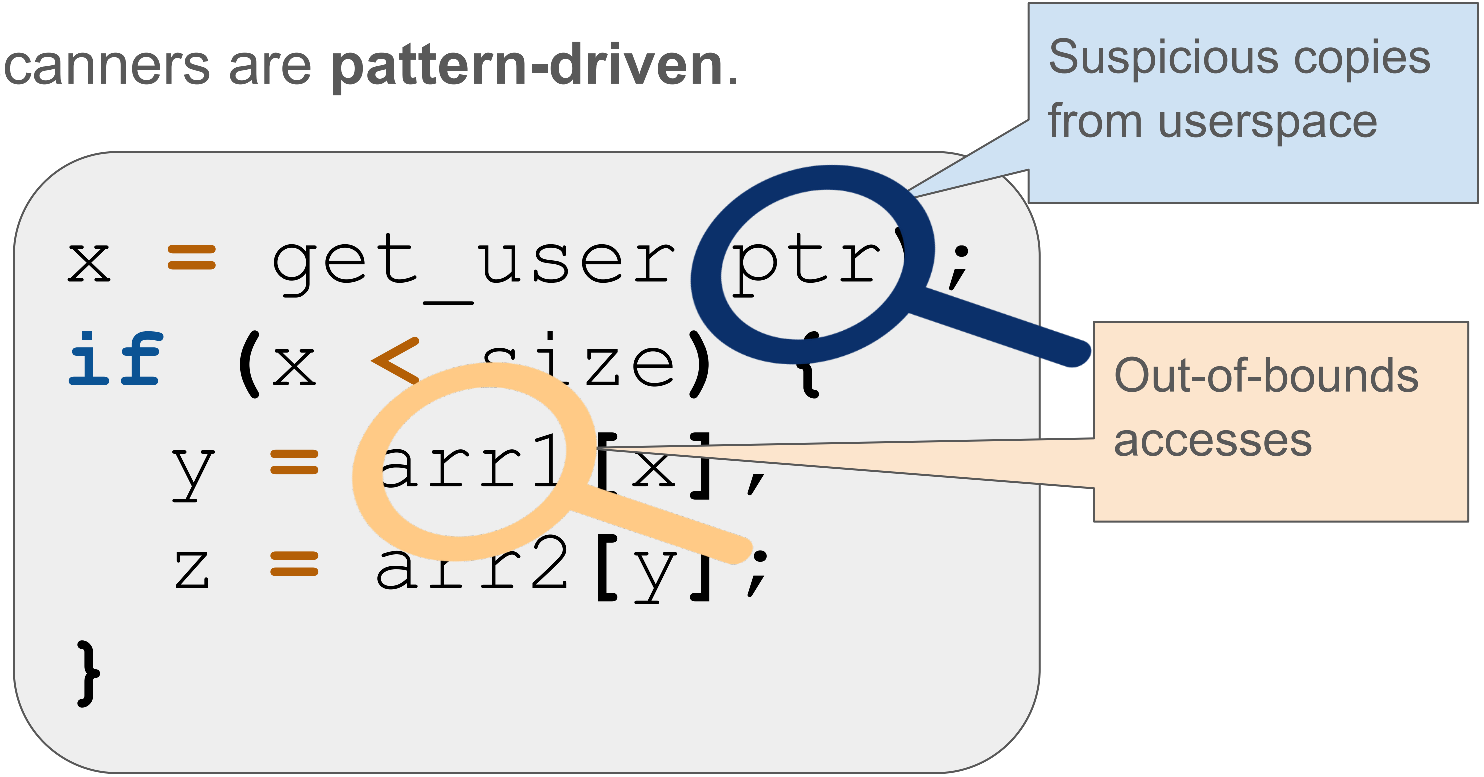
Existing scanners are **pattern-driven**.

Suspicious copies from userspace

```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

How do existing gadget scanners work?

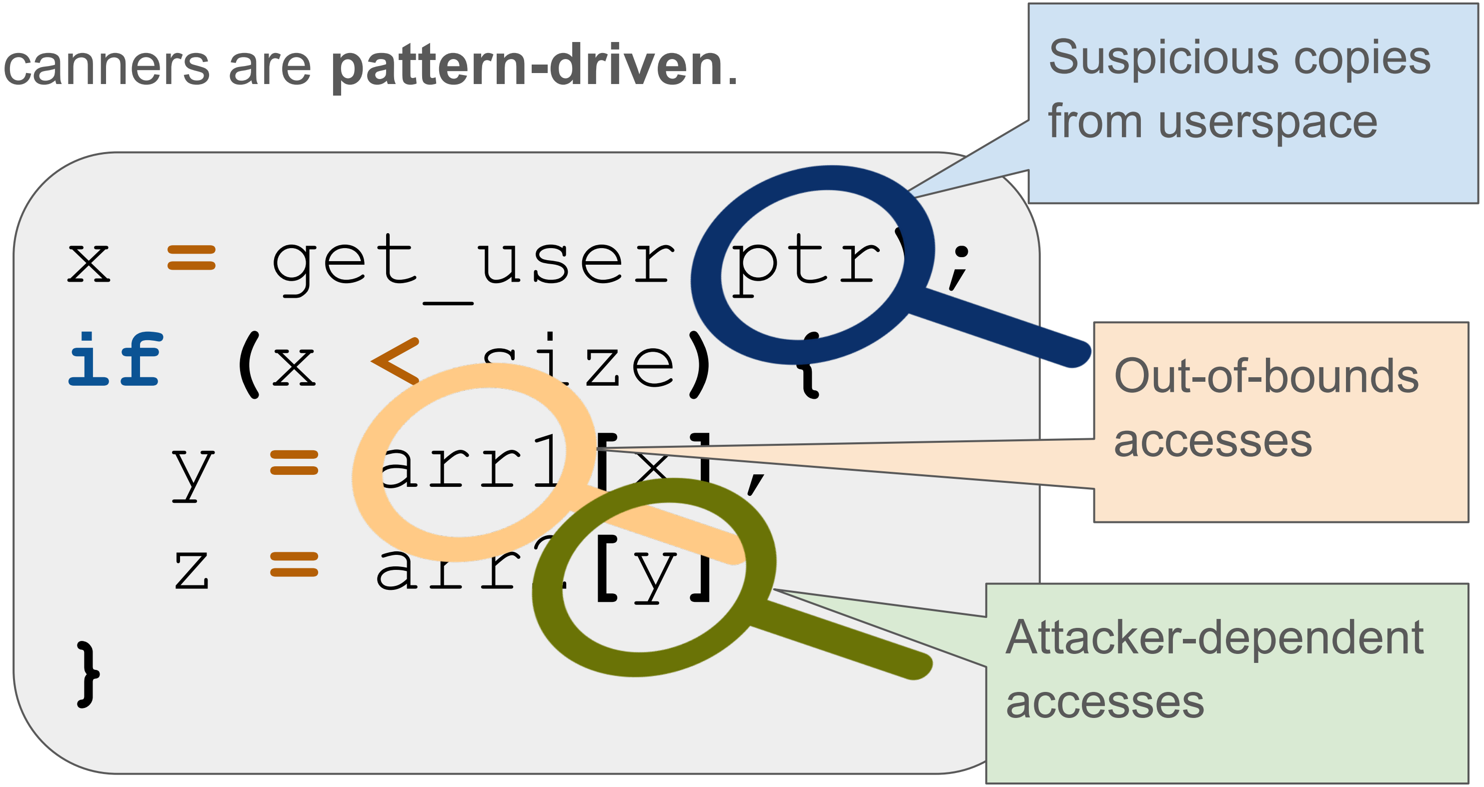
Existing scanners are **pattern-driven**.



```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

How do existing gadget scanners work?

Existing scanners are **pattern-driven**.



```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

How do existing gadget scanners work?

```
x = get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```

How do existing gadget scanners work?

⚠ Existing scanners are **limited in scope**.

```
x = get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```

How do existing gadget scanners work?

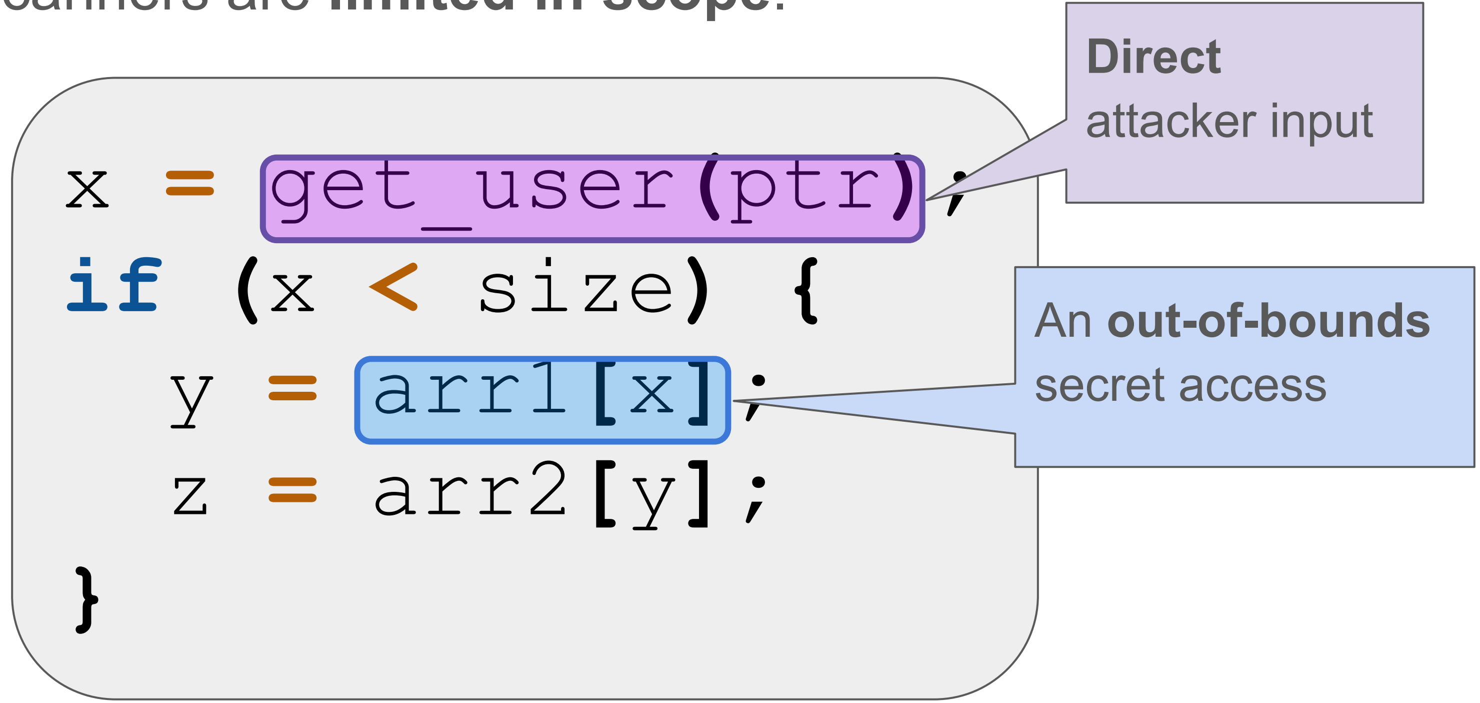
⚠ Existing scanners are **limited in scope**.

```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

Direct
attacker input

How do existing gadget scanners work?

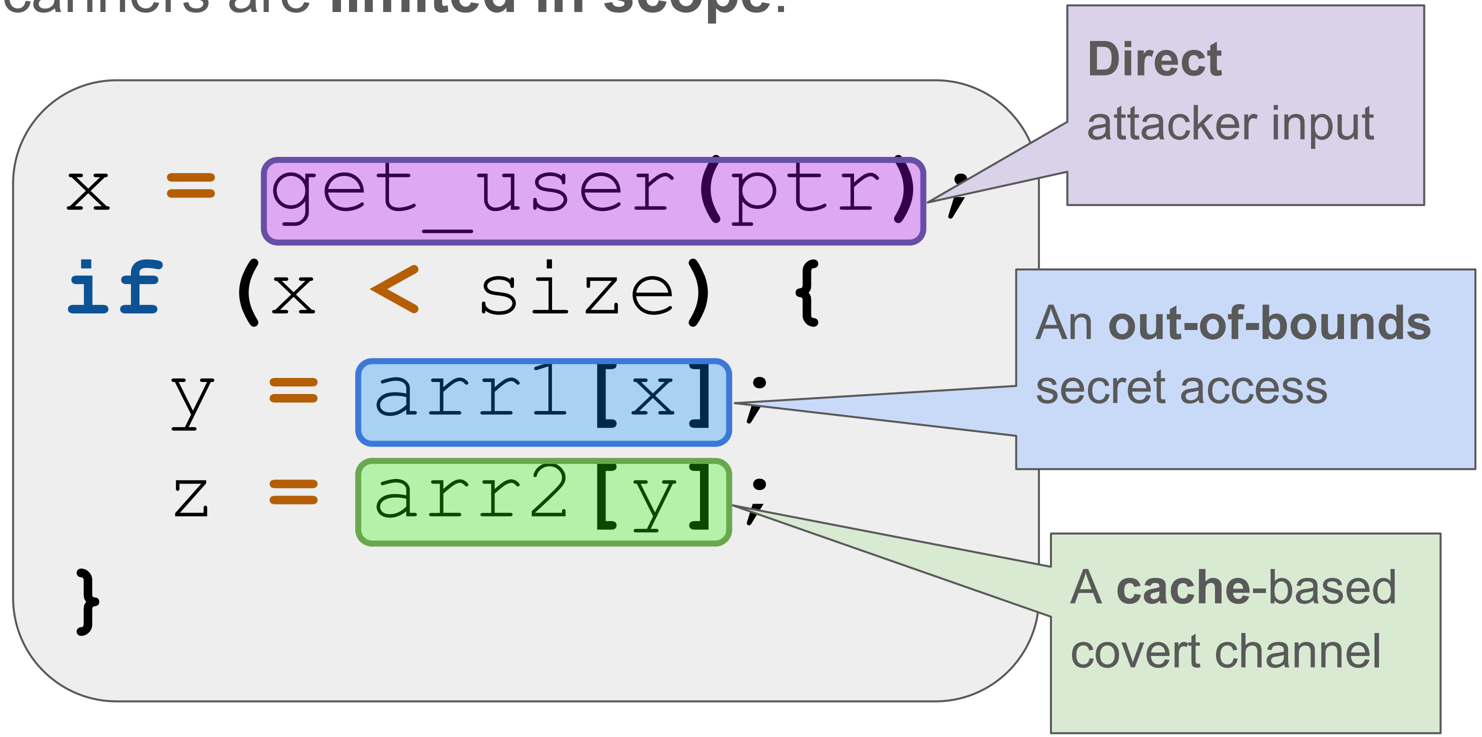
Existing scanners are **limited in scope**.



```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

How do existing gadget scanners work?

Existing scanners are **limited in scope**.



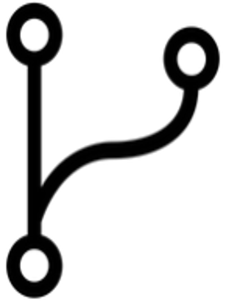
```
x = get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```

Direct attacker input
An **out-of-bounds** secret access
A **cache**-based covert channel

However, in reality, **attackers do not care about patterns**.

However, in reality, **attackers do not care about patterns**.

However, in reality, **attackers do not care about patterns**.



However, in reality, **attackers do not care about patterns**.

```
mov
call
lea
```

However, in reality, **attackers do not care about patterns**.

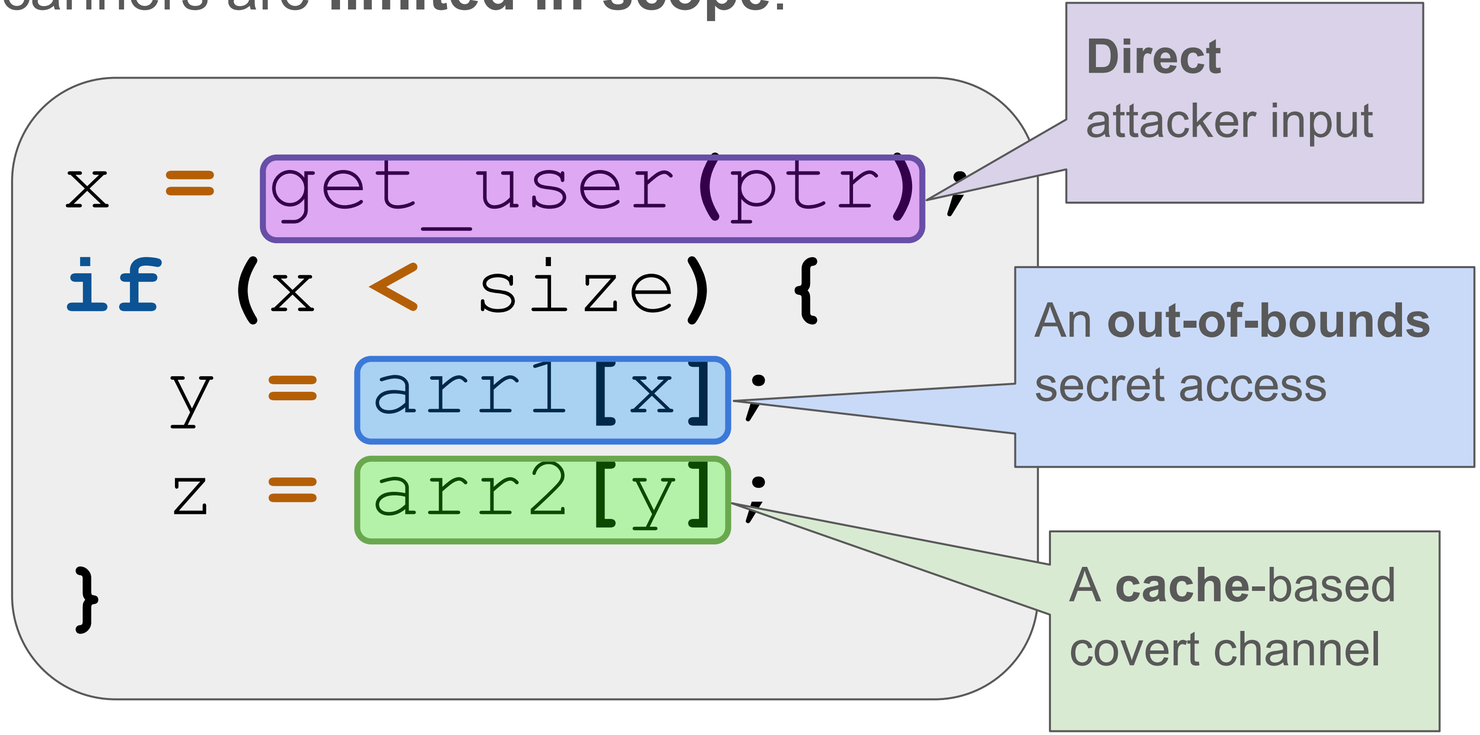
```
mov
call
lea
```

However, in reality, **attackers do not care about patterns**.

```
mov
call
lea
```

How do existing gadget scanners work?

⚠ Existing scanners are **limited in scope**.

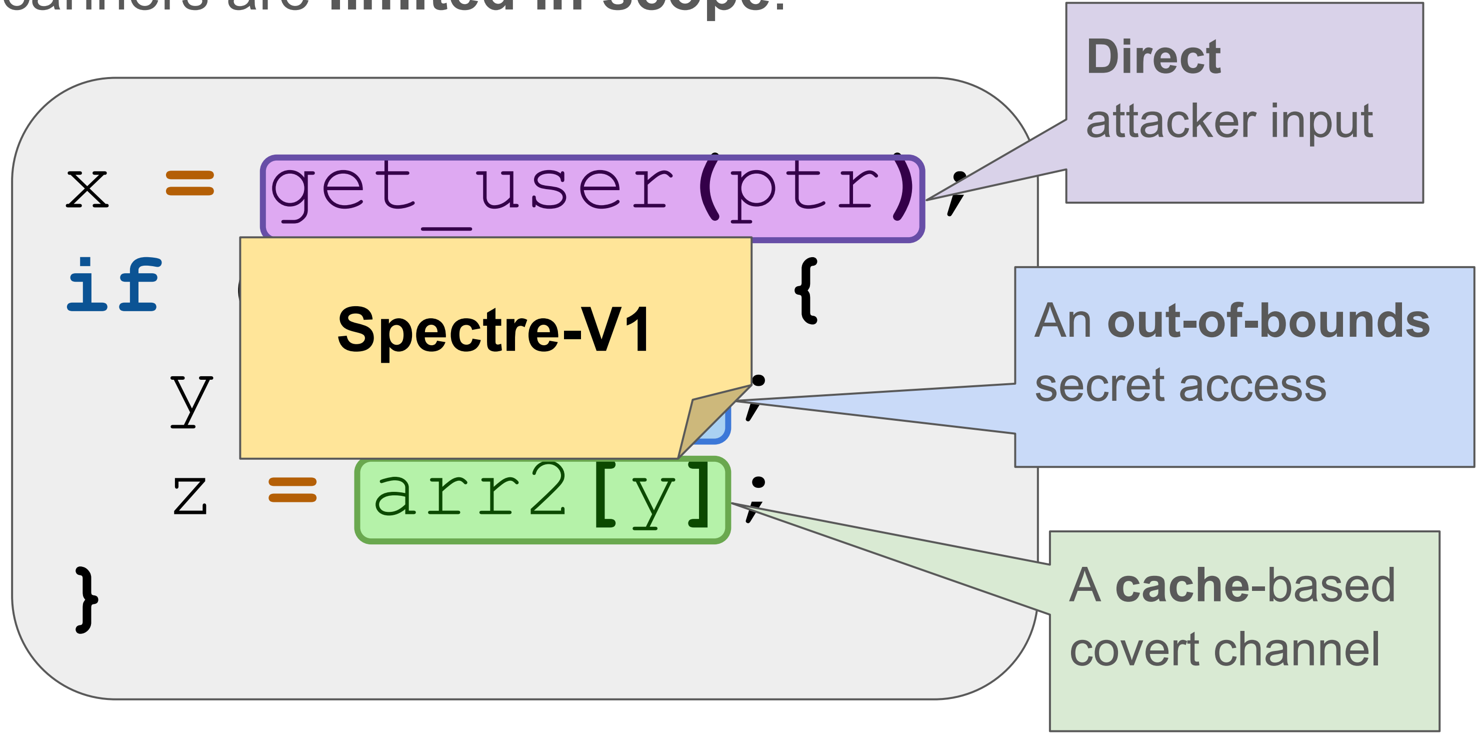


```
x = get_user(ptr);
if (x < size) {
    y = arr1[x];
    z = arr2[y];
}
```

Direct attacker input
An **out-of-bounds** secret access
A **cache**-based covert channel

How do existing gadget scanners work?

Existing scanners are **limited in scope**.

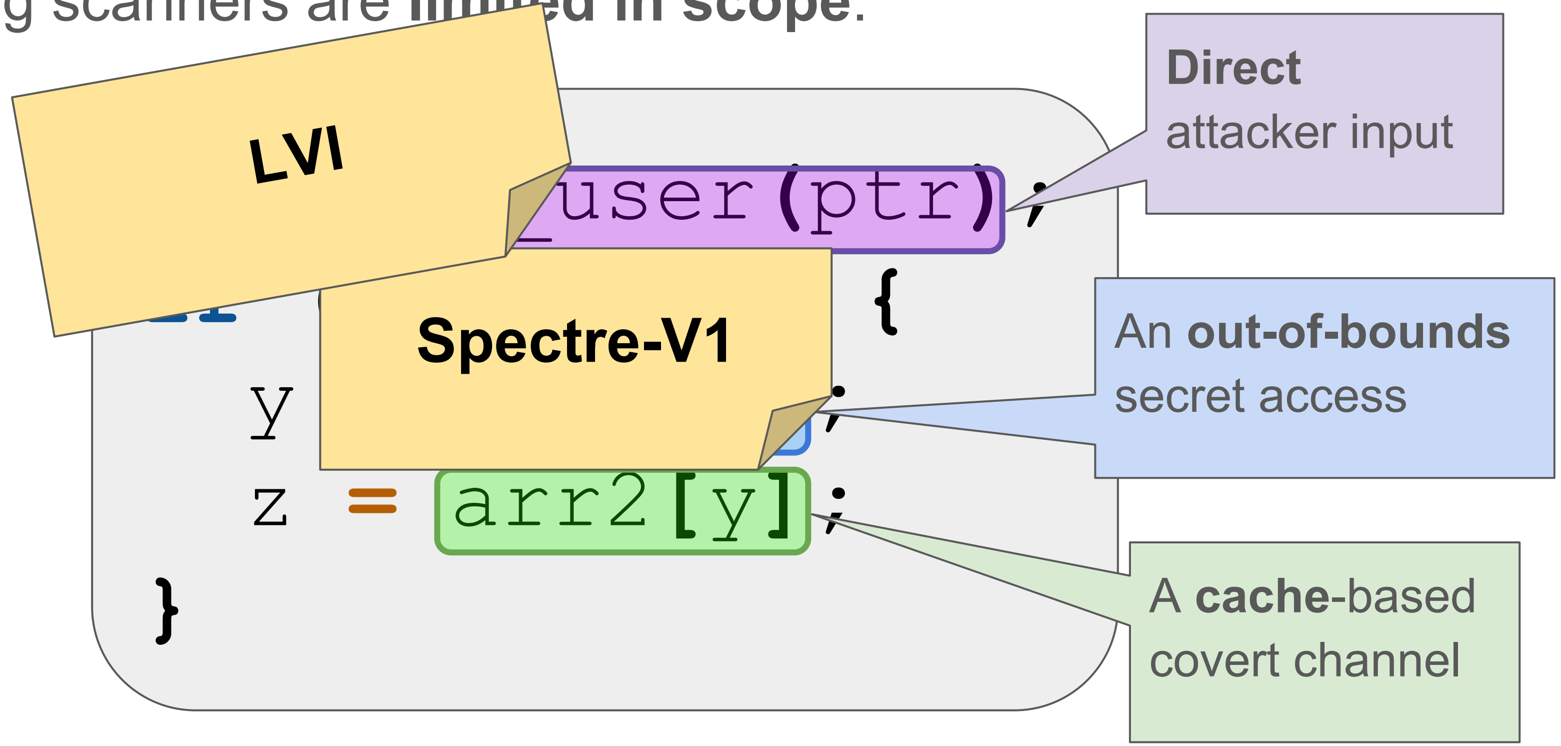


```
x = get_user(ptr);
if (                   {
    y                  ;
    z = arr2[y];
}
```

Spectre-V1
Direct attacker input
An **out-of-bounds** secret access
A **cache**-based covert channel

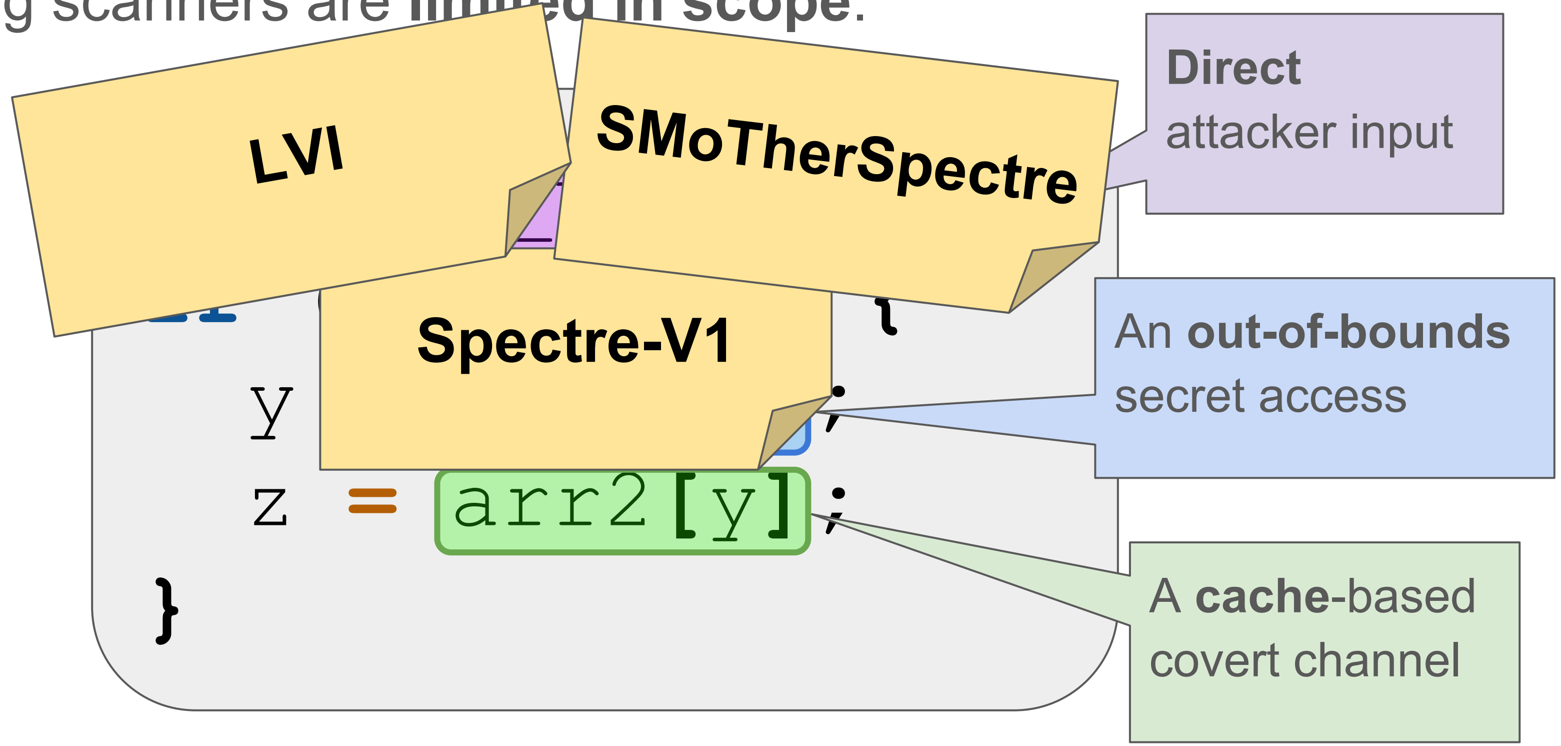
How do existing gadget scanners work?

Existing scanners are **limited in scope**.



How do existing gadget scanners work?

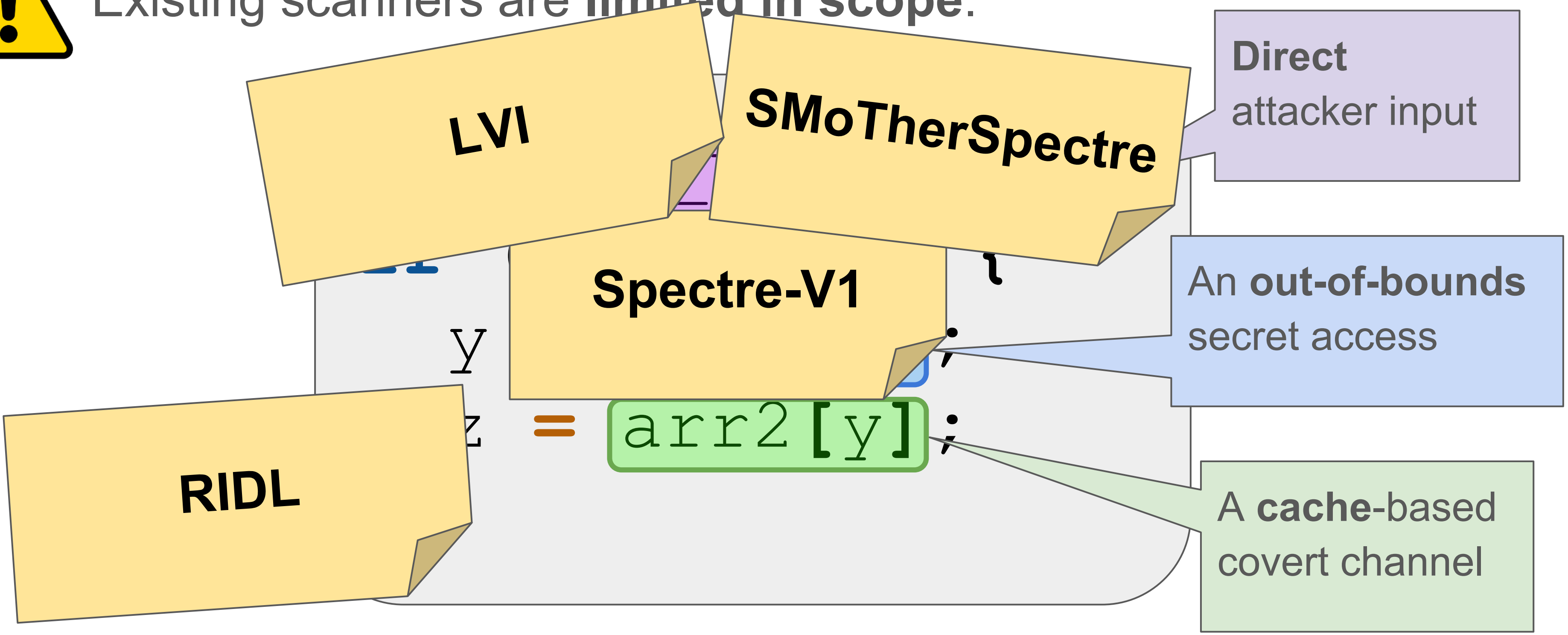
Existing scanners are **limited in scope**.



LVI
SMoTherSpectre
Spectre-V1
y
z = arr2[y];
}
Direct attacker input
An **out-of-bounds** secret access
A **cache**-based covert channel

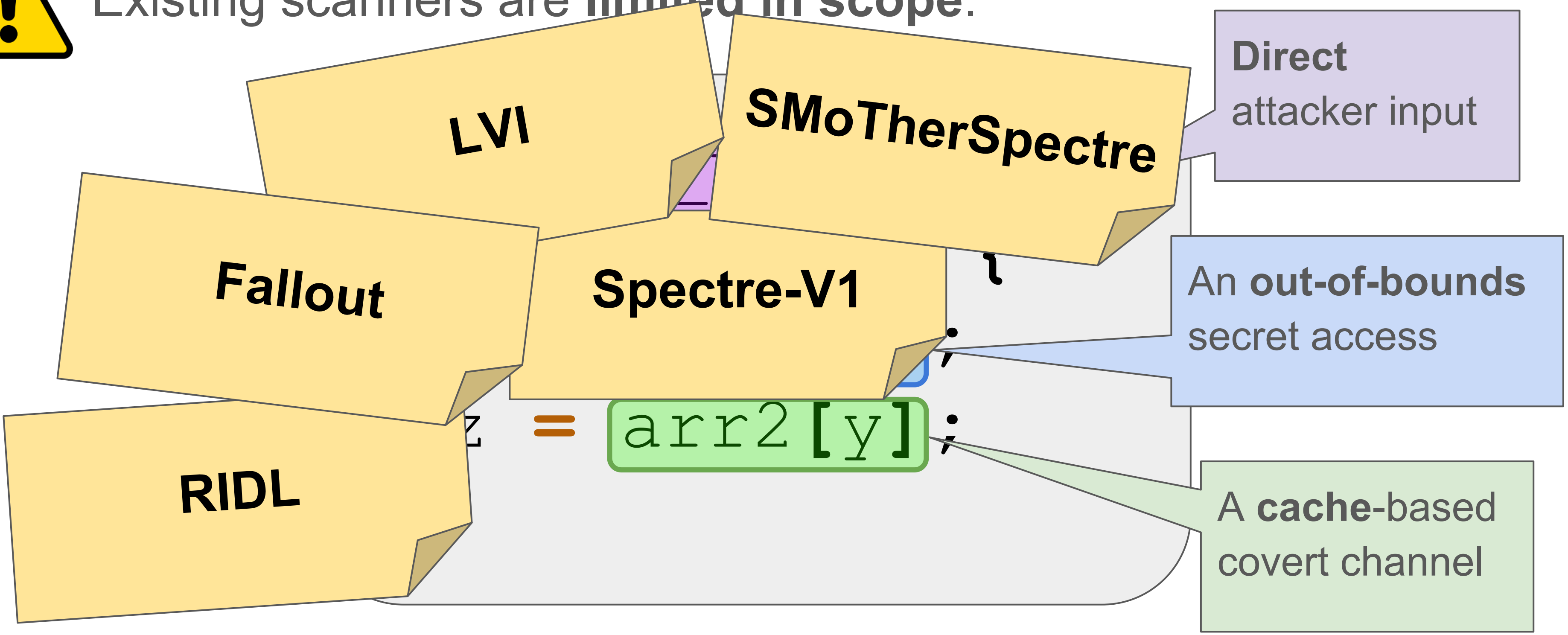
How do existing gadget scanners work?

Existing scanners are **limited in scope**.


LVI
SMoTherSpectre
Spectre-V1
RIDL
y
z = arr2[y];
Direct attacker input
An **out-of-bounds** secret access
A **cache**-based covert channel

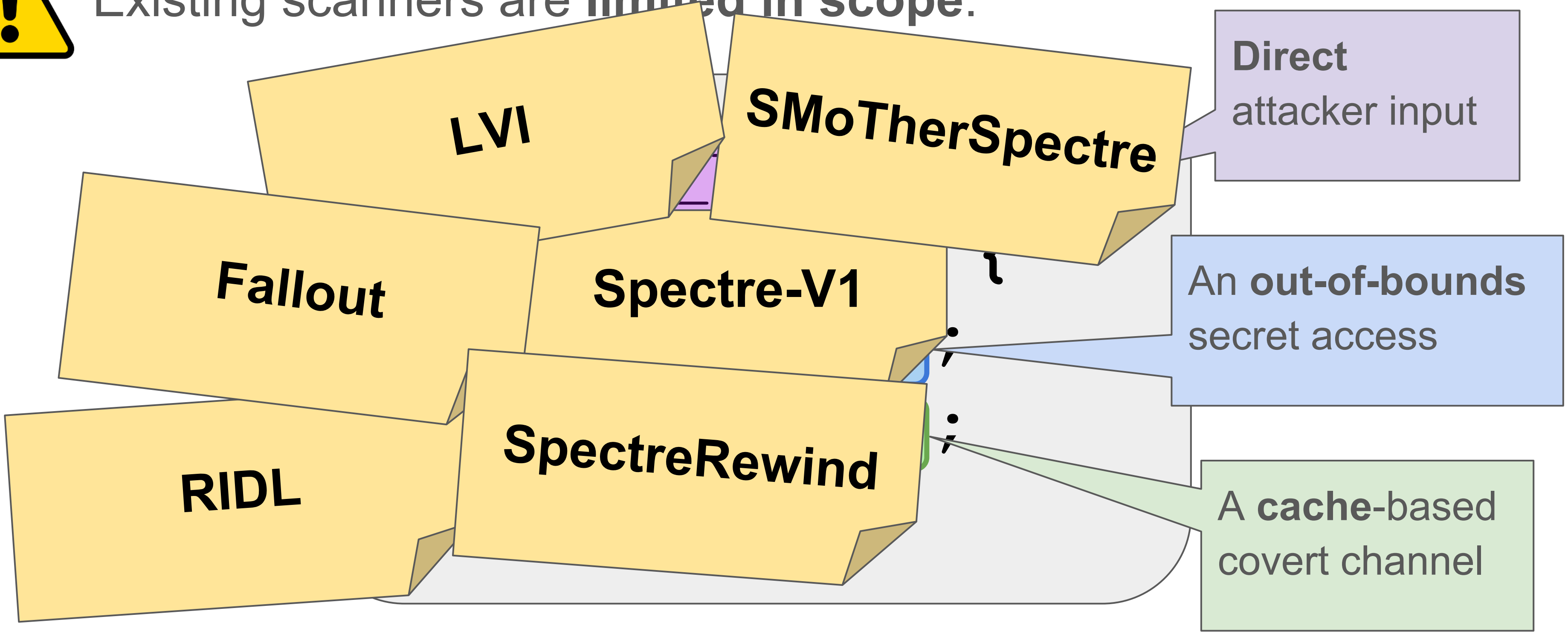
How do existing gadget scanners work?

Existing scanners are **limited in scope**.


LVI
SMoTherSpectre
Fallout
Spectre-V1
RIDL
z = arr2[y];
Direct attacker input
An **out-of-bounds** secret access
A **cache**-based covert channel

How do existing gadget scanners work?

Existing scanners are **limited in scope**.


LVI
SMoTherSpectre
Fallout
Spectre-V1
RIDL
SpectreRewind
Direct attacker input
An **out-of-bounds** secret access
A **cache**-based covert channel

How do existing gadget scanners work?

⚠ Existing scanners are **limited in scope**.

LVI

SMoTherSpectre

Fallout

Spectre-V1

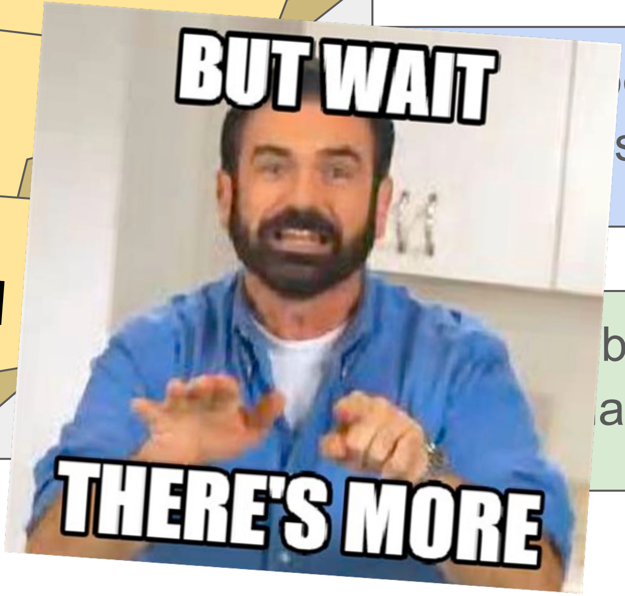
RIDL

SpectreRewind

Direct
attacker input

ounds
ss

based
annel



An attacker has **all of these primitives** at their disposal.

An attacker has **all of these primitives** at their disposal.

An attacker has **all of these primitives** at their disposal.

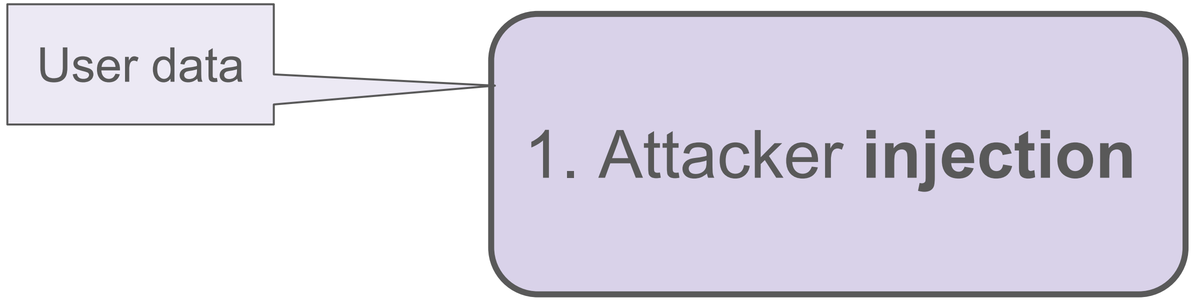
But how can a gadget scanner even begin to **model all of them**?

Our approach: Modeling generalized gadgets

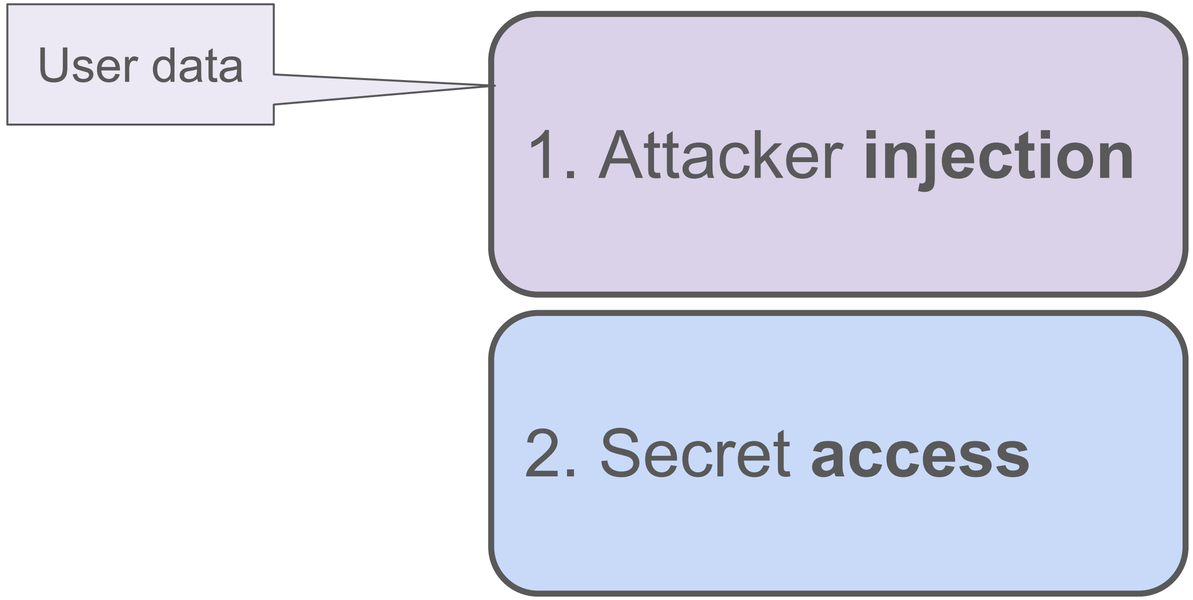
Our approach: Modeling generalized gadgets

1. Attacker **injection**

Our approach: Modeling generalized gadgets



Our approach: Modeling generalized gadgets



Our approach: Modeling generalized gadgets

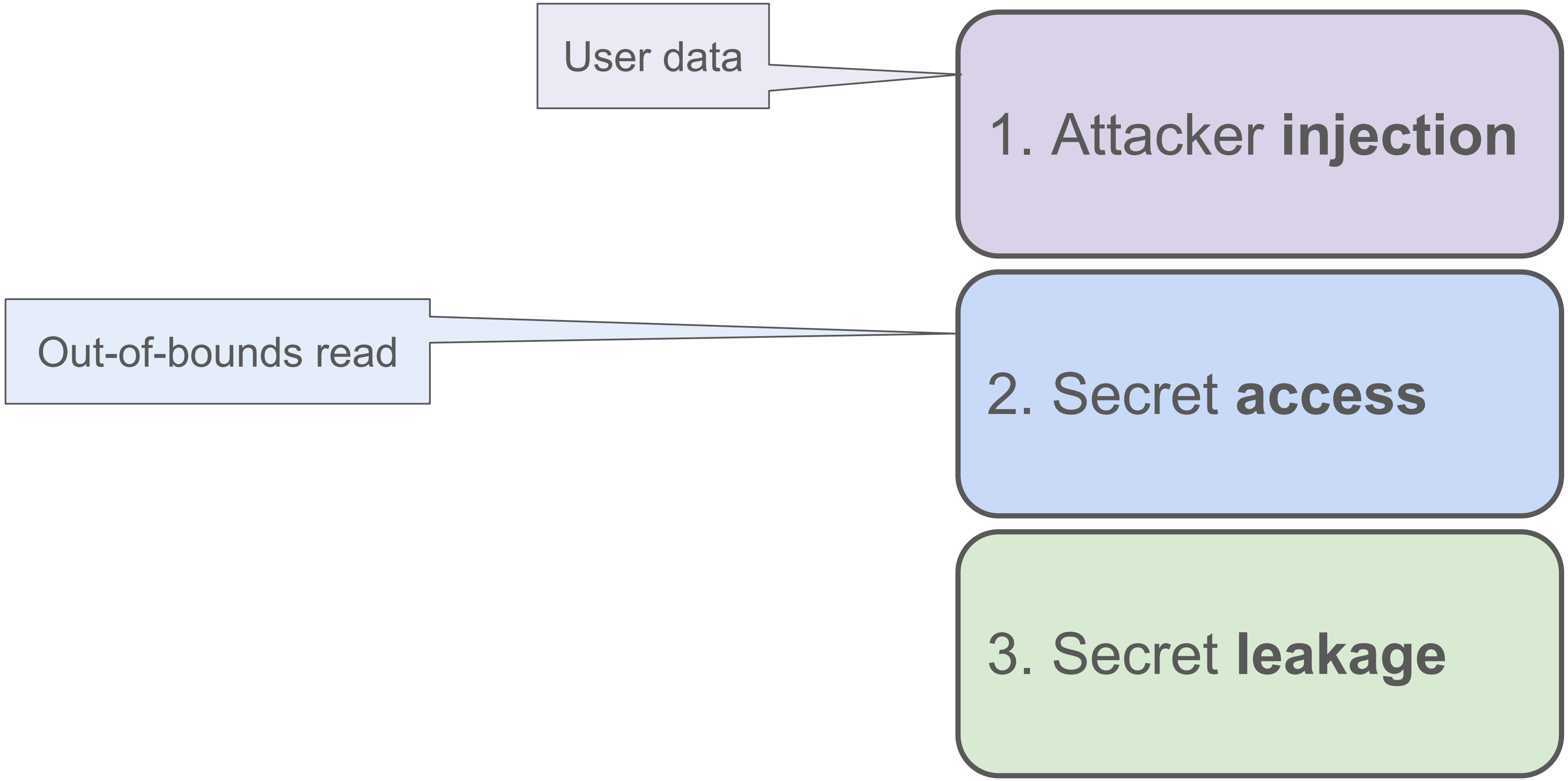
User data

1. Attacker **injection**

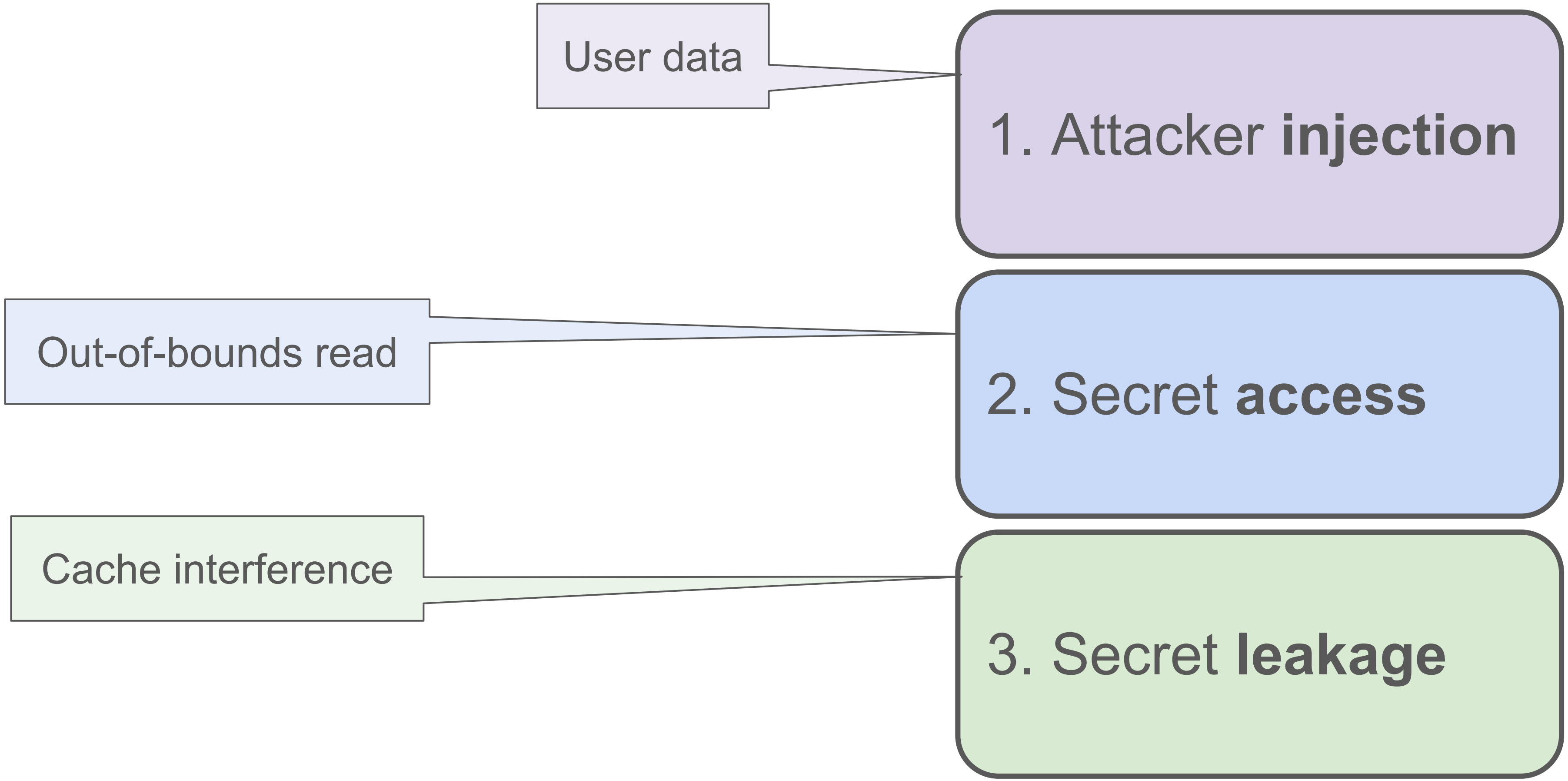
Out-of-bounds read

2. Secret **access**

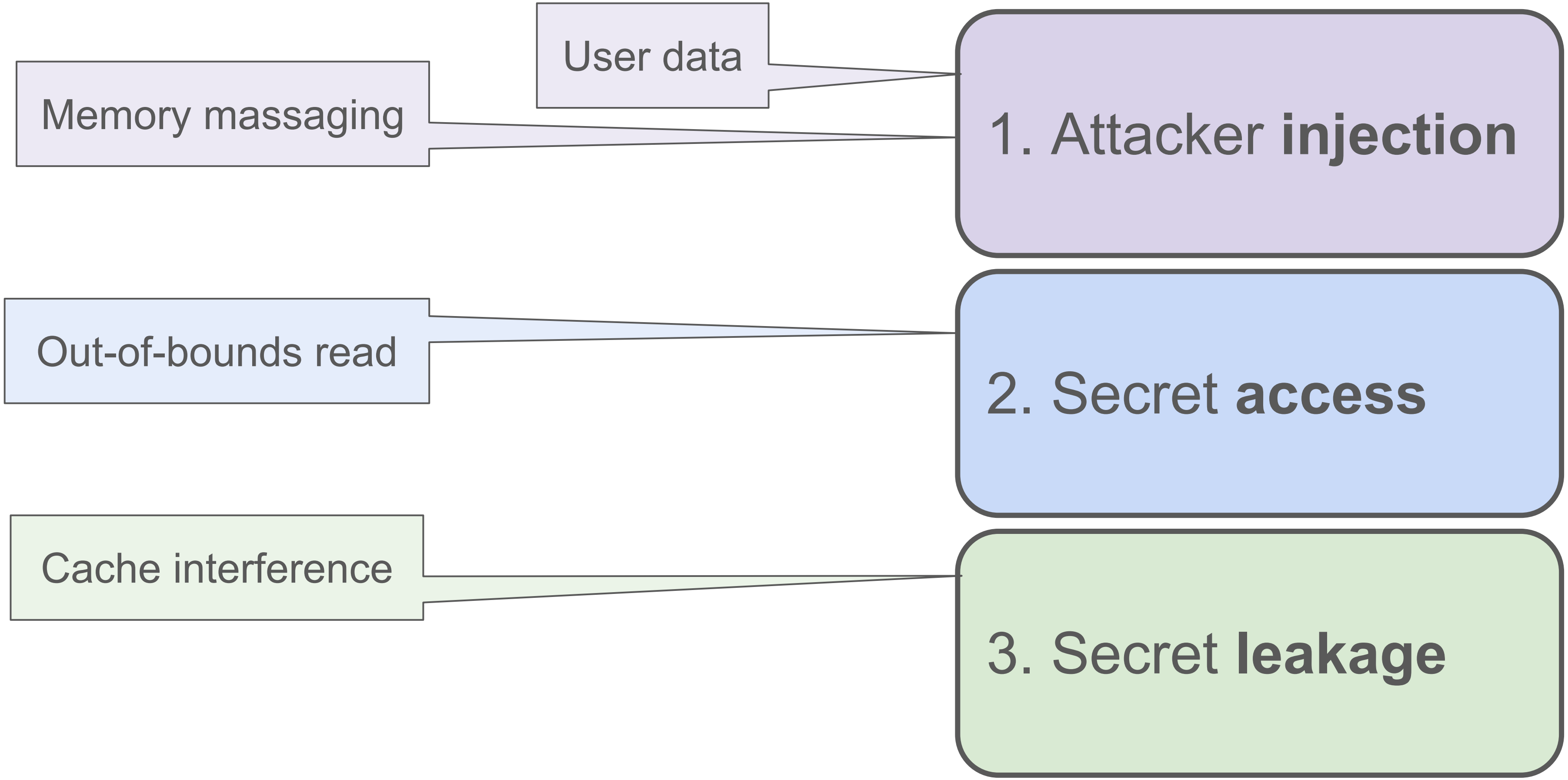
Our approach: Modeling generalized gadgets



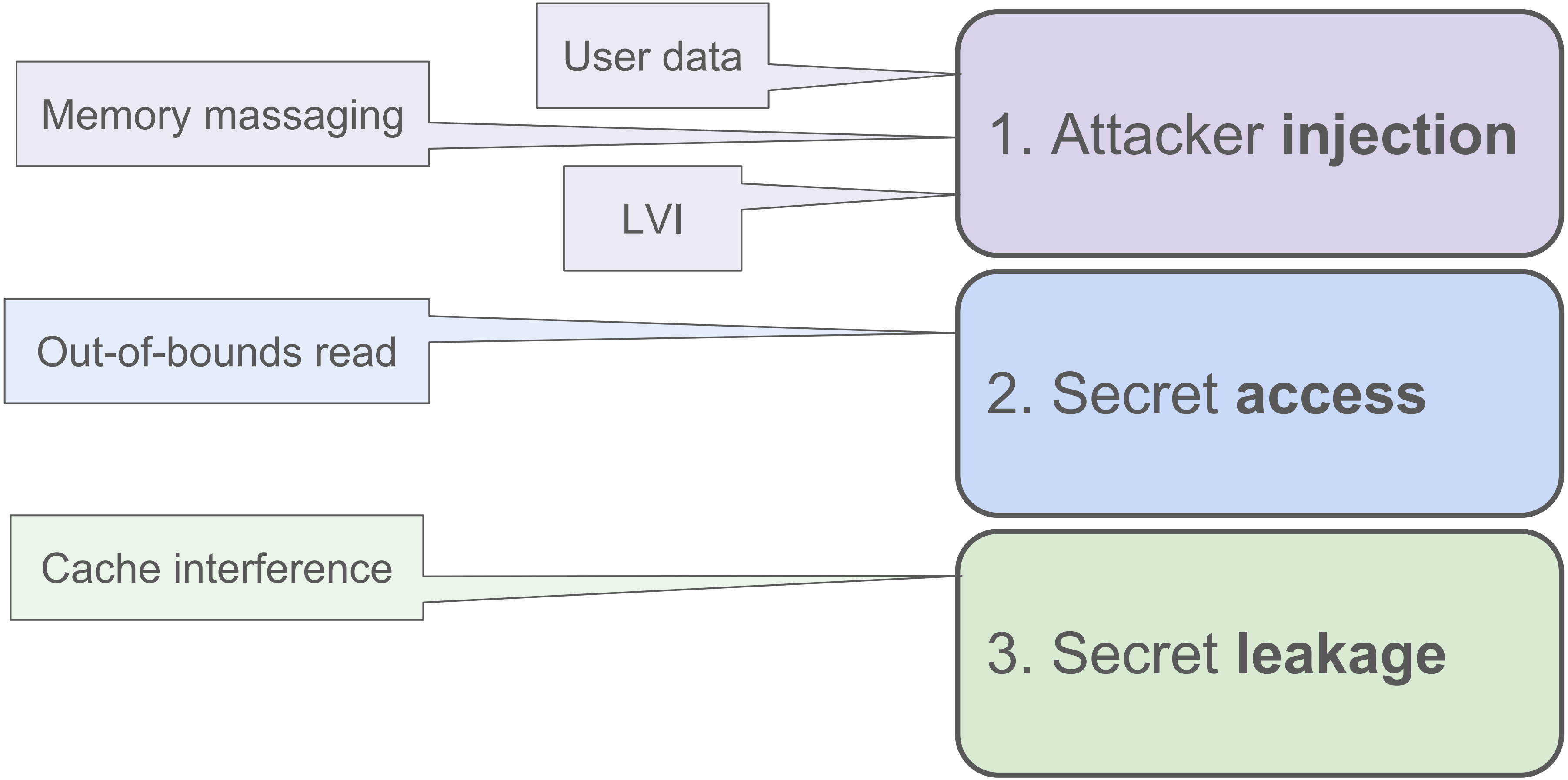
Our approach: Modeling generalized gadgets



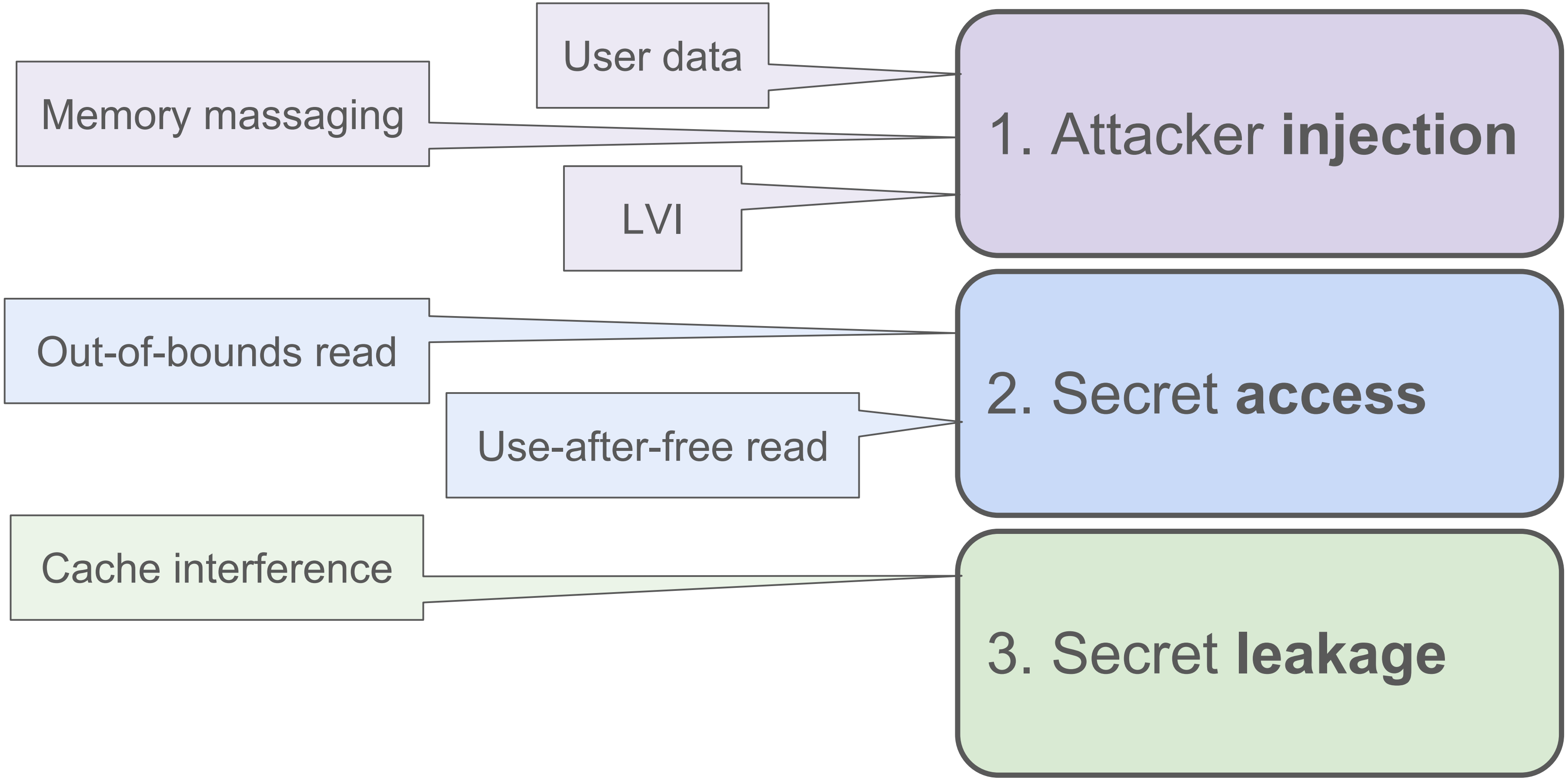
Our approach: Modeling generalized gadgets



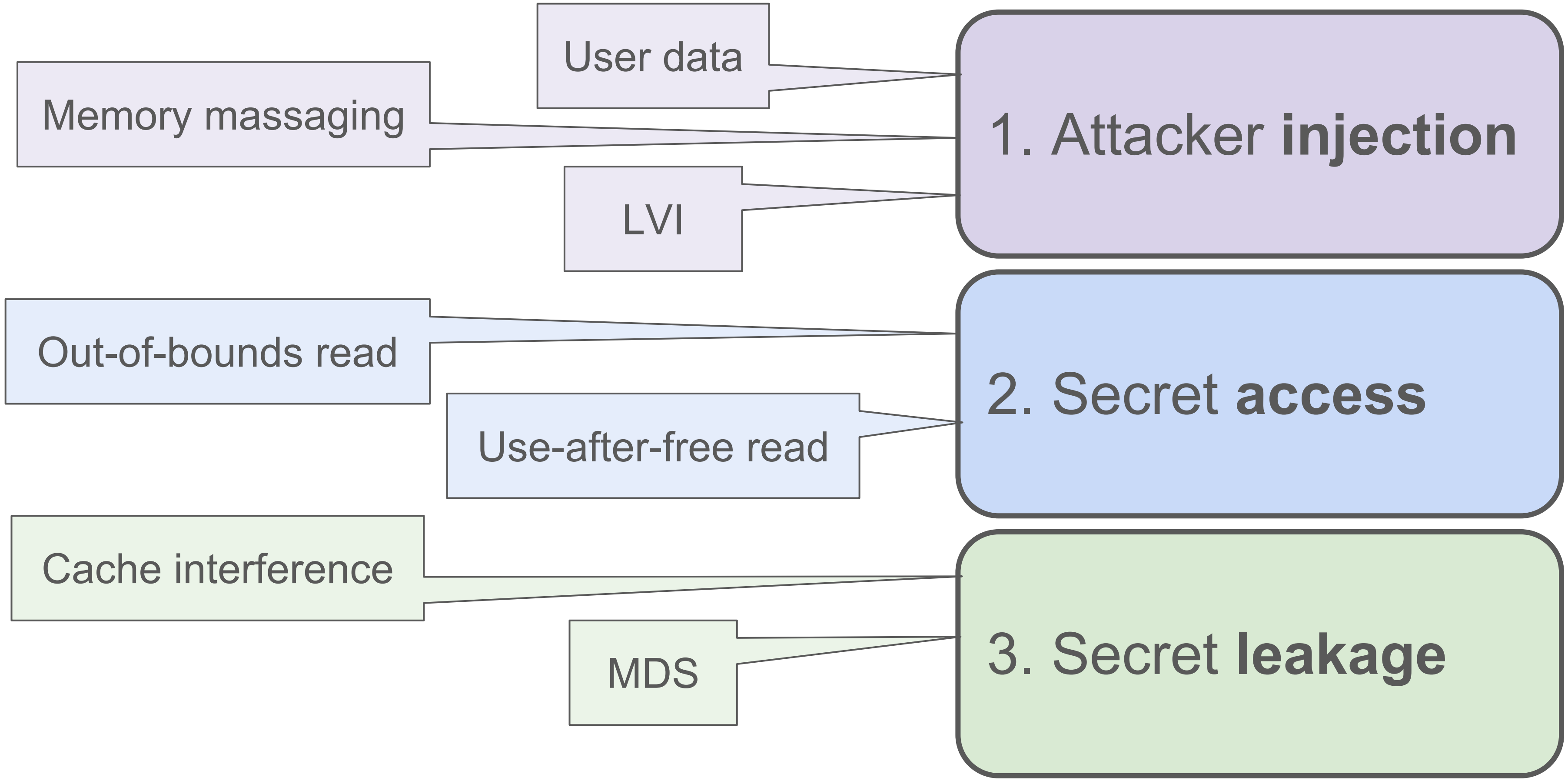
Our approach: Modeling generalized gadgets



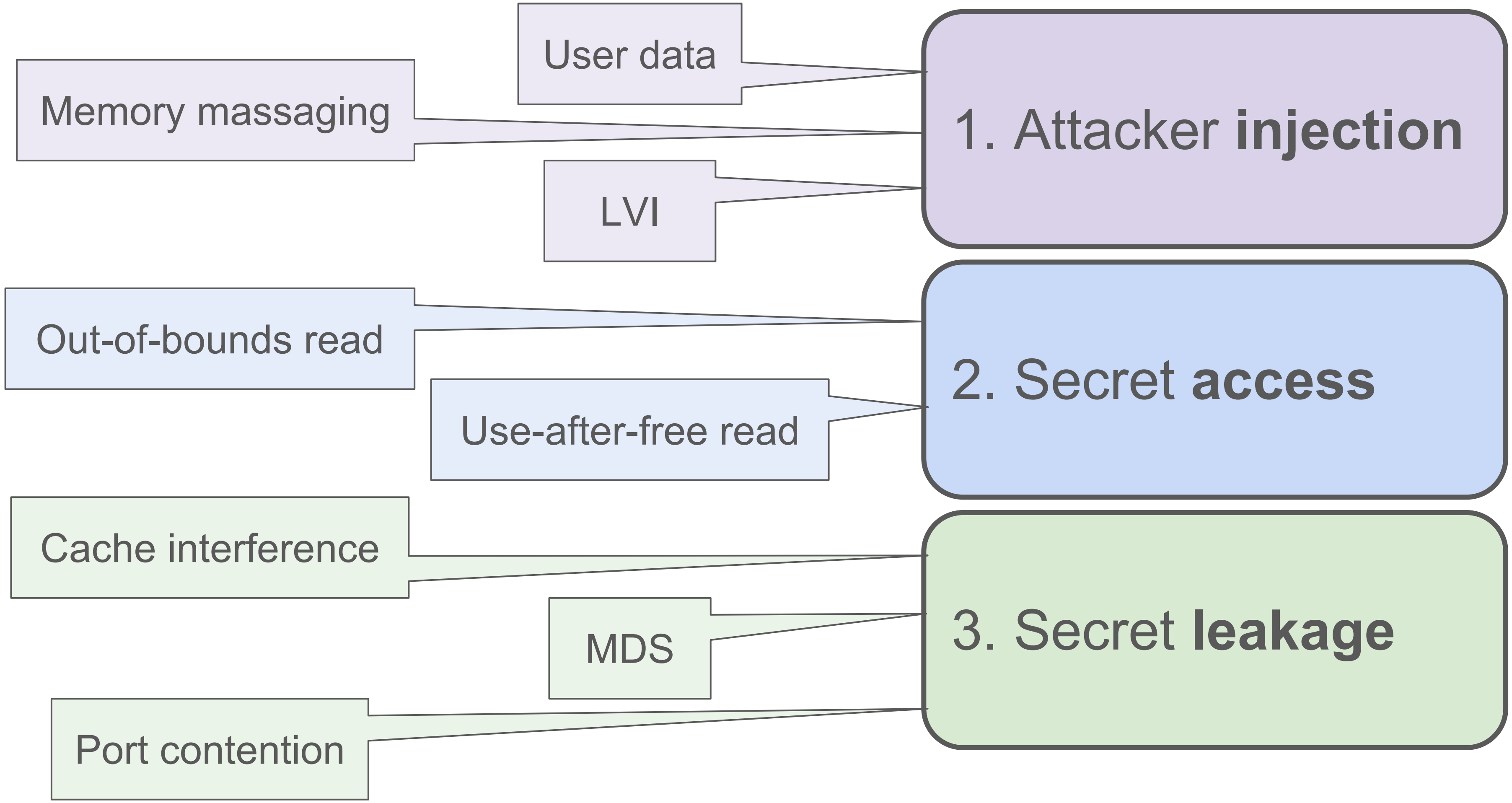
Our approach: Modeling generalized gadgets



Our approach: Modeling generalized gadgets



Our approach: Modeling generalized gadgets



Okay, so how do we actually **identify** these **gadgets**?

Our approach: An example

Our approach: An example

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

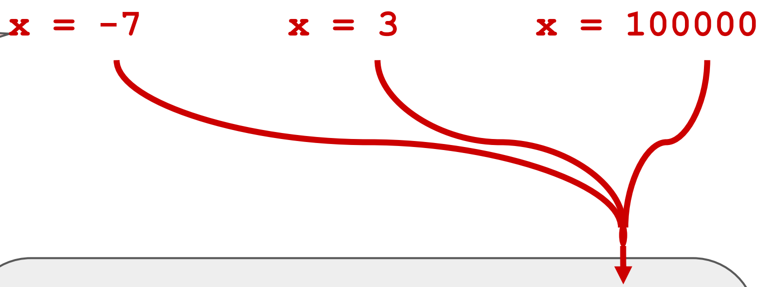
Our approach: An example

1. **Fuzz** the syscall interface

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example

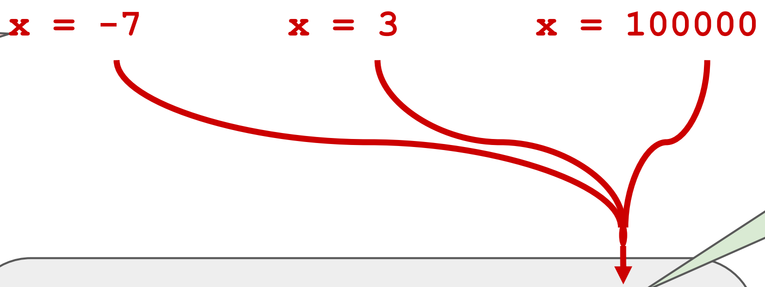
1. **Fuzz** the syscall interface



```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example

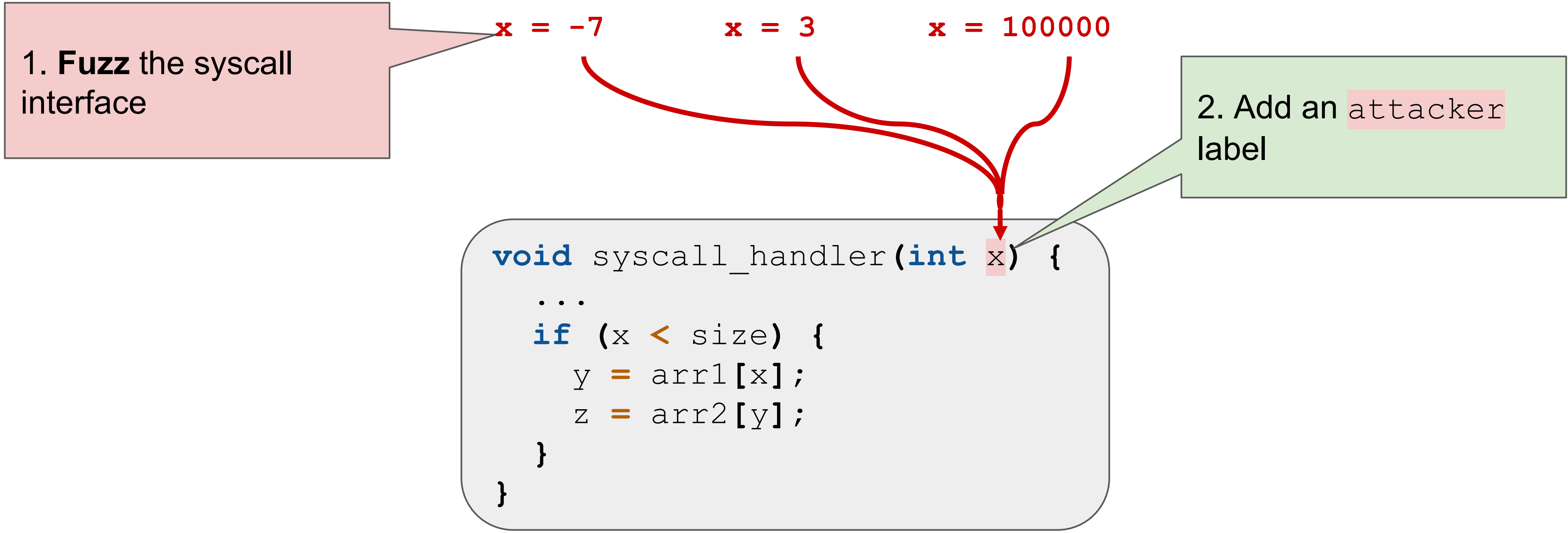
1. **Fuzz** the syscall interface



2. Add an `attacker` label

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example

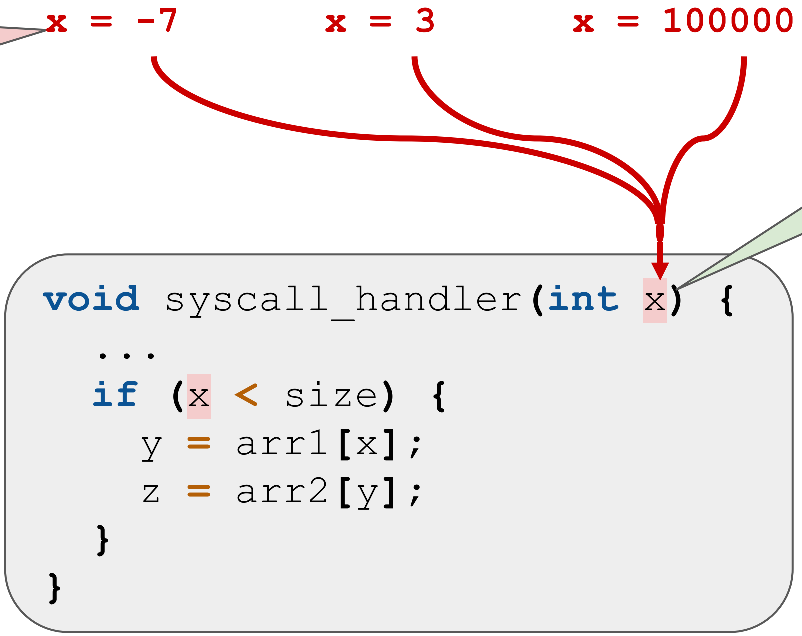


```
void syscall_handler(int x) {
   ...
   if (x < size) {
     y = arr1[x];
     z = arr2[y];
   }
}
```

Our approach: An example

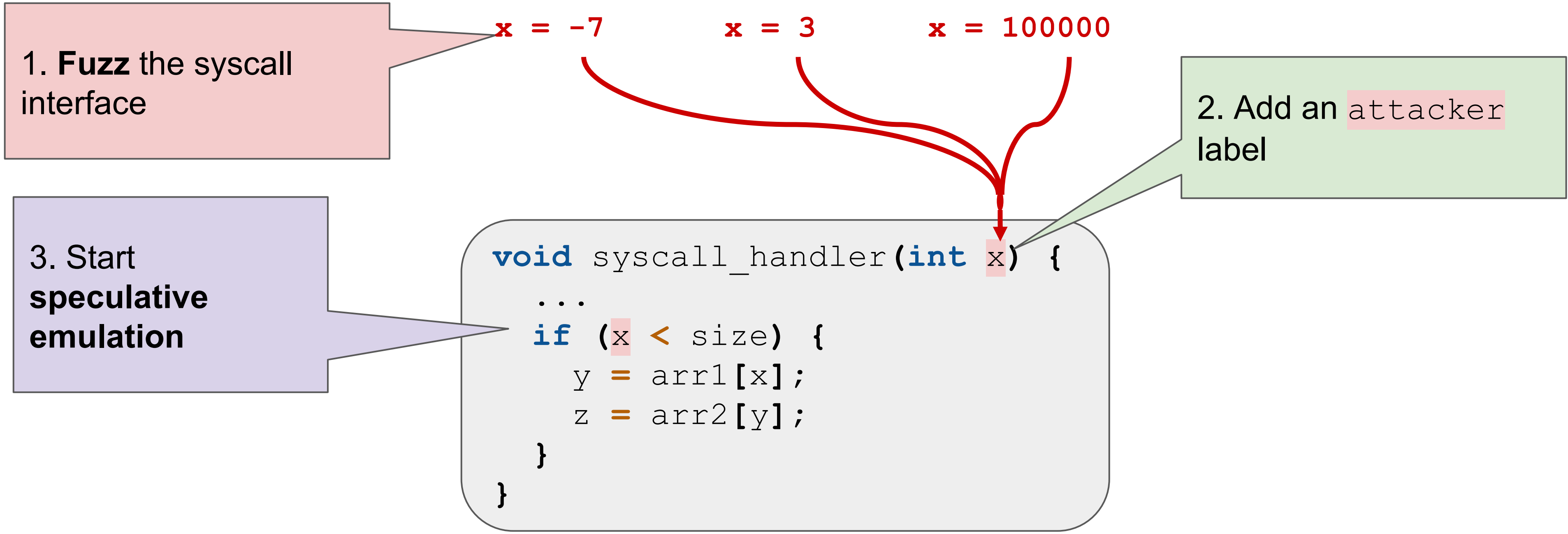
1. **Fuzz** the syscall interface

2. Add an `attacker` label



```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example


1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**

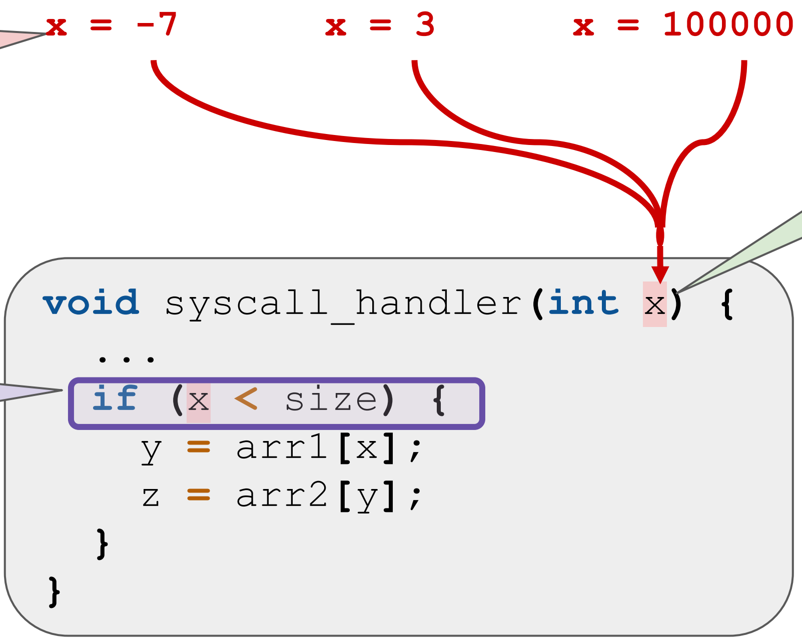
```
void syscall_handler(int x) {
   ...
   if (x < size) {
     y = arr1[x];
     z = arr2[y];
   }
}
```

Our approach: An example

1. **Fuzz** the syscall interface

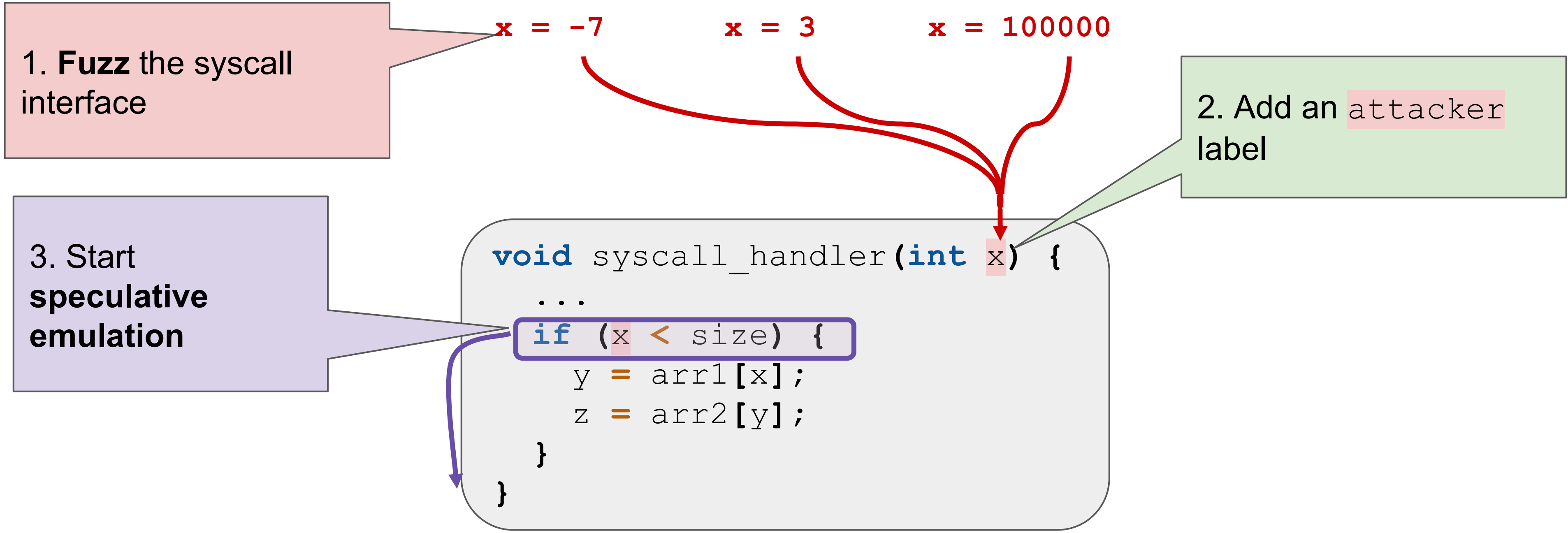
2. Add an `attacker` label

3. Start **speculative emulation**



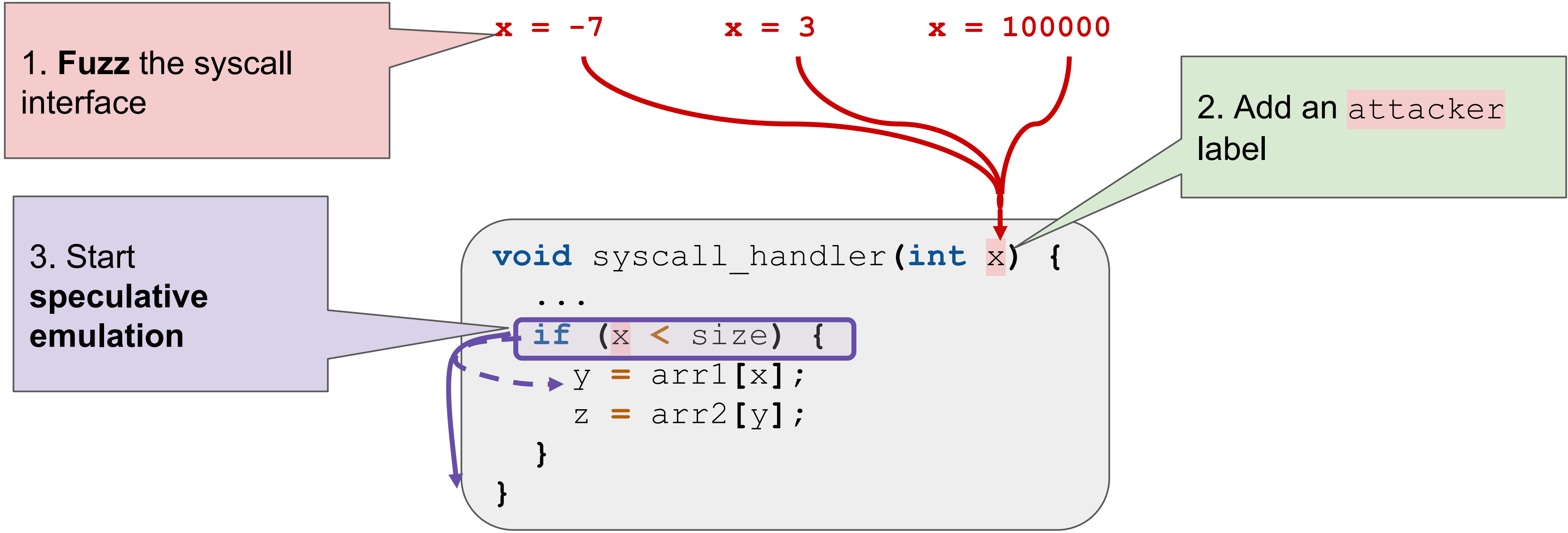
```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example


1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**

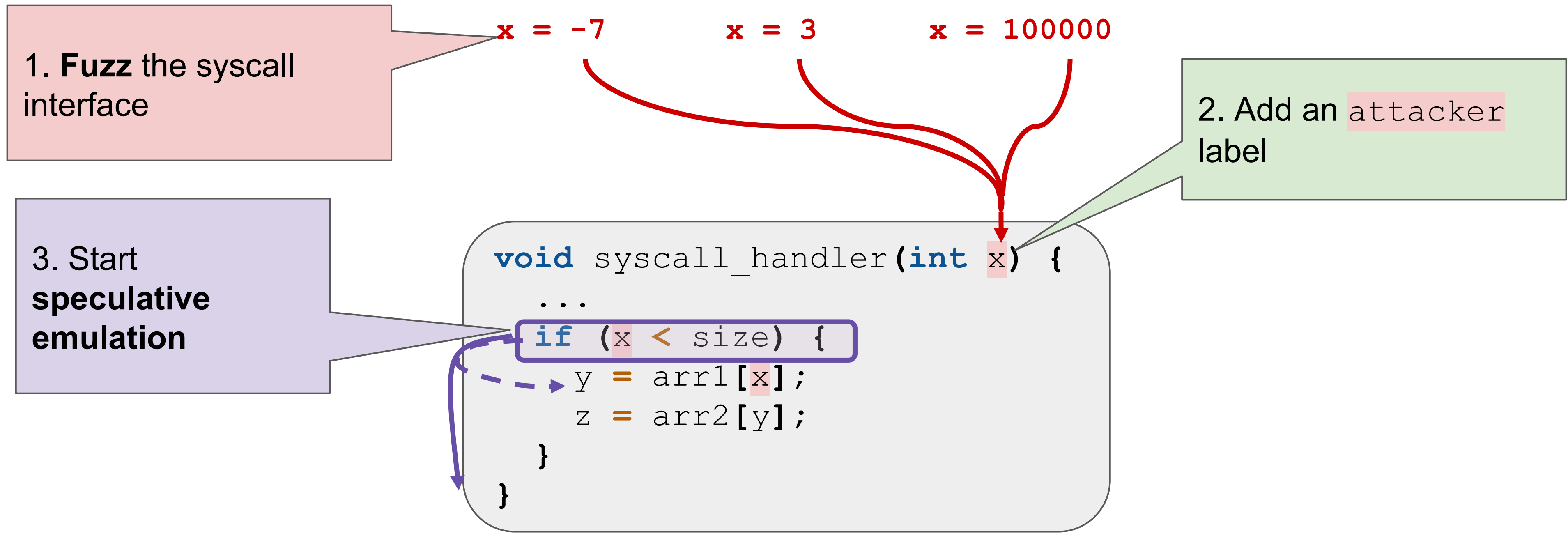
```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example


1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

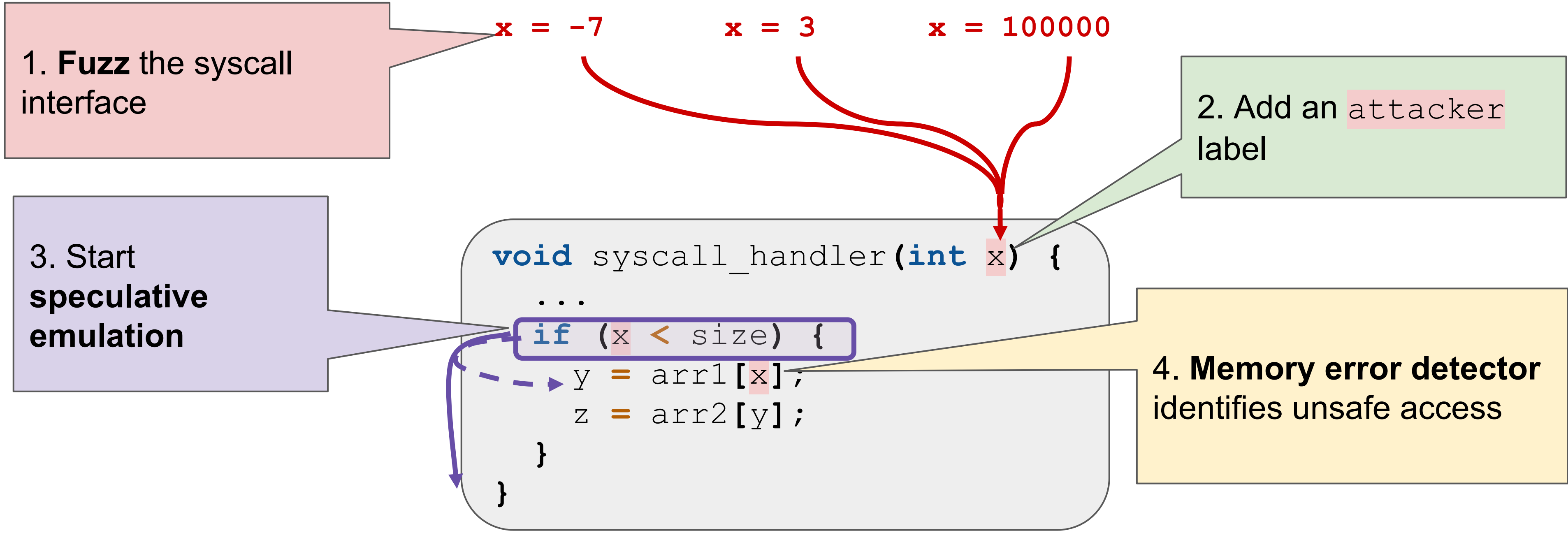
Our approach: An example



1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example



1. **Fuzz** the syscall interface

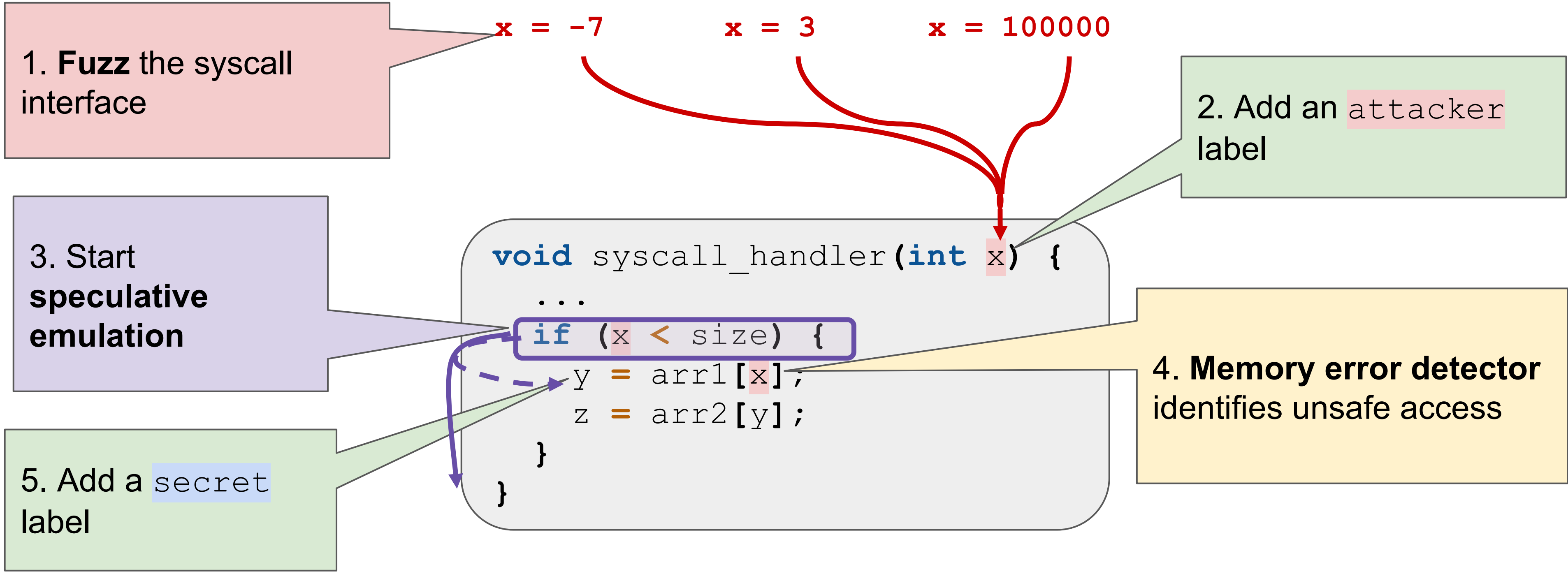
2. Add an `attacker` label

3. Start **speculative emulation**

4. **Memory error detector** identifies unsafe access

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

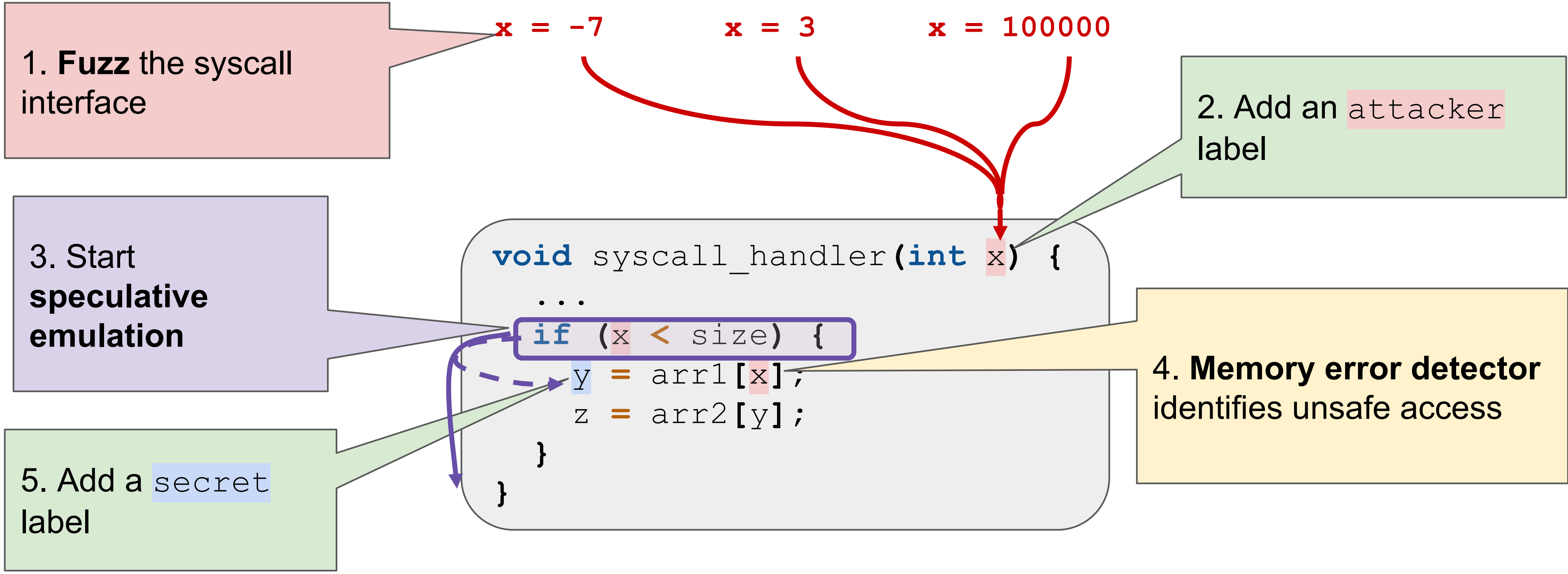
Our approach: An example



1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**
4. **Memory error detector** identifies unsafe access
5. Add a secret label

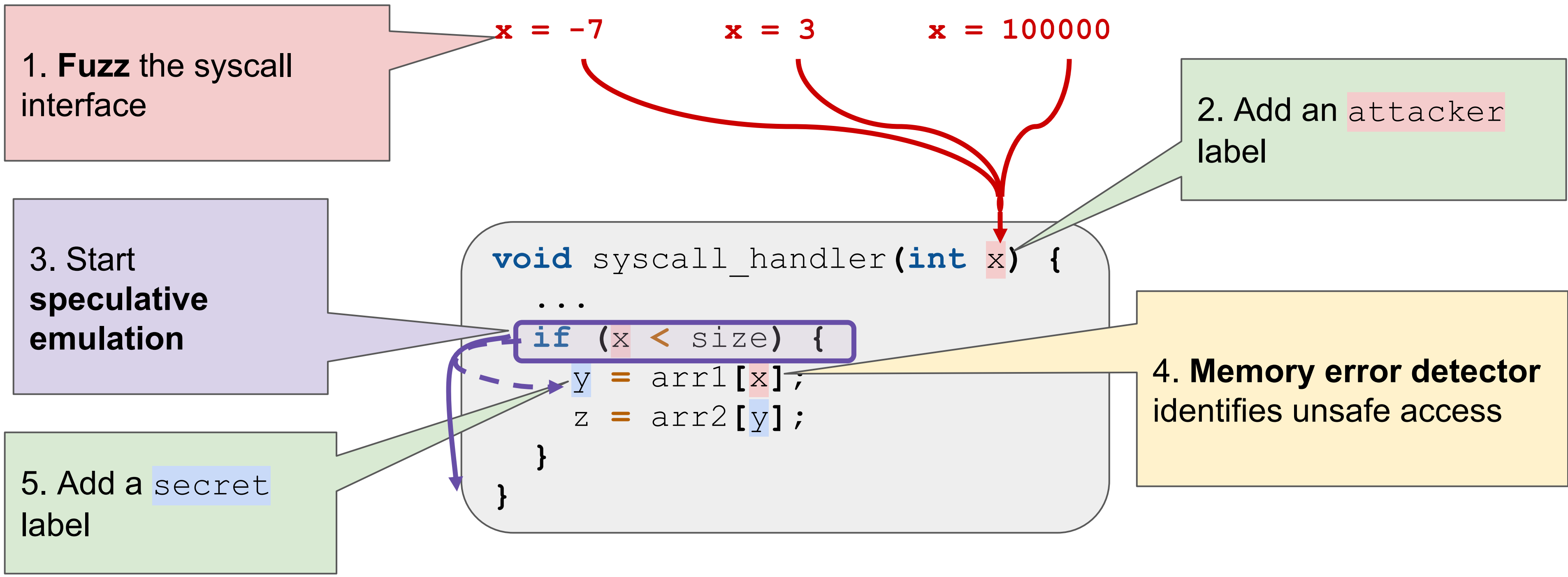
```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example


1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**
4. **Memory error detector** identifies unsafe access
5. Add a secret label

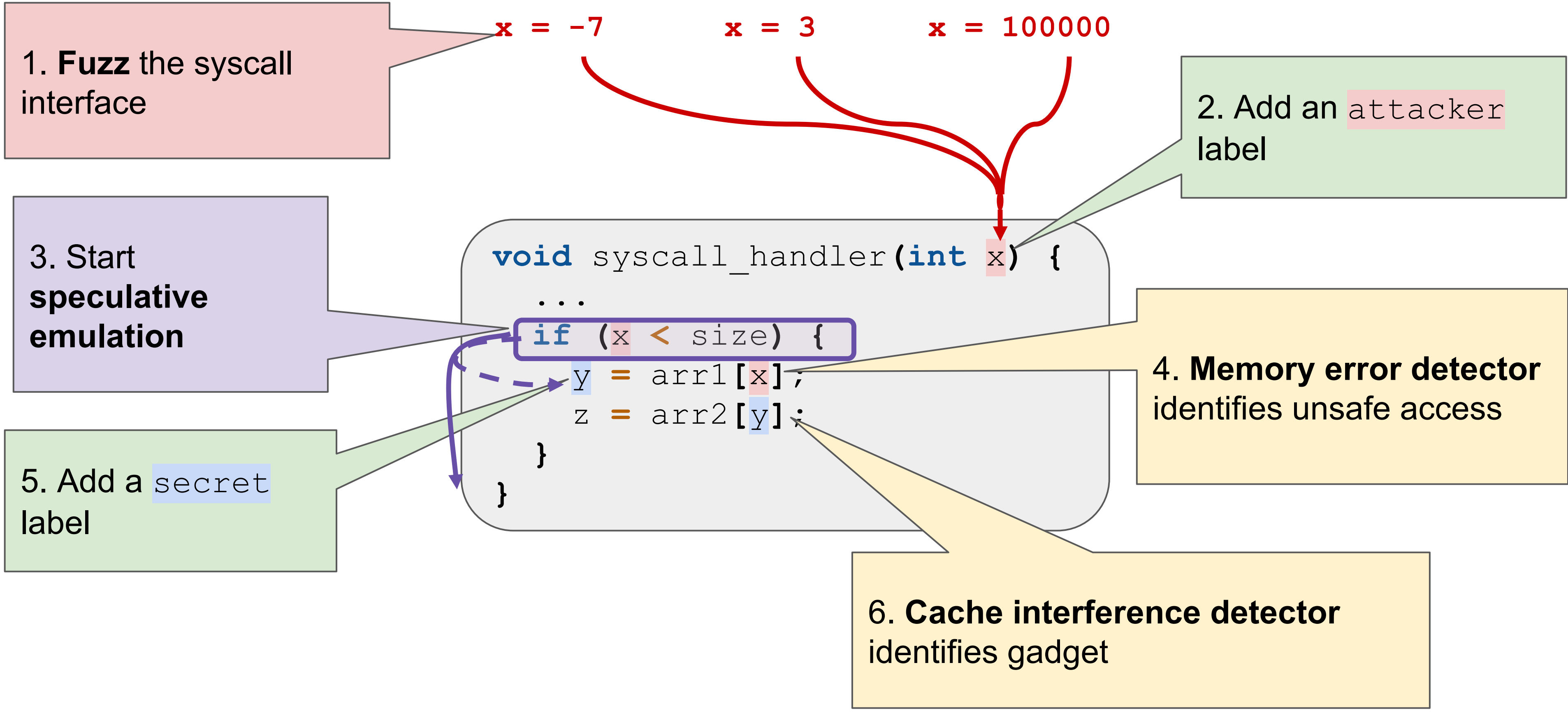
```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example


1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**
4. **Memory error detector** identifies unsafe access
5. Add a secret label

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example



1. **Fuzz** the syscall interface

2. Add an `attacker` label

3. Start **speculative emulation**

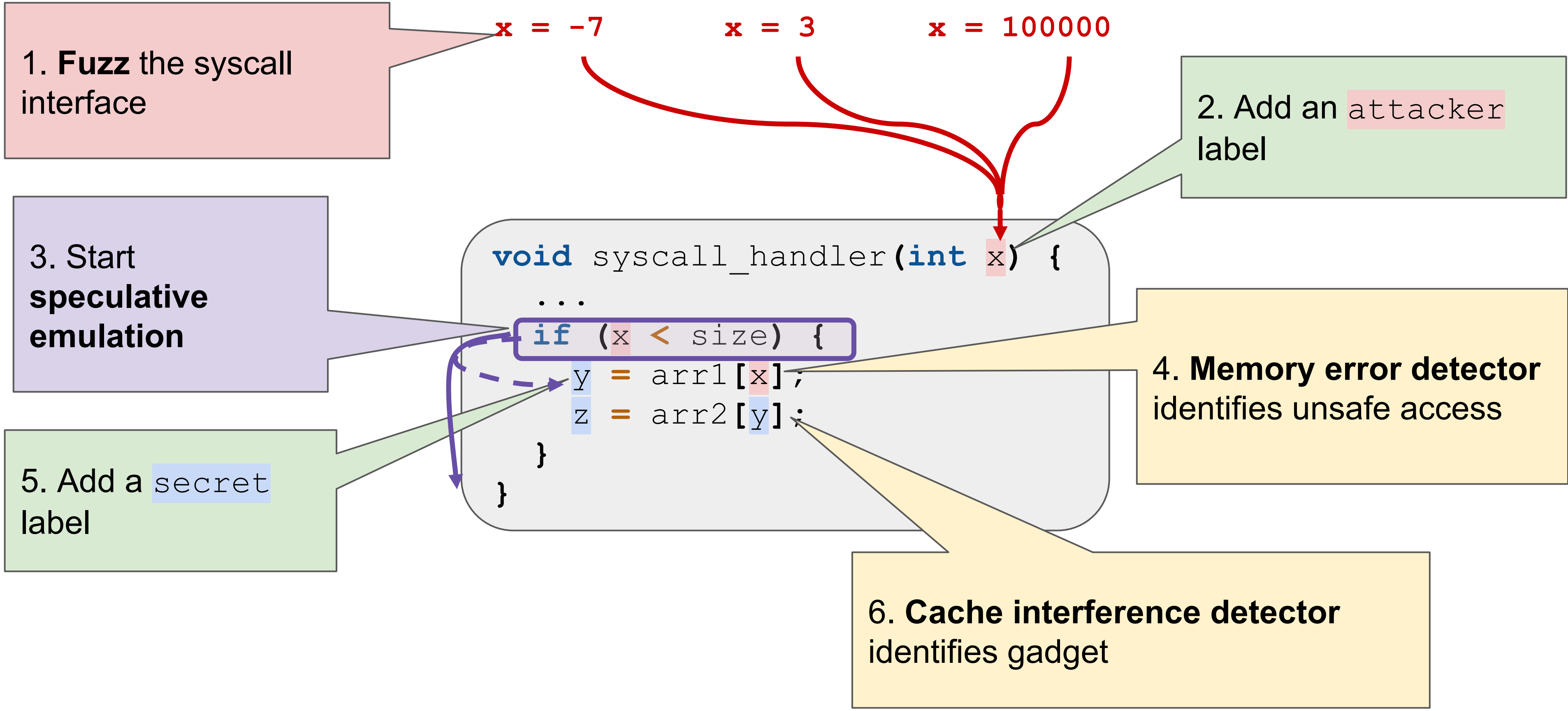
4. **Memory error detector** identifies unsafe access

5. Add a `secret` label

6. **Cache interference detector** identifies gadget

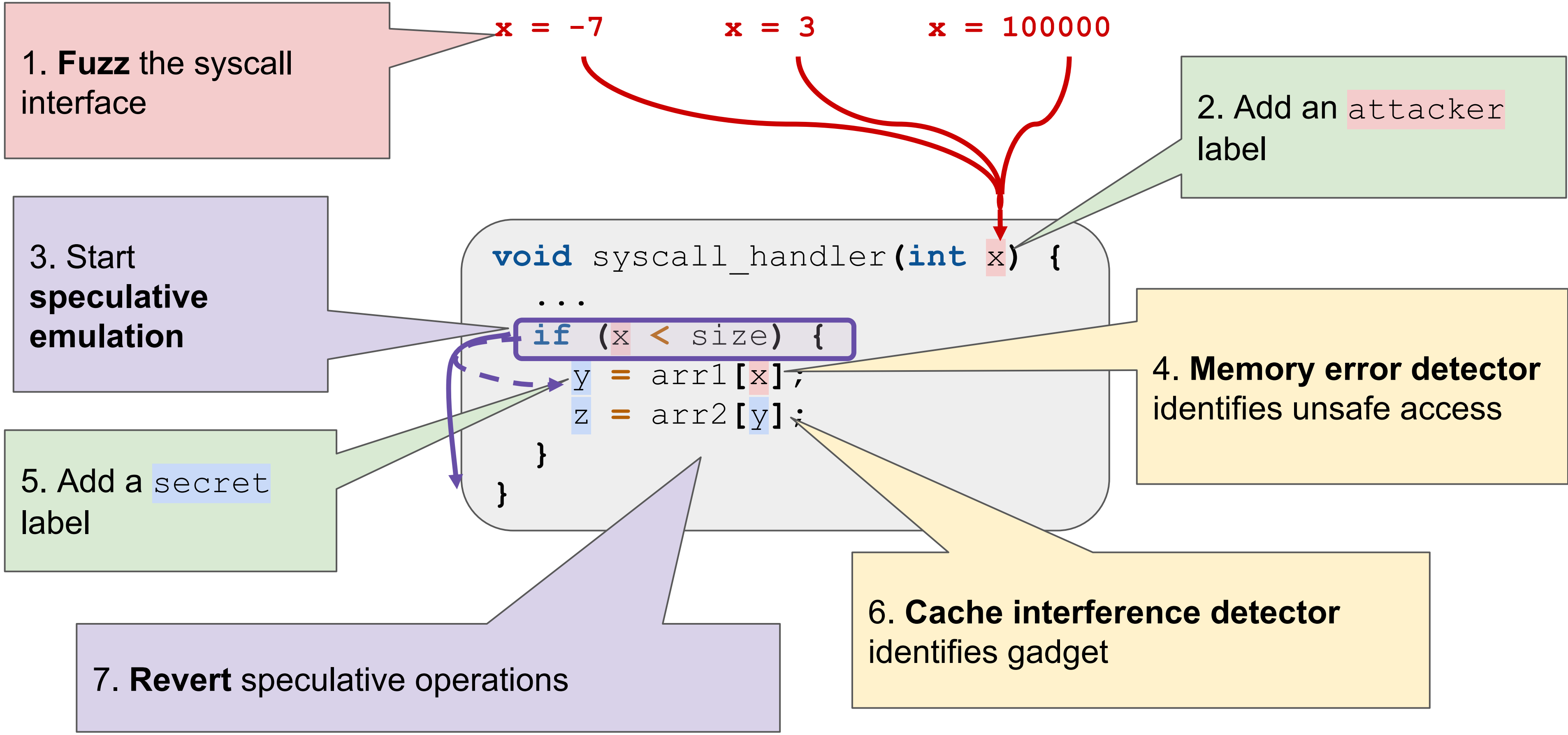
```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example


1. **Fuzz** the syscall interface
x = -7
x = 3
x = 100000
2. Add an attacker label
3. Start **speculative emulation**
4. **Memory error detector** identifies unsafe access
5. Add a secret label
6. **Cache interference detector** identifies gadget

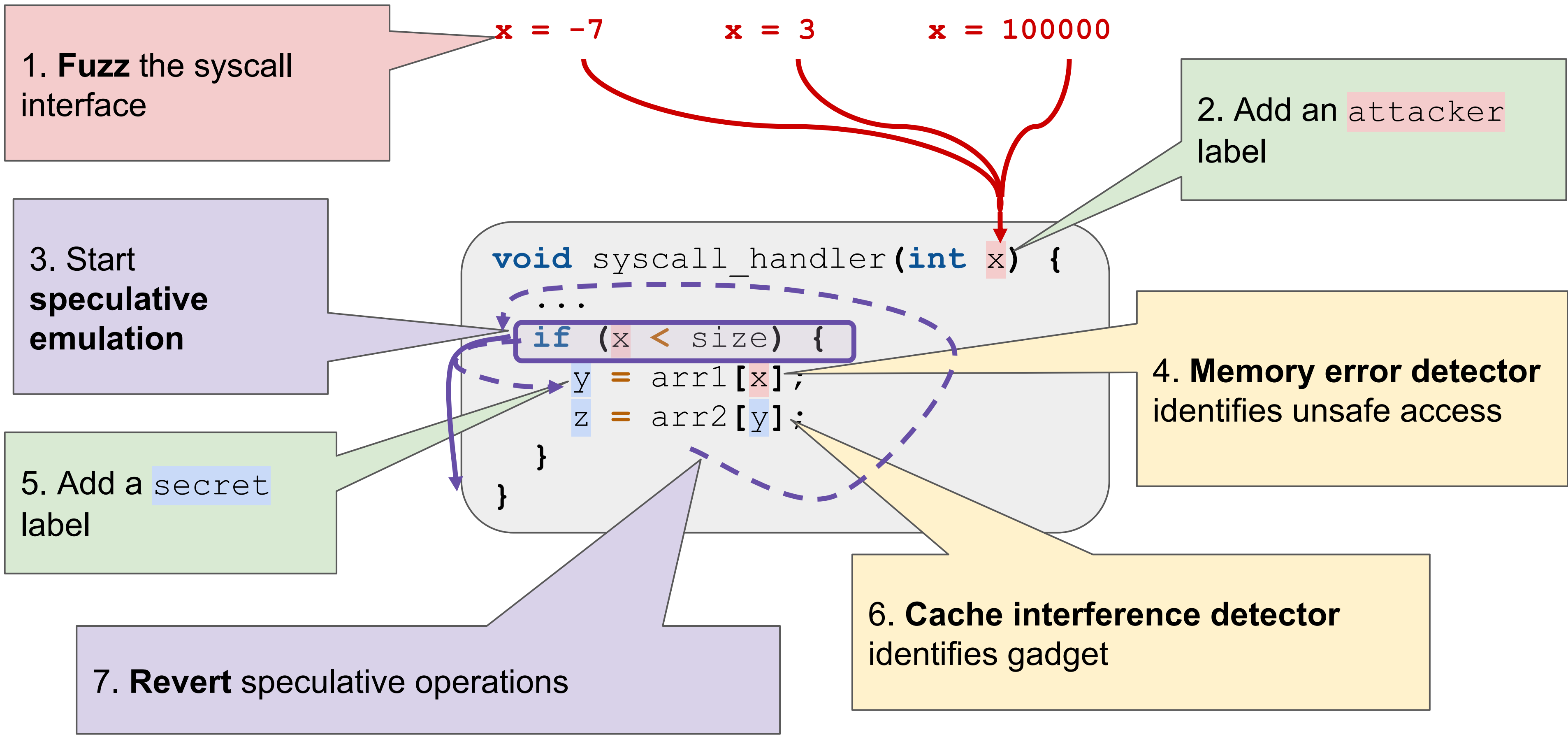
```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example


x = -7
x = 3
x = 100000
1. **Fuzz** the syscall interface
2. Add an attacker label
3. Start **speculative emulation**
4. **Memory error detector** identifies unsafe access
5. Add a secret label
6. **Cache interference detector** identifies gadget
7. **Revert** speculative operations

```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

Our approach: An example



```
void syscall_handler(int x) {
  ...
  if (x < size) {
    y = arr1[x];
    z = arr2[y];
  }
}
```

x = -7
x = 3
x = 100000
1. **Fuzz** the syscall interface
2. Add an attacker label
3. Start **speculative emulation**
4. **Memory error detector** identifies unsafe access
5. Add a secret label
6. **Cache interference detector** identifies gadget
7. **Revert** speculative operations

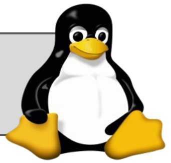
Our implementation: KASPER

Our implementation: KASPER

Linux kernel

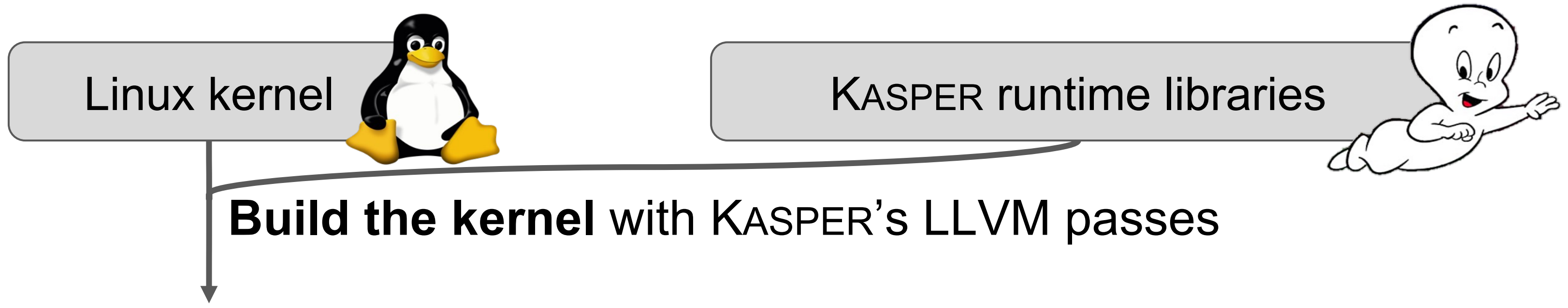
Our implementation: KASPER

Linux kernel

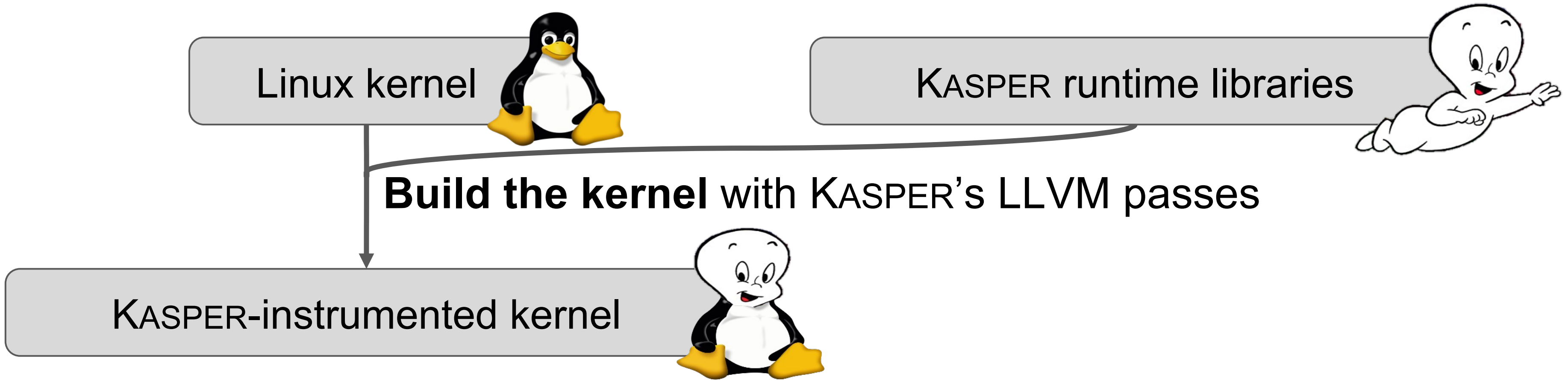


KASPER runtime libraries

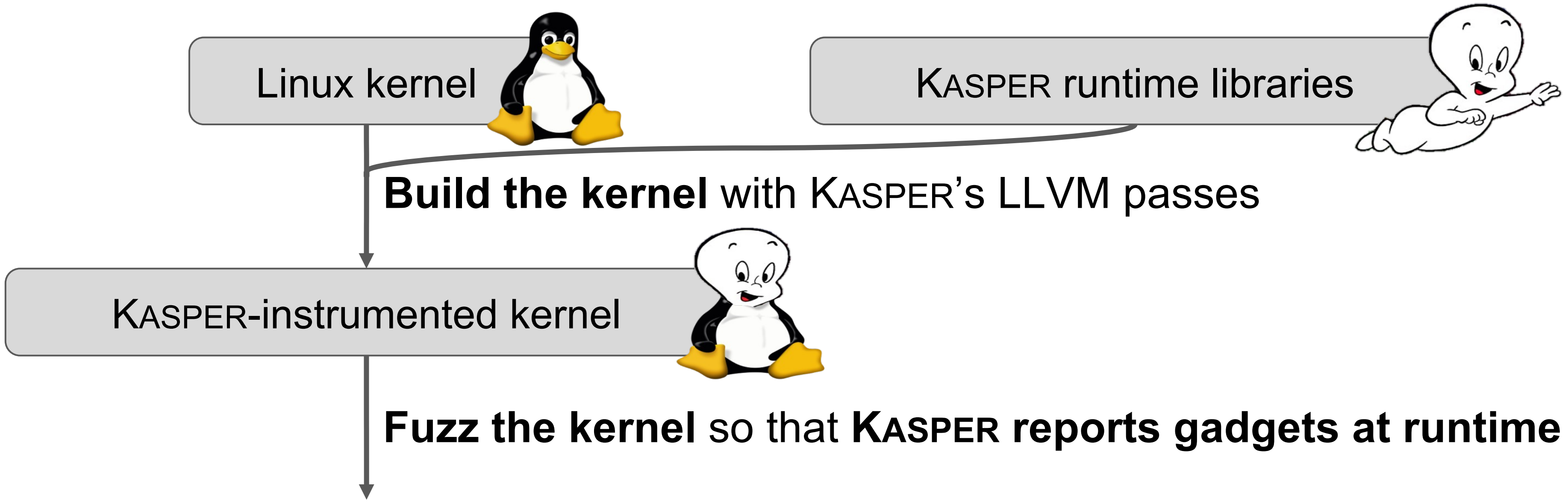
Our implementation: KASPER



Our implementation: KASPER


Linux kernel
KASPER runtime libraries
Build the kernel with KASPER’s LLVM passes
KASPER-instrumented kernel

Our implementation: KASPER



Linux kernel

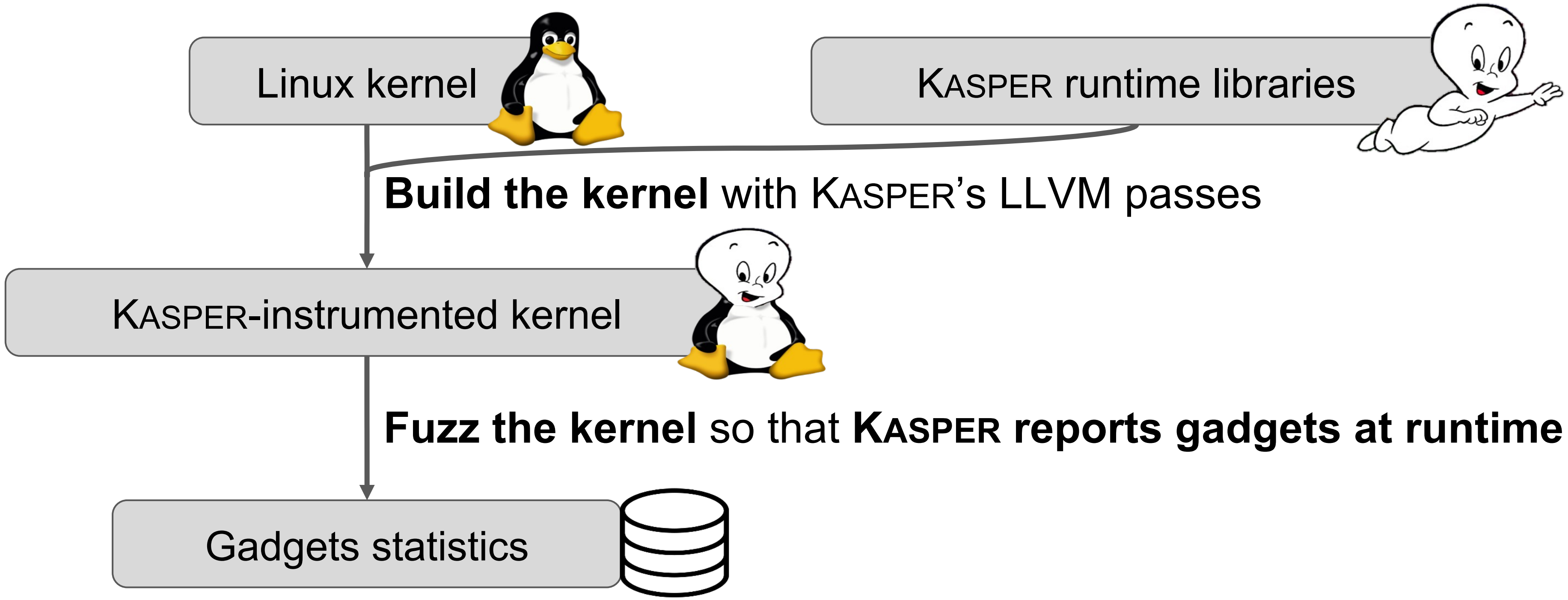
KASPER runtime libraries

Build the kernel with KASPER's LLVM passes

KASPER-instrumented kernel

Fuzz the kernel so that **KASPER reports gadgets at runtime**

Our implementation: KASPER



Build the kernel with KASPER's LLVM passes

Fuzz the kernel so that **KASPER reports gadgets at runtime**

Comparison with previous approaches

Comparison with previous approaches

Gadgets reported in the kernel when running UNIX's `ls` command

Comparison with previous approaches

Gadgets reported in the kernel when running UNIX’s `ls` command

	Total gadgets reported	**FP rate**	**FN rate (“Spectre-V1” only)**
SpecFuzz	662	99%	33%
SpecTaint	688	99%	0%
KASPER (“Spectre-V1” only)	8	25%	0%

Comparison with previous approaches

Gadgets reported in the kernel when running UNIX's `ls` command

	Total gadgets reported	**FP rate**	**FN rate ("Spectre-V1" only)**
SpecFuzz	662	99%	33%
SpecTaint	688	99%	0%
KASPER ("Spectre-V1" only)	8	25%	0%

Only targets speculative **out-of-bounds accesses**

Comparison with previous approaches

Gadgets reported in the kernel when running UNIX's `ls` command

	Total gadgets reported	**FP rate**	**FN rate ("Spectre-V1" only)**
SpecFuzz	662	99%	33%
SpecTaint	688	99%	0%
KASPER ("Spectre-V1" only)	8	25%	0%

Only targets **attacker-dependent accesses**

Only targets speculative **out-of-bounds accesses**

Comparison with previous approaches

Gadgets reported in the kernel when running UNIX's `ls` command

	Total gadgets reported	FP rate	FN rate ("Spectre-V1" only)
SpecFuzz	662	99%	33%
SpecTaint	688	99%	0%
KASPER ("Spectre-V1" only)	8	25%	0%

Only targets **attacker-dependent accesses**

Non-deterministic **overtainting**

Only targets speculative **out-of-bounds accesses**

Comparison with previous approaches

Gadgets reported in the kernel when running UNIX's `ls` command

	Total gadgets reported	FP rate	FN rate ("Spectre-V1" only)
SpecFuzz	662	99%	33%
SpecTaint	688	99%	0%
KASPER ("Spectre-V1" only)	8	25%	0%

Fewer, **higher quality** gadgets

Only targets **attacker-dependent accesses**

Non-deterministic **overtainting**

Only targets speculative **out-of-bounds accesses**

Gadgets discovered

Gadgets discovered

Gadget type	Number of reports
USER-CACHE	147
MASSAGE-CACHE	47
LVI-CACHE	12
USER-MDS	600
MASSAGE-MDS	193
USER-PORT	407
MASSAGE-PORT	123
Total	**1379**

Gadgets discovered

Gadget type	Number of reports
USER-CACHE	147
MASSAGE-CACHE	47
LVI-CACHE	12
USER-MDS	600
MASSAGE-MDS	193
USER-PORT	407
MASSAGE-PORT	123
Total	**1379**

The original **“Spectre-V1”** remains **largely unmitigated**.

Gadgets discovered

Gadget type	Number of reports
USER-CACHE	147
MASSAGE-CACHE	47
LVI-CACHE	12
USER-MDS	600
MASSAGE-MDS	193
USER-PORT	407
MASSAGE-PORT	123
Total	**1379**

The original **“Spectre-V1”** remains **largely unmitigated**.

LVI is indeed an issue from a conditional branch misprediction.

Gadgets discovered

Gadget type	Number of reports
USER-CACHE	147
MASSAGE-CACHE	47
LVI-CACHE	12
USER-MDS	600
MASSAGE-MDS	193
USER-PORT	407
MASSAGE-PORT	123
Total	**1379**

The original **"Spectre-V1"** remains **largely unmitigated**.

LVI is indeed an issue from a conditional branch misprediction. 👀

Transient **memory massaging** is a **legitimate attack vector**. 👀

Gadgets discovered

Gadget type	Number of reports
USER-CACHE	147
MASSAGE-CACHE	47
LVI-CACHE	12
USER-MDS	600
MASSAGE-MDS	193
USER-PORT	407
MASSAGE-PORT	123
Total	**1379**

The original **“Spectre-V1”** remains **largely unmitigated**.

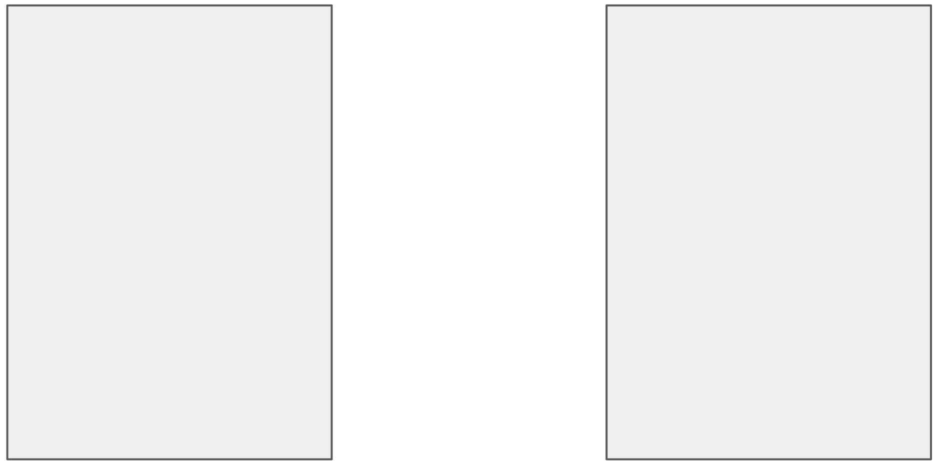
LVI is indeed an issue from a conditional branch misprediction. 👀

Transient **memory massaging** is a **legitimate attack vector**. 👀

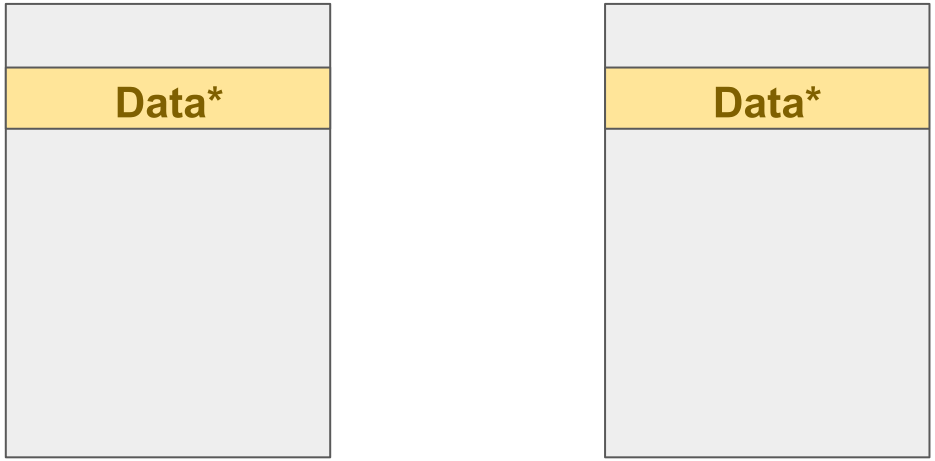
There are a ton of gadgets! But are any of them **actually exploitable**…

Case study: Linux’s list iterator

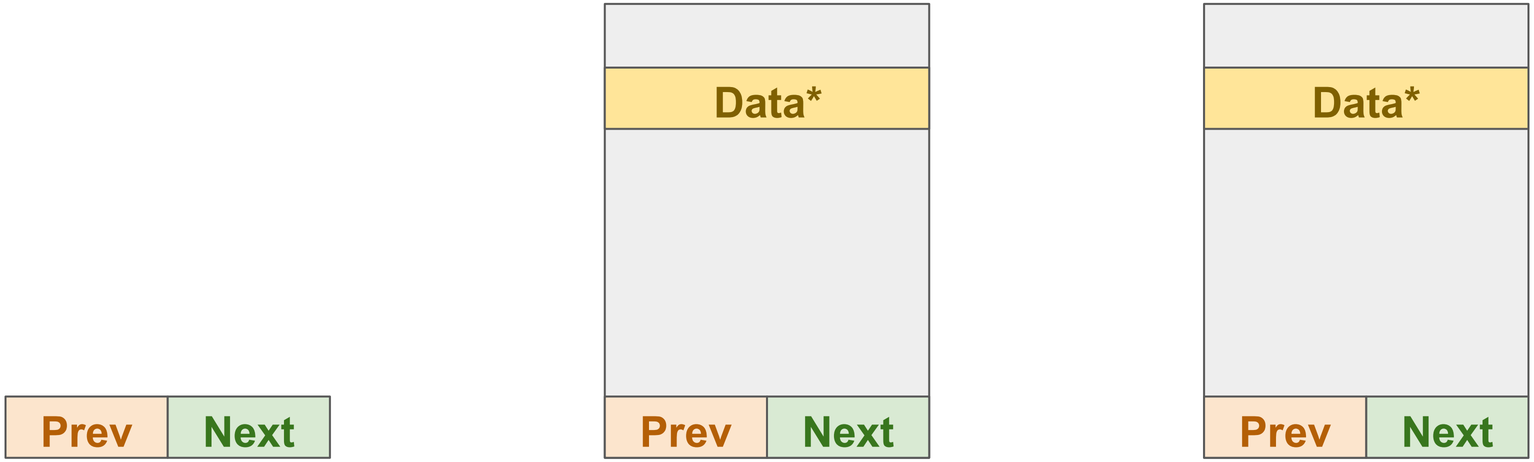
Case study: Linux’s list iterator



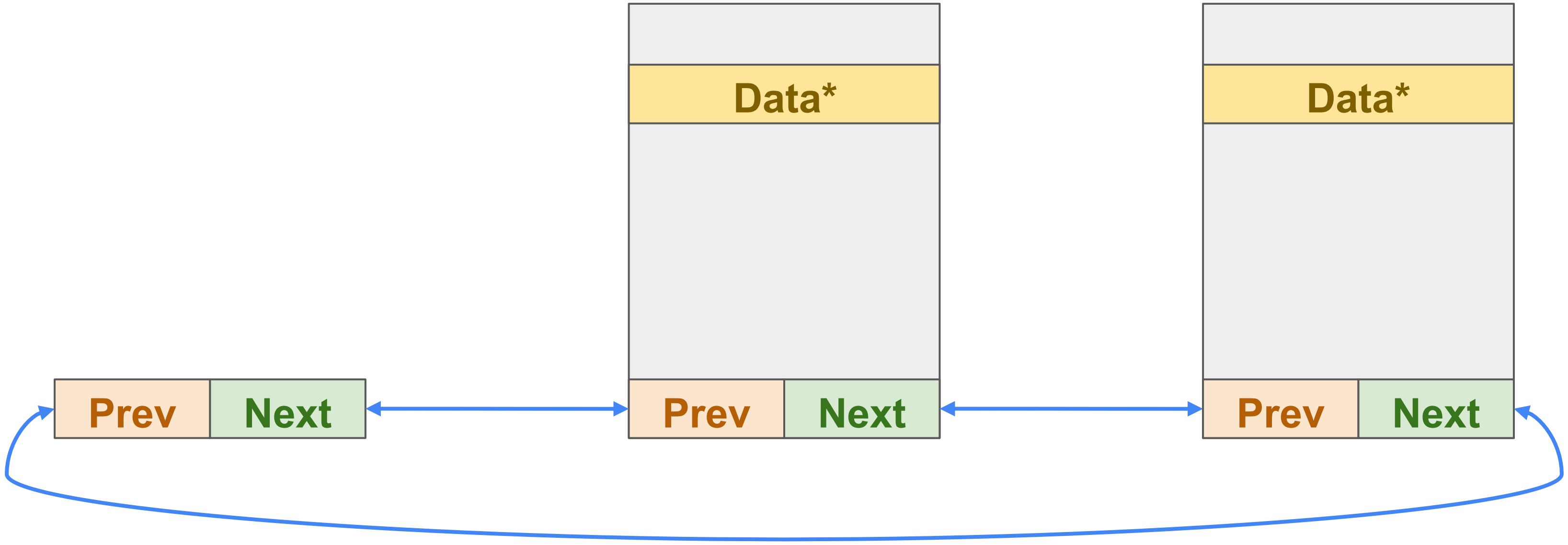
Case study: Linux's list iterator



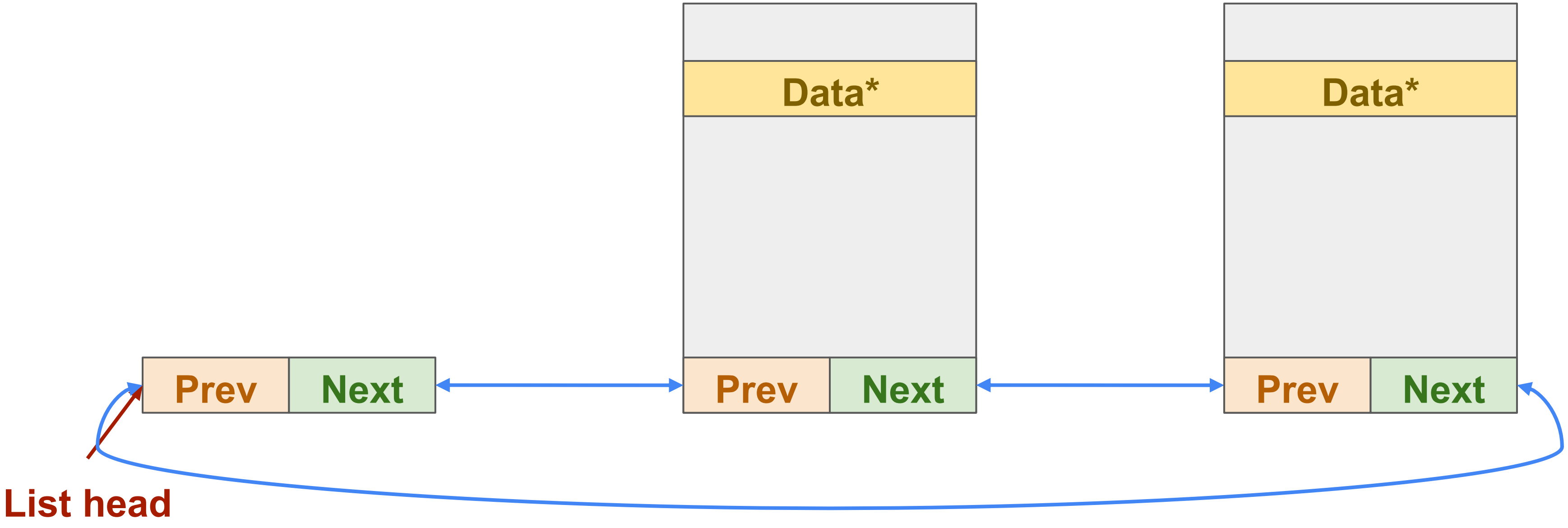
Case study: Linux's list iterator



Case study: Linux's list iterator

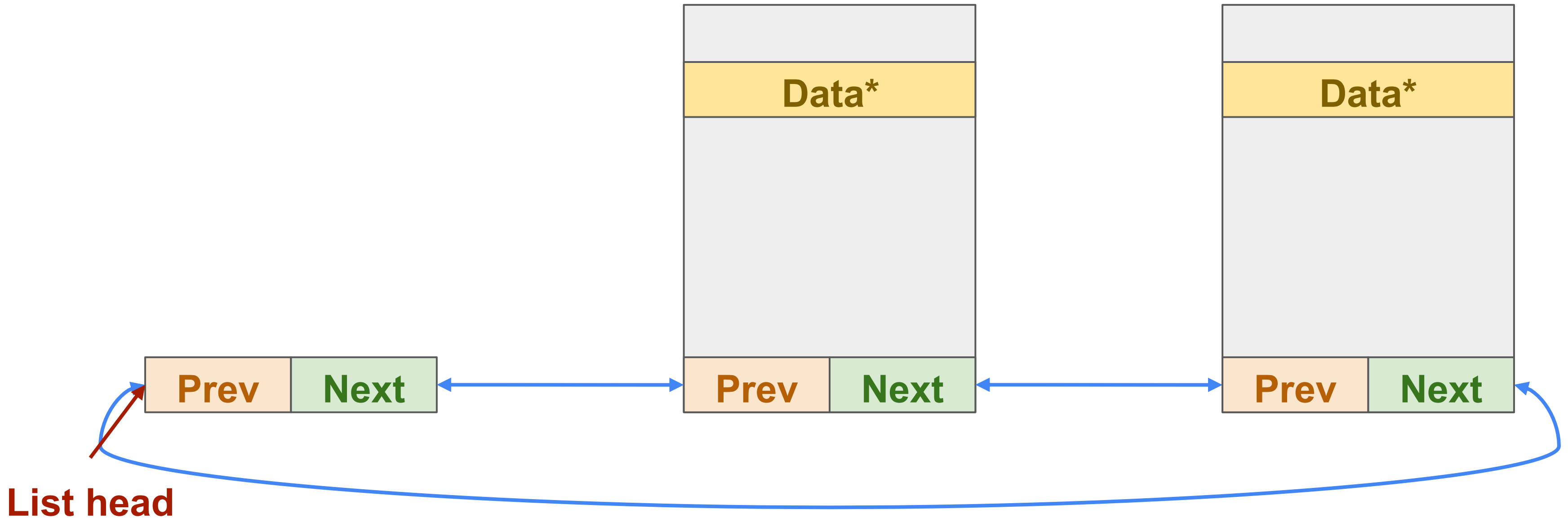


Case study: Linux's list iterator



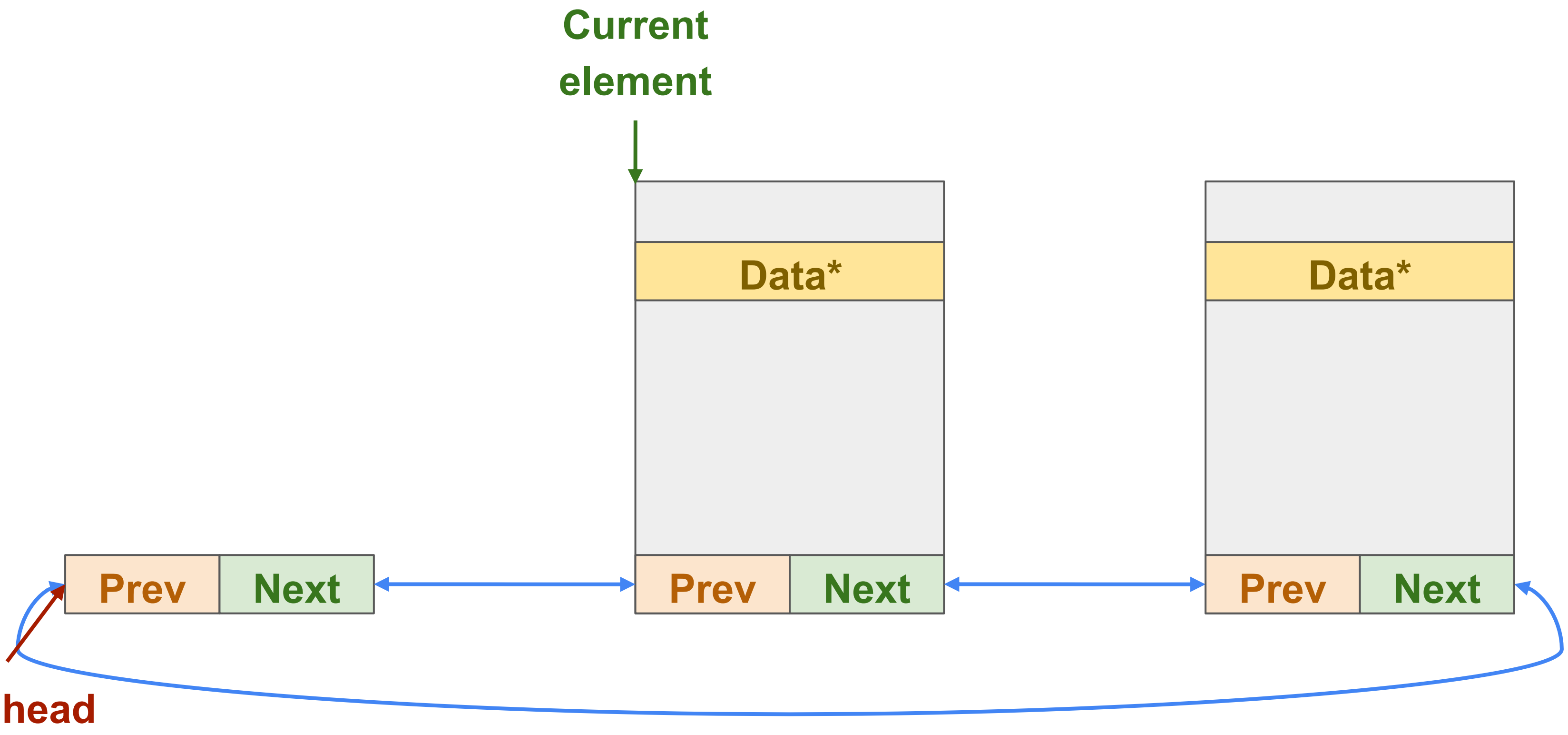
Case study: Linux's list iterator

Iteration 1



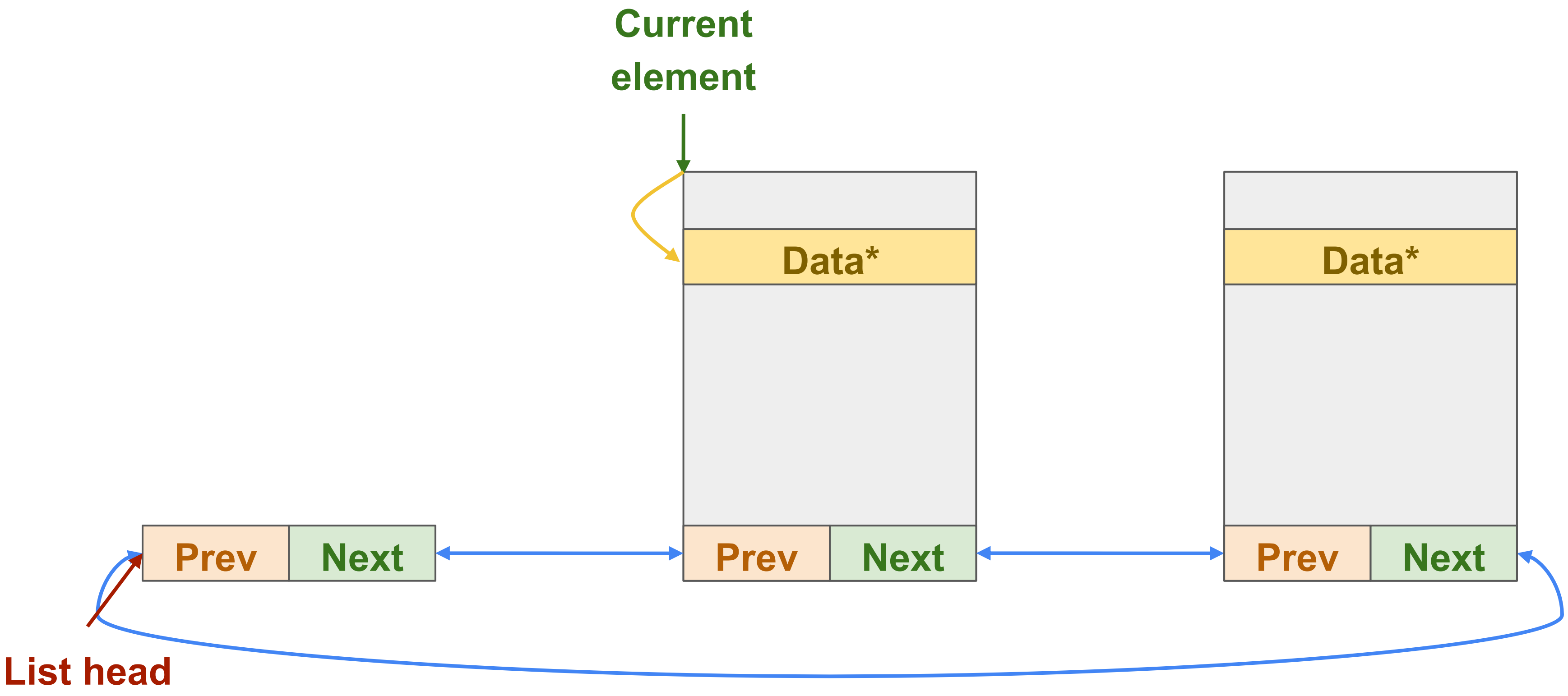
Case study: Linux's list iterator

Iteration 1



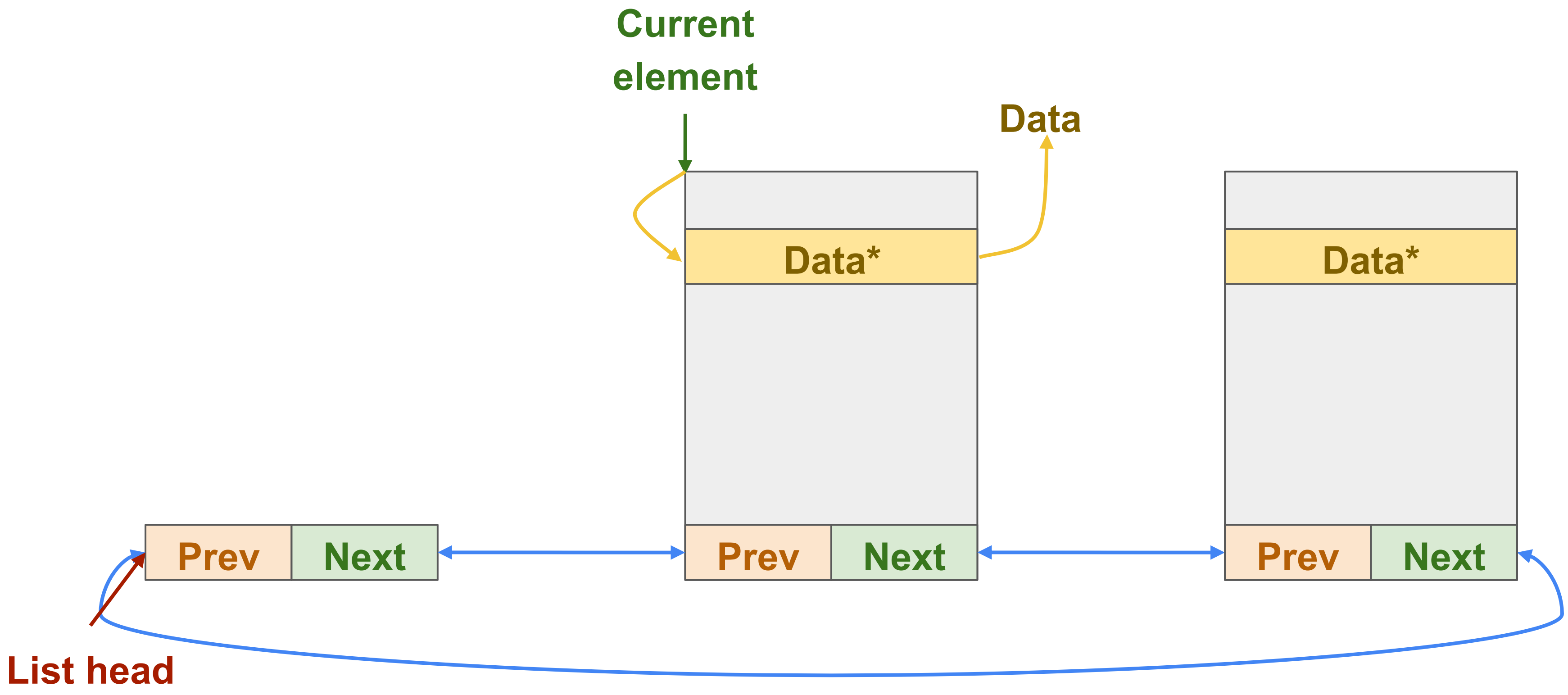
Case study: Linux's list iterator

Iteration 1



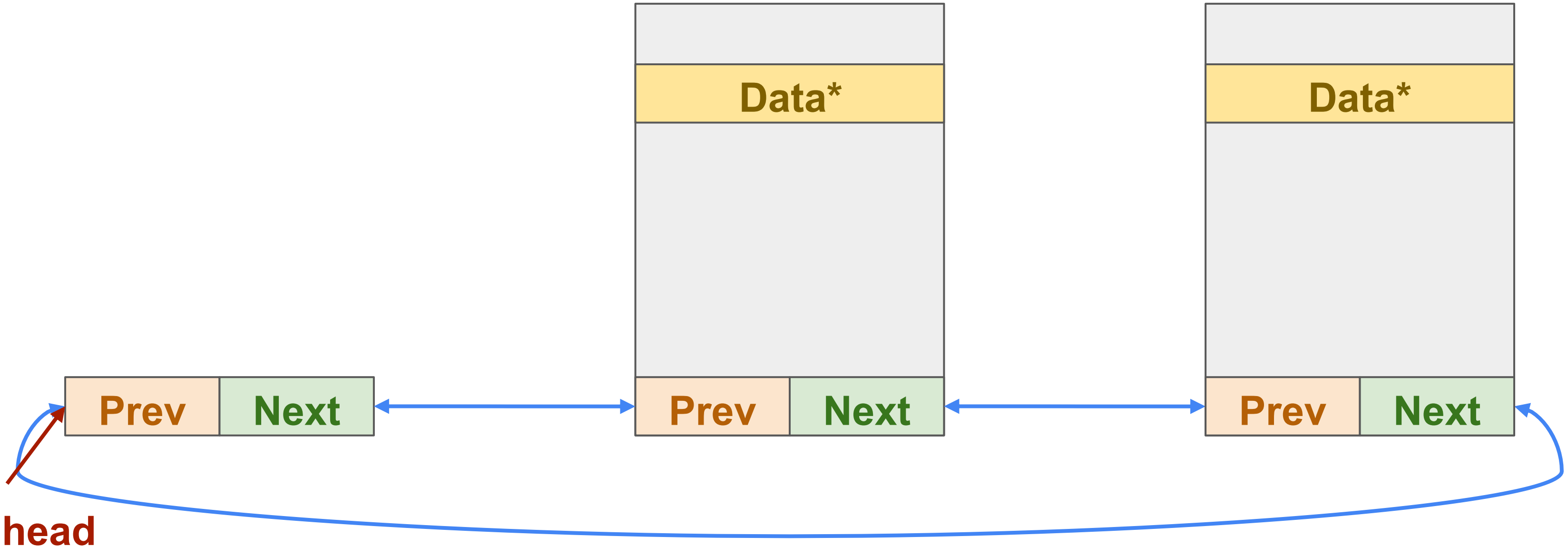
Case study: Linux's list iterator

Iteration 1



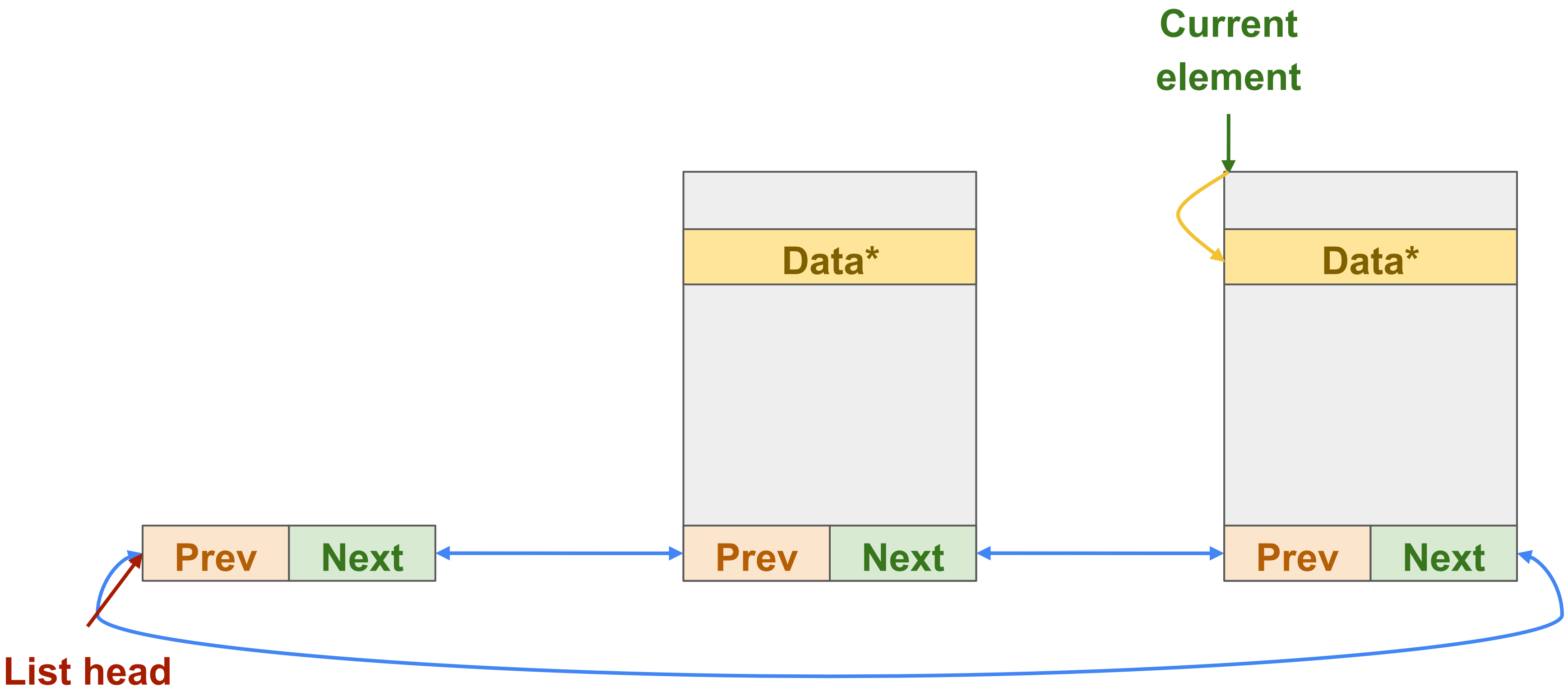
Case study: Linux’s list iterator

Iteration 2



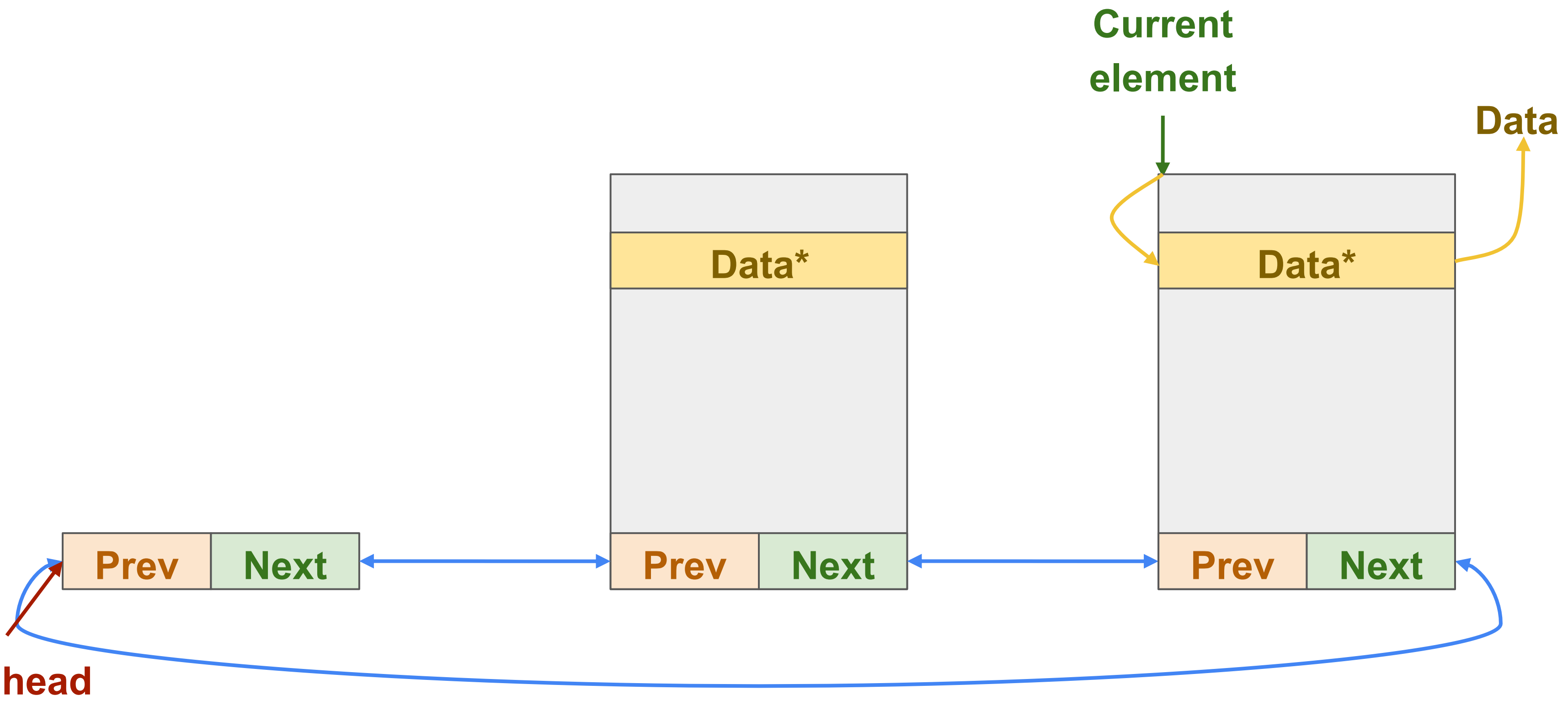
Case study: Linux's list iterator

Iteration 2



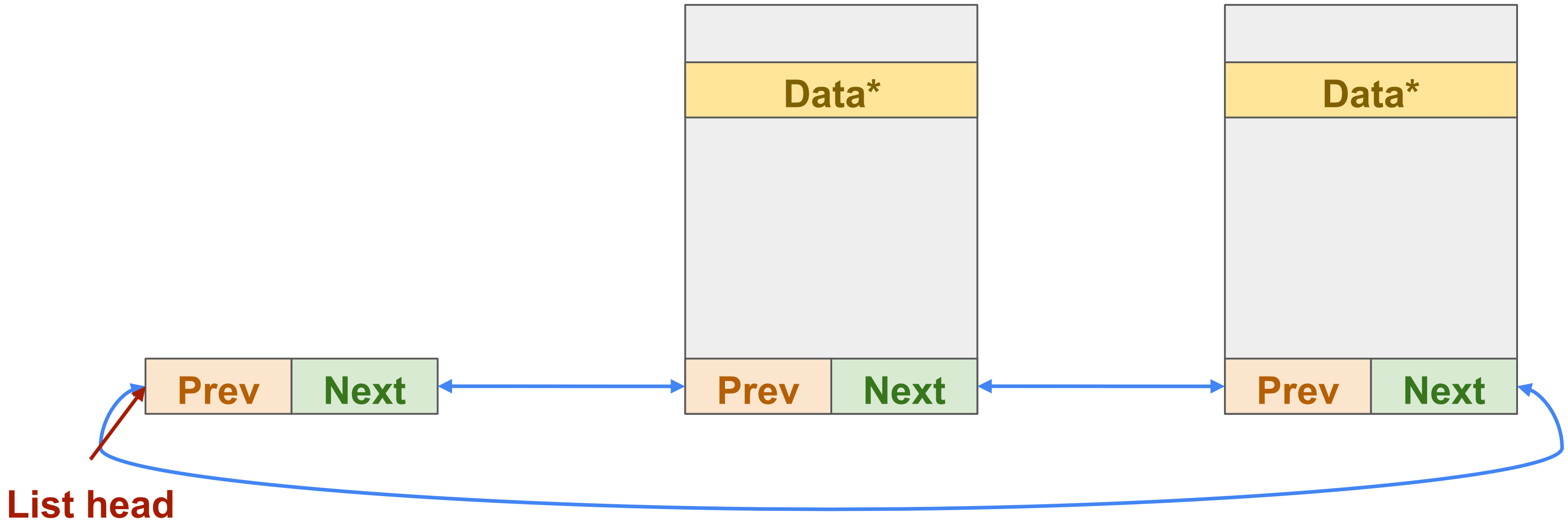
Case study: Linux’s list iterator

Iteration 2



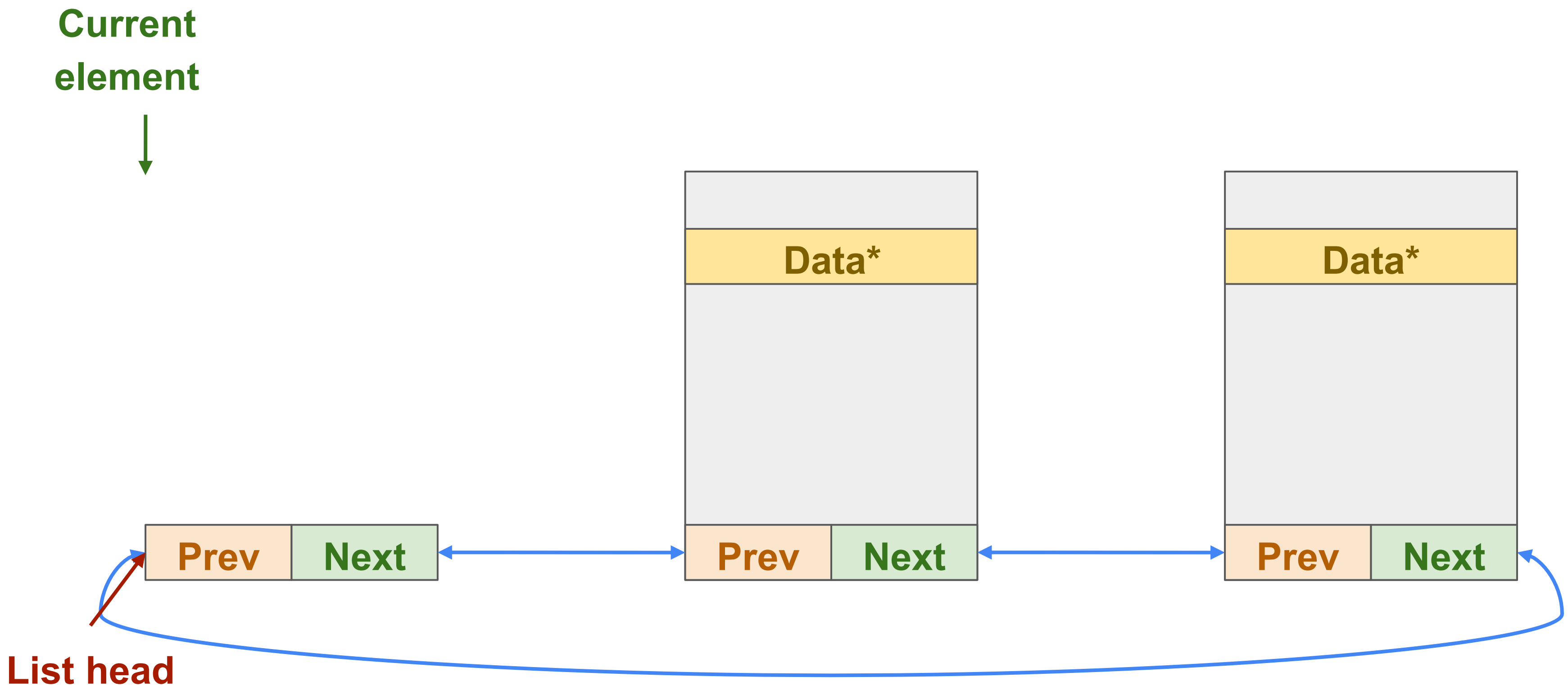
Case study: Linux's list iterator

Iteration 3 (termination)



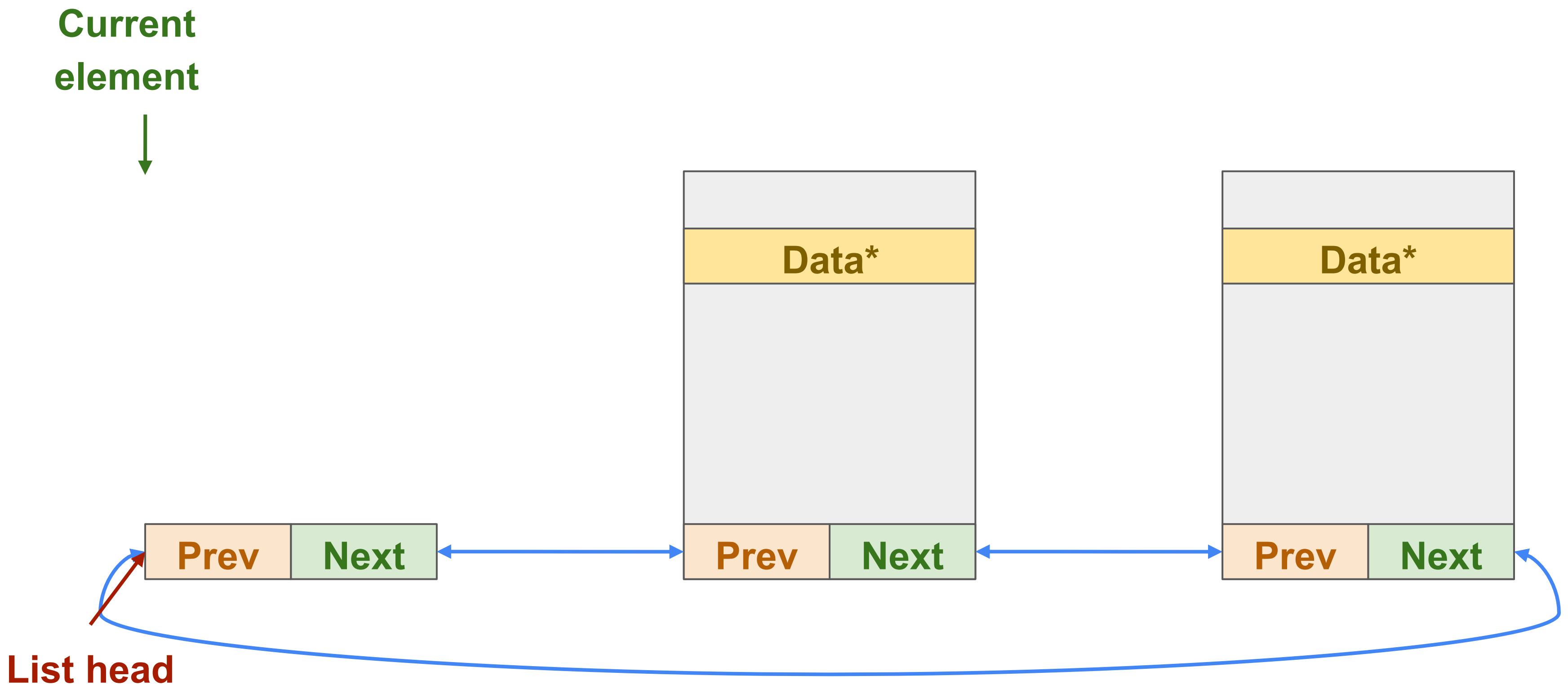
Case study: Linux's list iterator

Iteration 3 (termination)



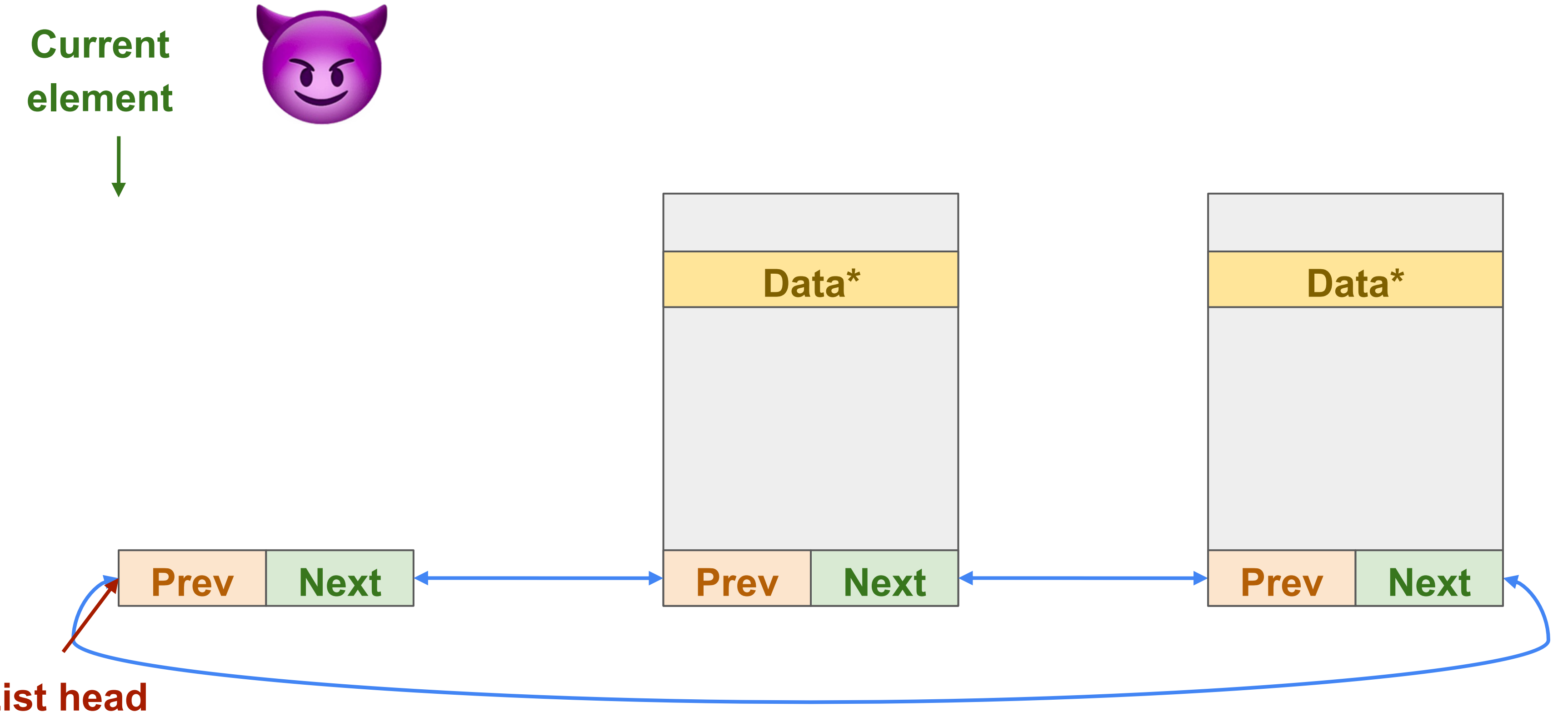
Case study: Linux's list iterator

Iteration 3 (misprediction)



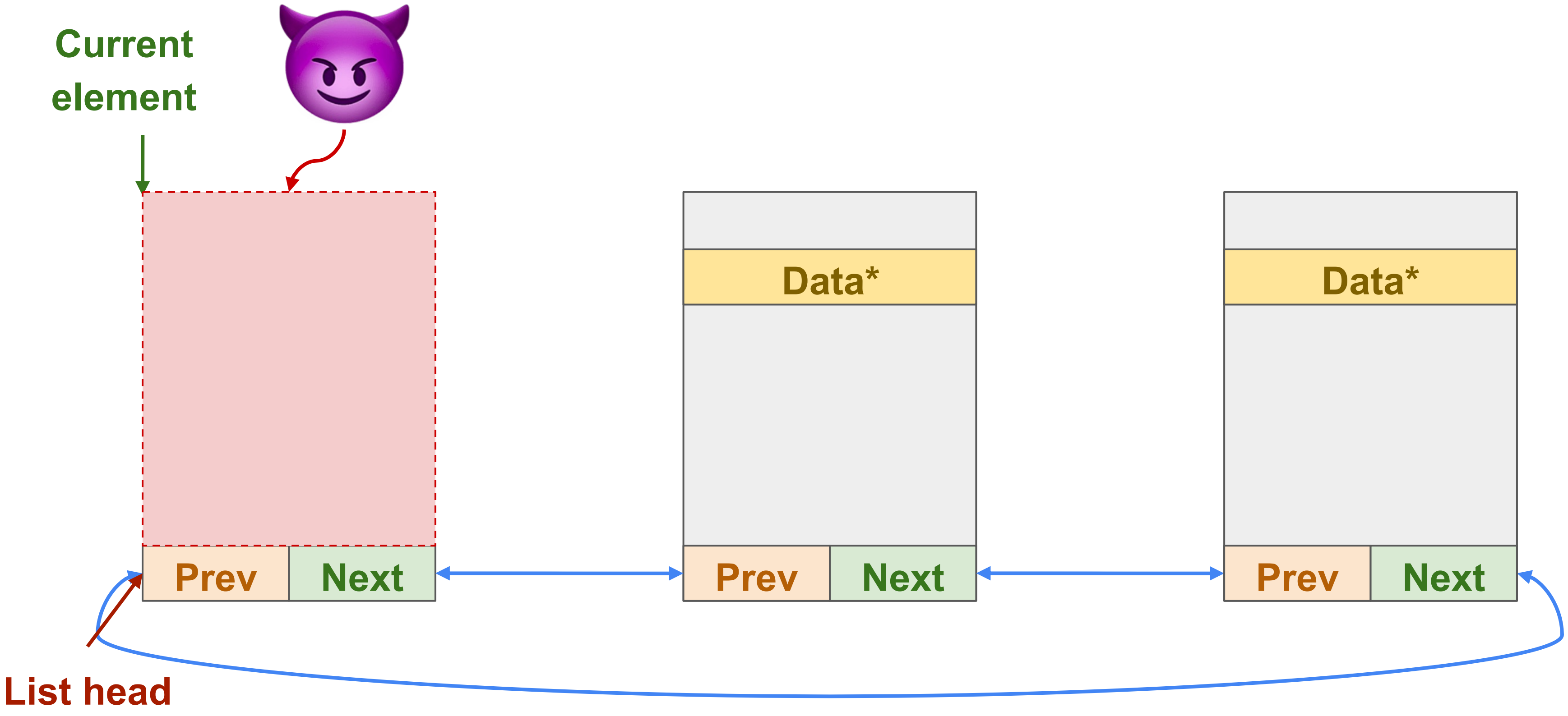
Case study: Linux’s list iterator

Iteration 3 (misprediction)



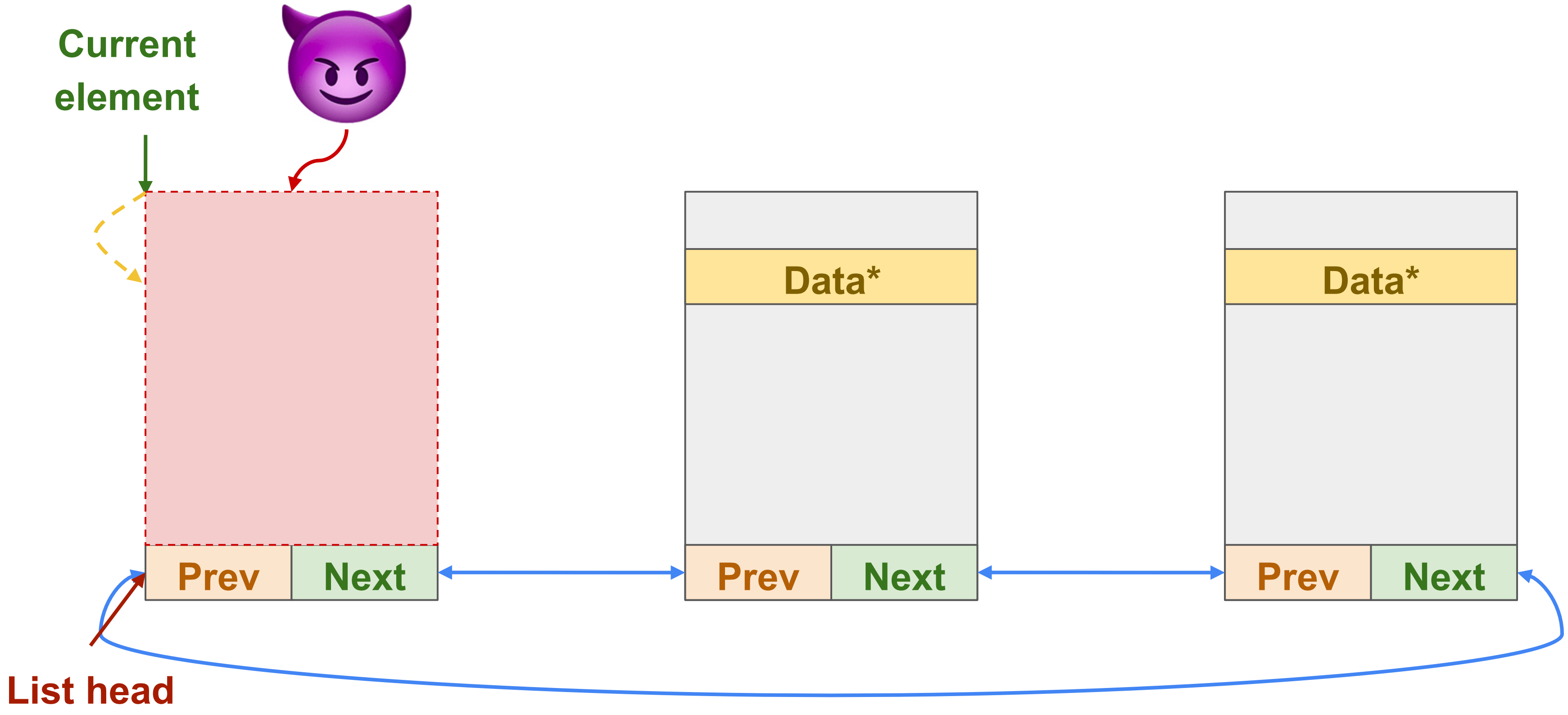
Case study: Linux’s list iterator

Iteration 3 (misprediction)



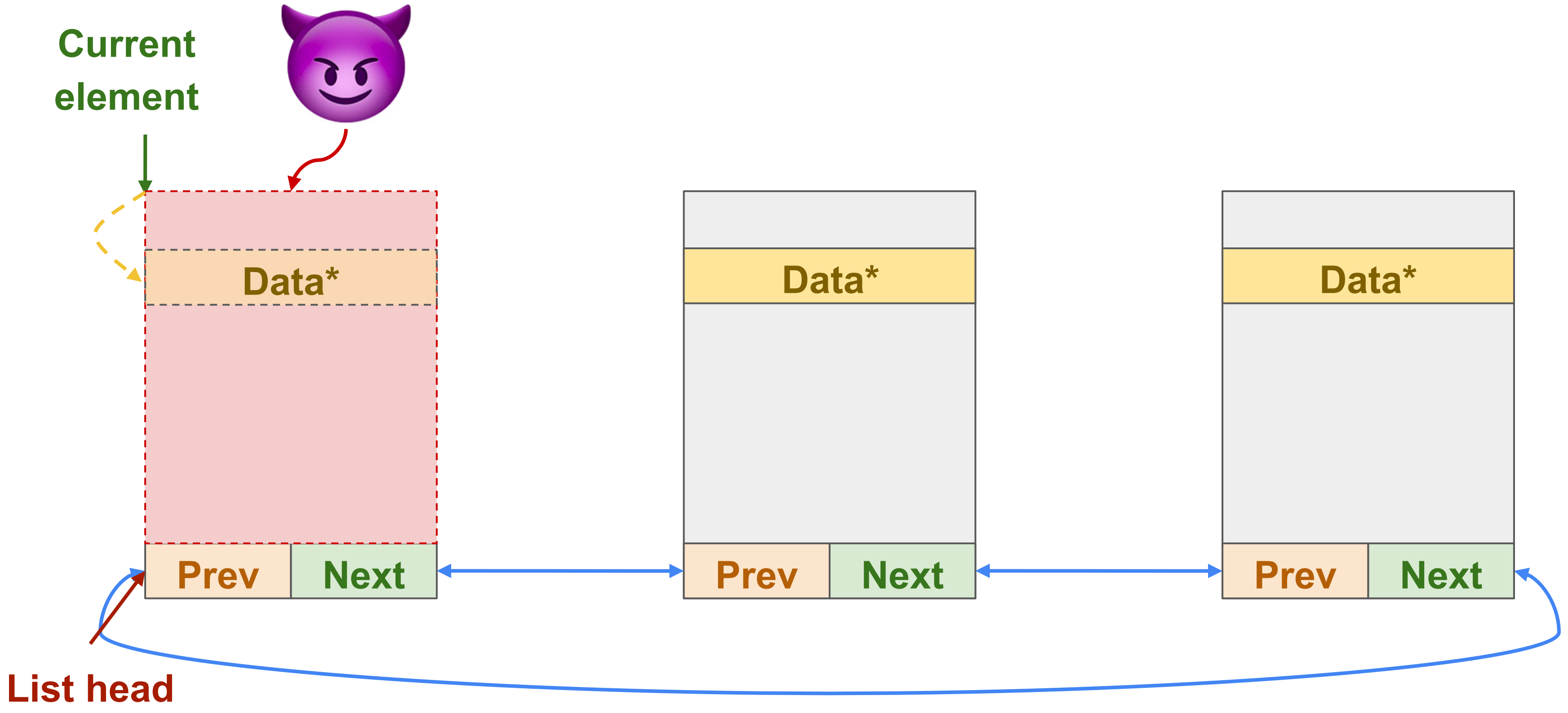
Case study: Linux’s list iterator

Iteration 3 (misprediction)



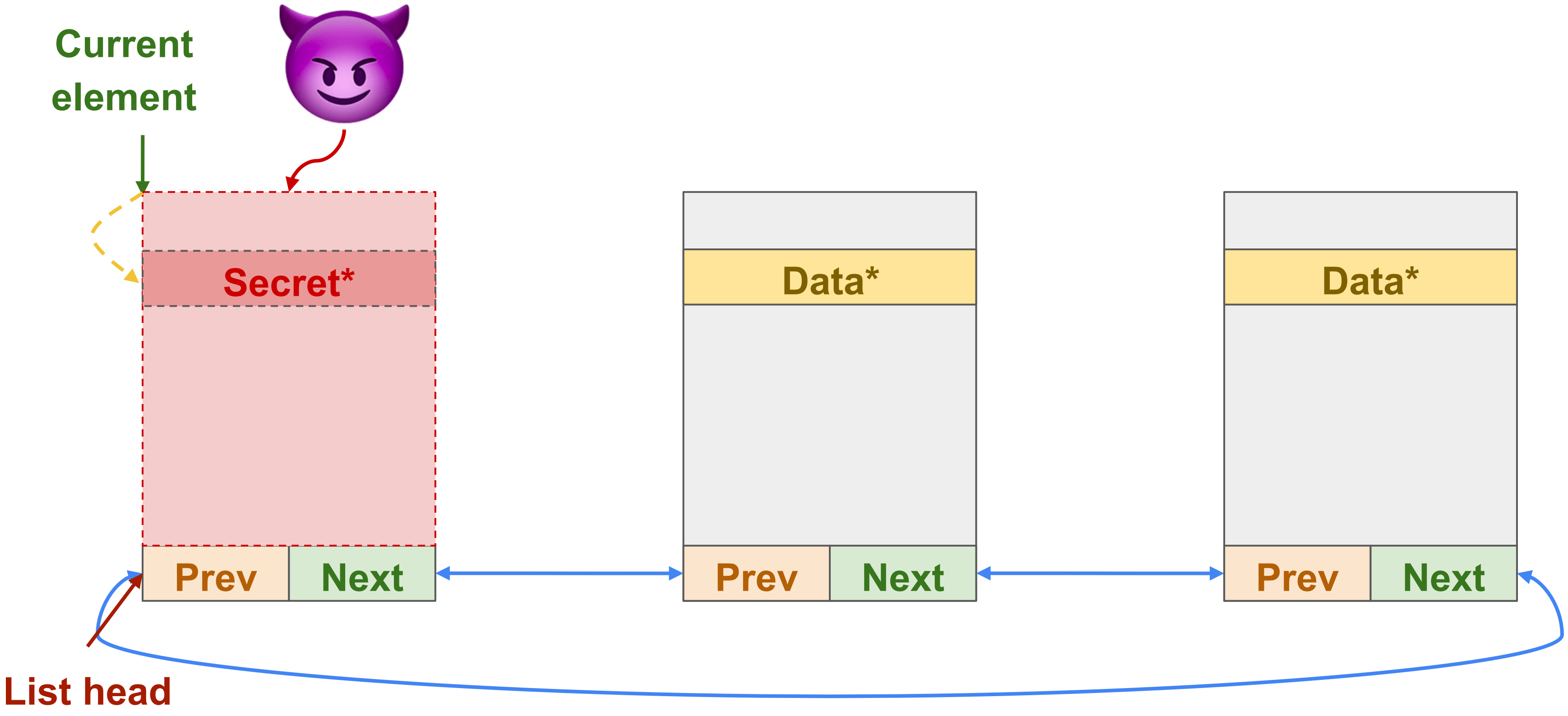
Case study: Linux's list iterator

Iteration 3 (misprediction)



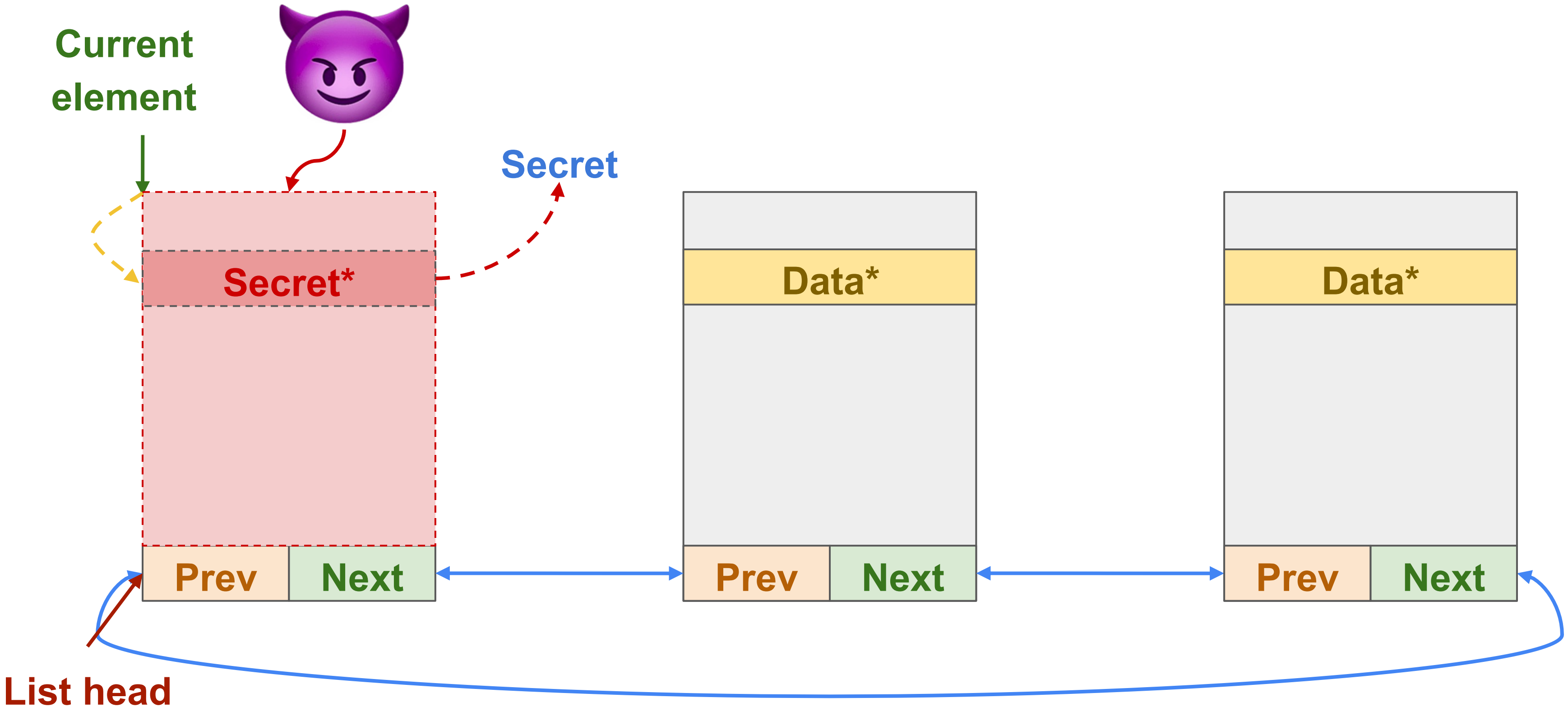
Case study: Linux's list iterator

Iteration 3 (misprediction)



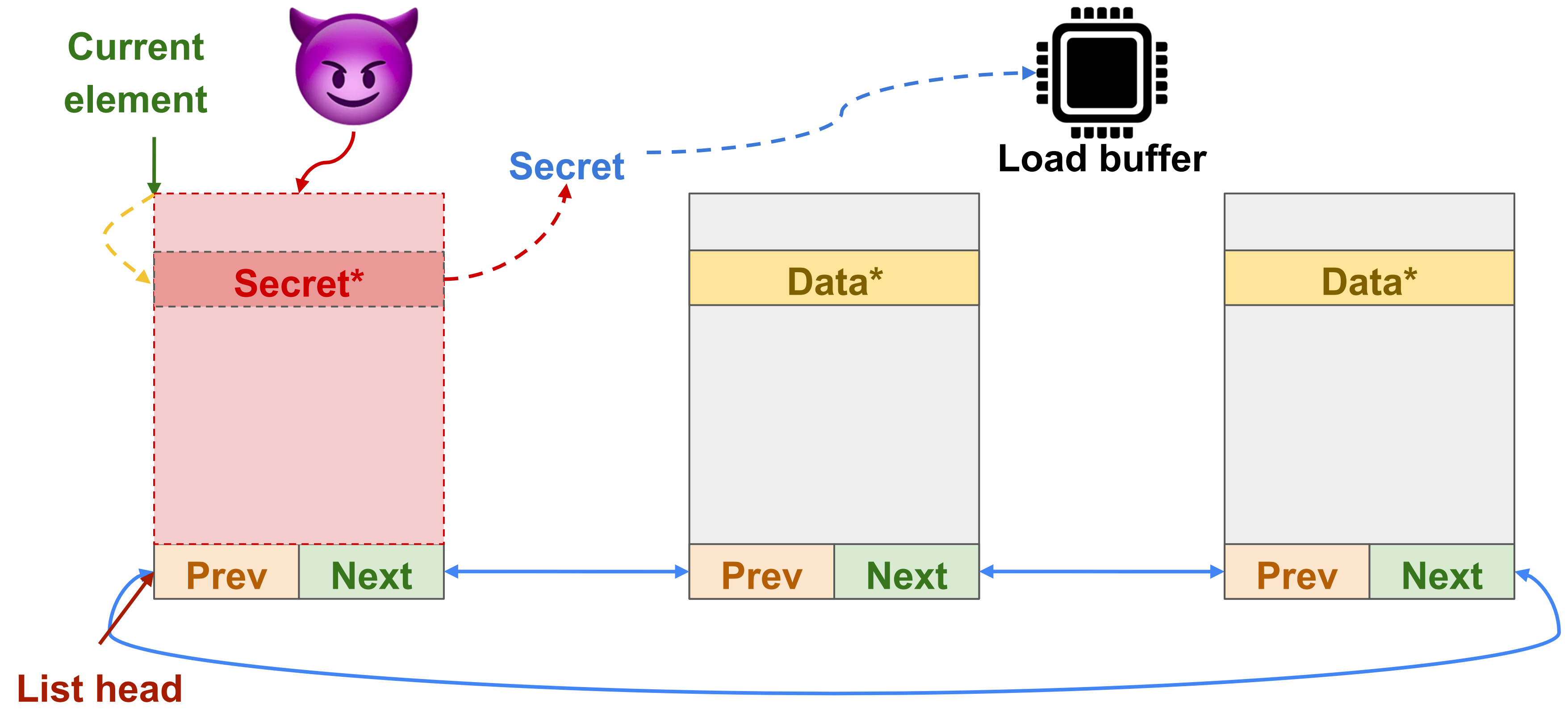
Case study: Linux's list iterator

Iteration 3 (misprediction)



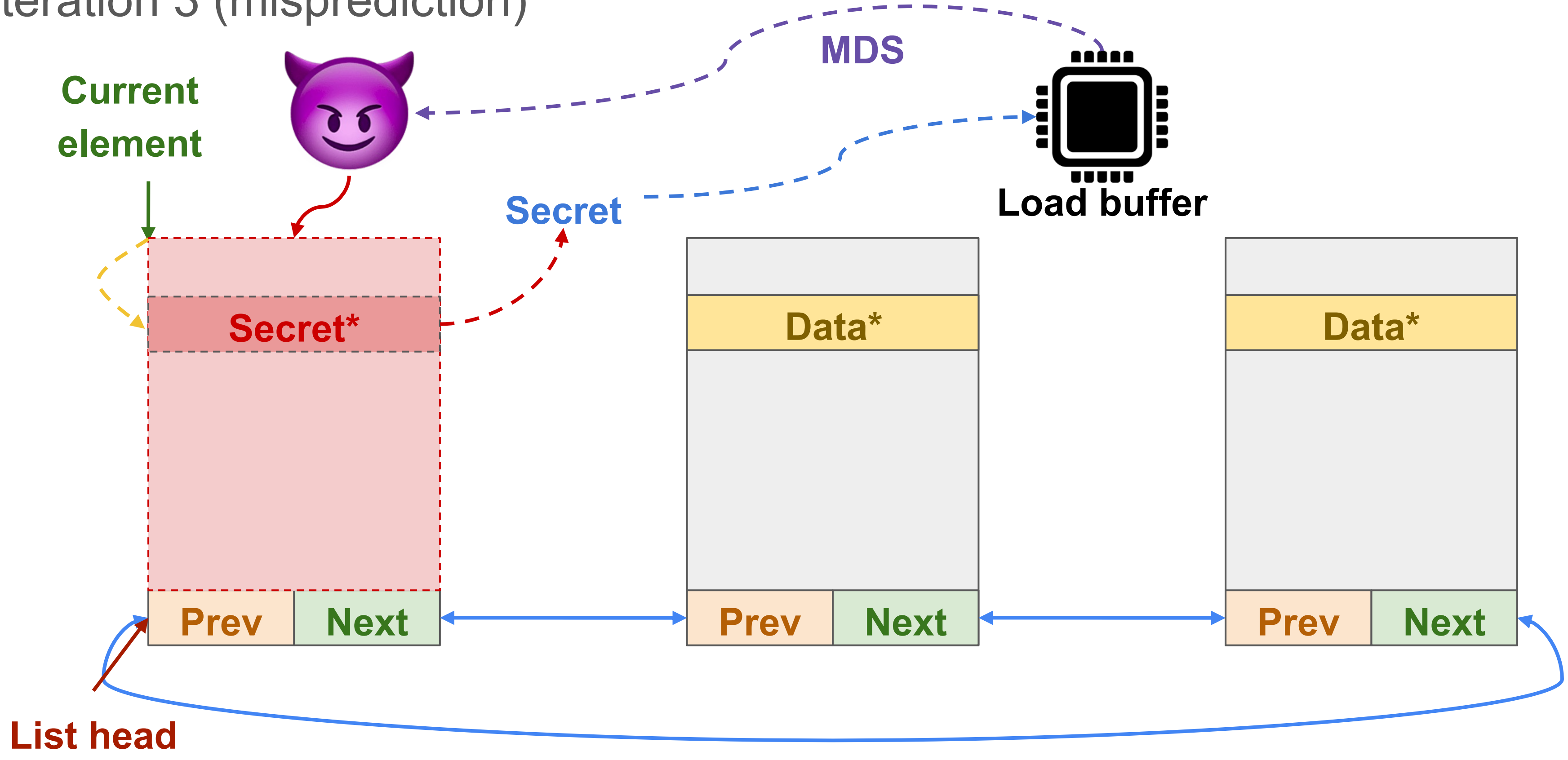
Case study: Linux's list iterator

Iteration 3 (misprediction)



Case study: Linux's list iterator

Iteration 3 (misprediction)



Finally, we implemented a **proof-of-concept exploit** of an instance of this gadget.

Finally, we implemented a **proof-of-concept exploit** of an instance of this gadget.

Finally, we implemented a **proof-of-concept exploit** of an instance of this gadget.

But that’s just the **beginning of the story**…

Finally, we implemented a **proof-of-concept exploit** of an instance of this gadget.

But that’s just the **beginning of the story**…

Mitigation of the list gadget

Mitigation of the list gadget

To **mitigate** the gadget, the kernel has to:

Mitigation of the list gadget

To **mitigate** the gadget, the kernel has to:

(1) Rewrite its **list implementation**

Mitigation of the list gadget

To **mitigate** the gadget, the kernel has to:

(1) Rewrite its **list implementation**

Mitigation of the list gadget

To **mitigate** the gadget, the kernel has to:

(1) Rewrite its **list implementation** , and (2) **Upgrade the version of C** that it uses

Mitigation of the list gadget

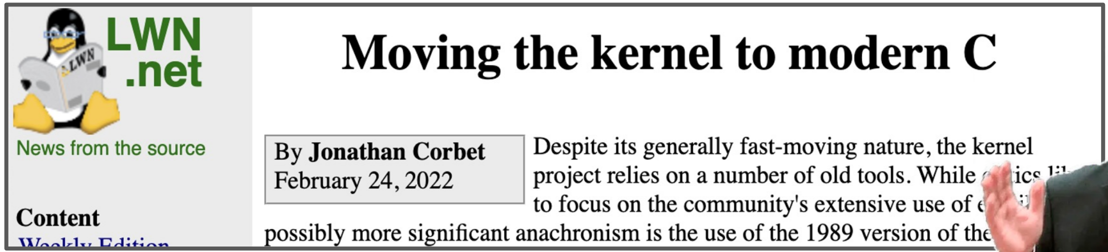
To **mitigate** the gadget, the kernel has to:

(1) Rewrite its **list implementation** , and (2) **Upgrade the version of C** that it uses

Mitigation of the list gadget

To **mitigate** the gadget, the kernel has to:

(1) Rewrite its **list implementation** , and (2) **Upgrade the version of C** that it uses



LWN .net
News from the source

Content

Moving the kernel to modern C

By Jonathan Corbet
February 24, 2022

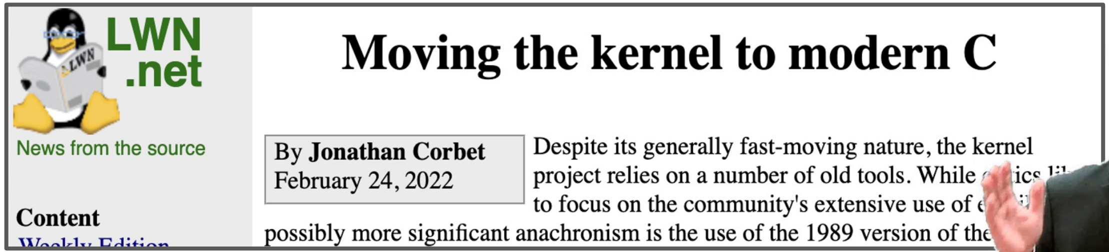
Despite its generally fast-moving nature, the kernel project relies on a number of old tools. While critics like to focus on the community's extensive use of e possibly more significant anachronism is the use of the 1989 version of the

(From **C89** to **C11**)

Mitigation of the list gadget

To **mitigate** the gadget, the kernel has to:

(1) Rewrite its **list implementation** , and (2) **Upgrade the version of C** that it uses



LWN .net

News from the source

Content

Weekly Edition

Moving the kernel to modern C

By **Jonathan Corbet**
February 24, 2022

Despite its generally fast-moving nature, the kernel project relies on a number of old tools. While c ics li to focus on the community's extensive use of e possibly more significant anachronism is the use of the 1989 version of the

(From **C89** to **C11**)

More than **500 treewide patches are required** to implement a reasonable defense.

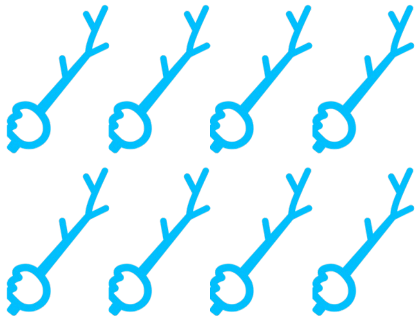
Conclusion

Conclusion

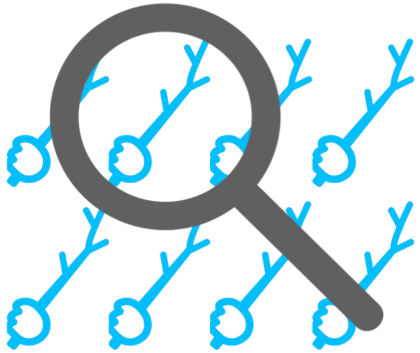
SPECTRE

Conclusion

Conclusion



Conclusion

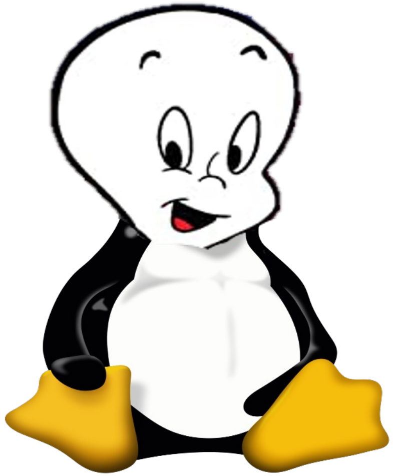


Conclusion

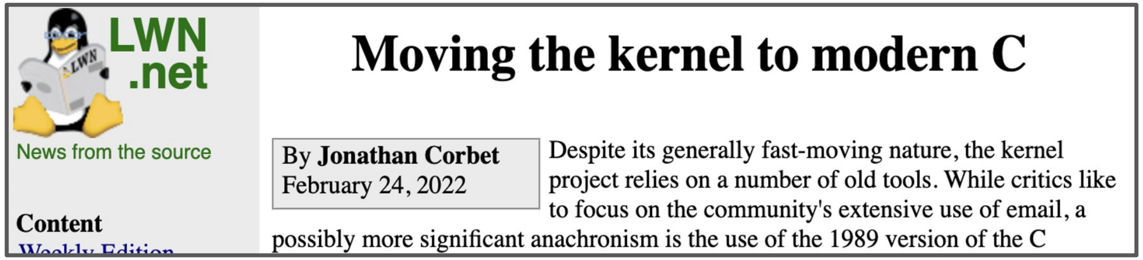
Conclusion

Conclusion

Conclusion



Conclusion



Moving the kernel to modern C

By **Jonathan Corbet**
February 24, 2022

Despite its generally fast-moving nature, the kernel project relies on a number of old tools. While critics like to focus on the community's extensive use of email, a possibly more significant anachronism is the use of the 1989 version of the C

Thank you!

Backup slides

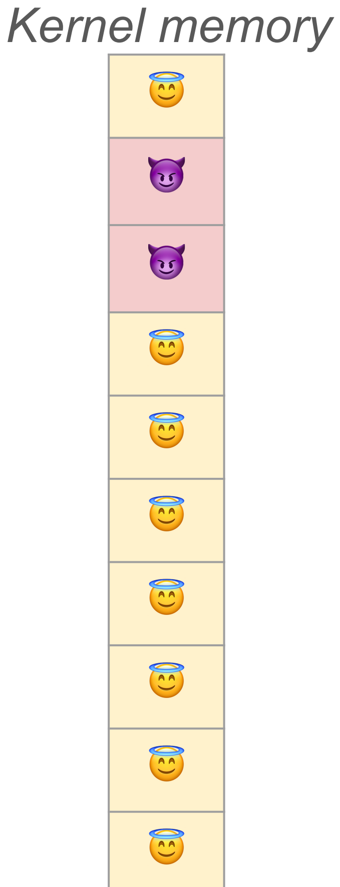
Background: Attacker injection via memory massaging

Background: Attacker injection via memory massaging

```
if (a < size) {
   b = arr1[a];
   c = arr2[b];
   d = arr3[c];
}
```

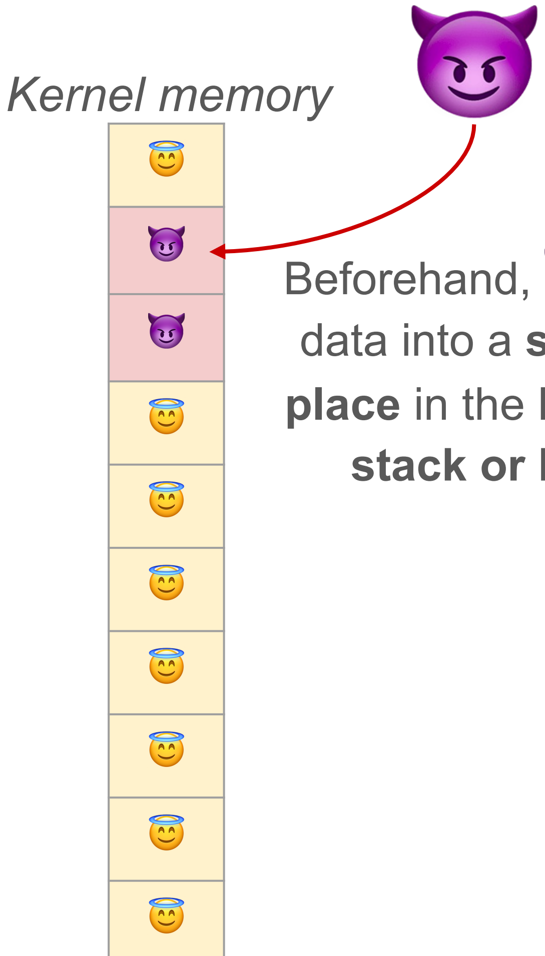
Background: Attacker injection via memory massaging

```
if (a < size) {
   b = arr1[a];
   c = arr2[b];
   d = arr3[c];
}
```



Background: Attacker injection via memory massaging

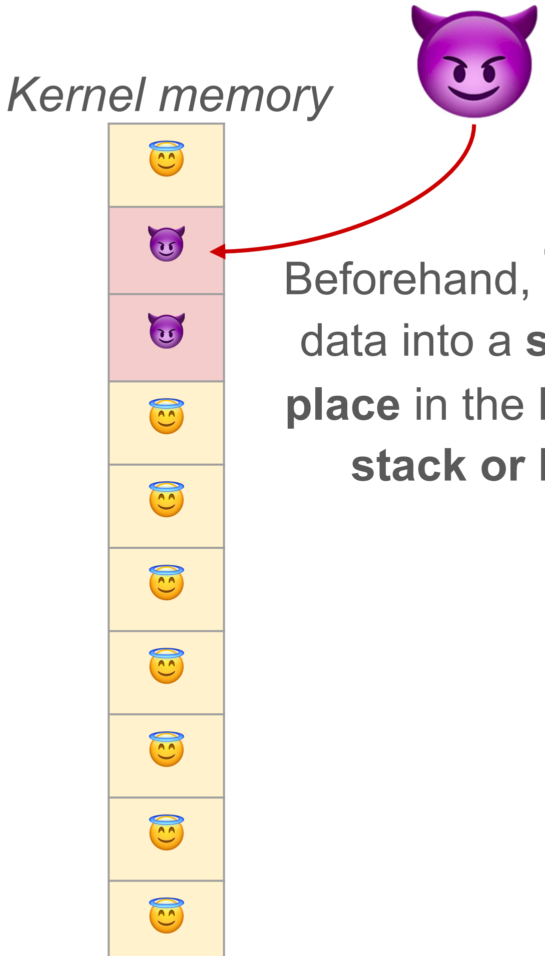
```
if (a < size) {
   b = arr1[a];
   c = arr2[b];
   d = arr3[c];
}
```



Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

Background: Attacker injection via memory massaging

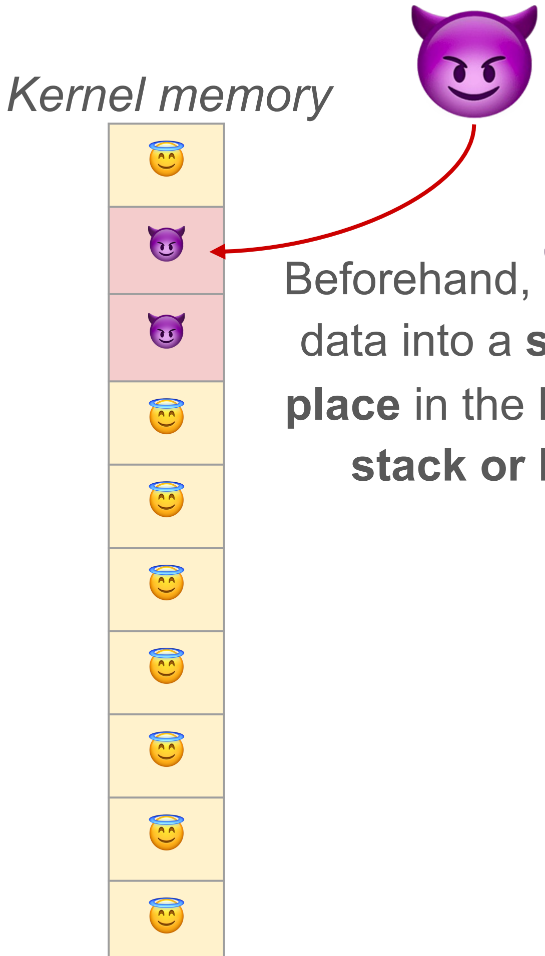
```
if (a < size) {
   b = arr1[a];
   c = arr2[b];
   d = arr3[c];
}
```



Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

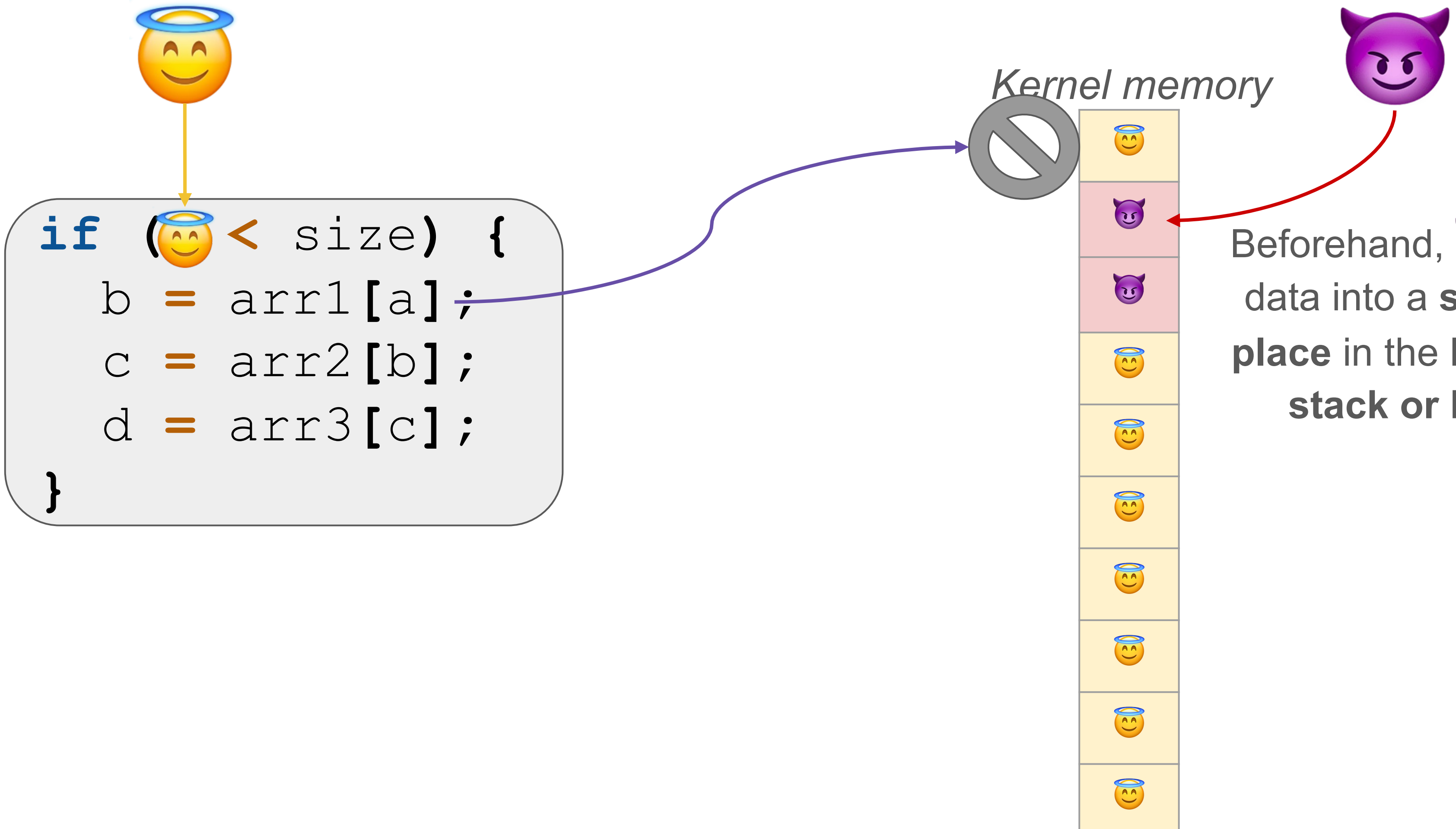
Background: Attacker injection via memory massaging

```
if (😇 < size) {
   b = arr1[a];
   c = arr2[b];
   d = arr3[c];
}
```



Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

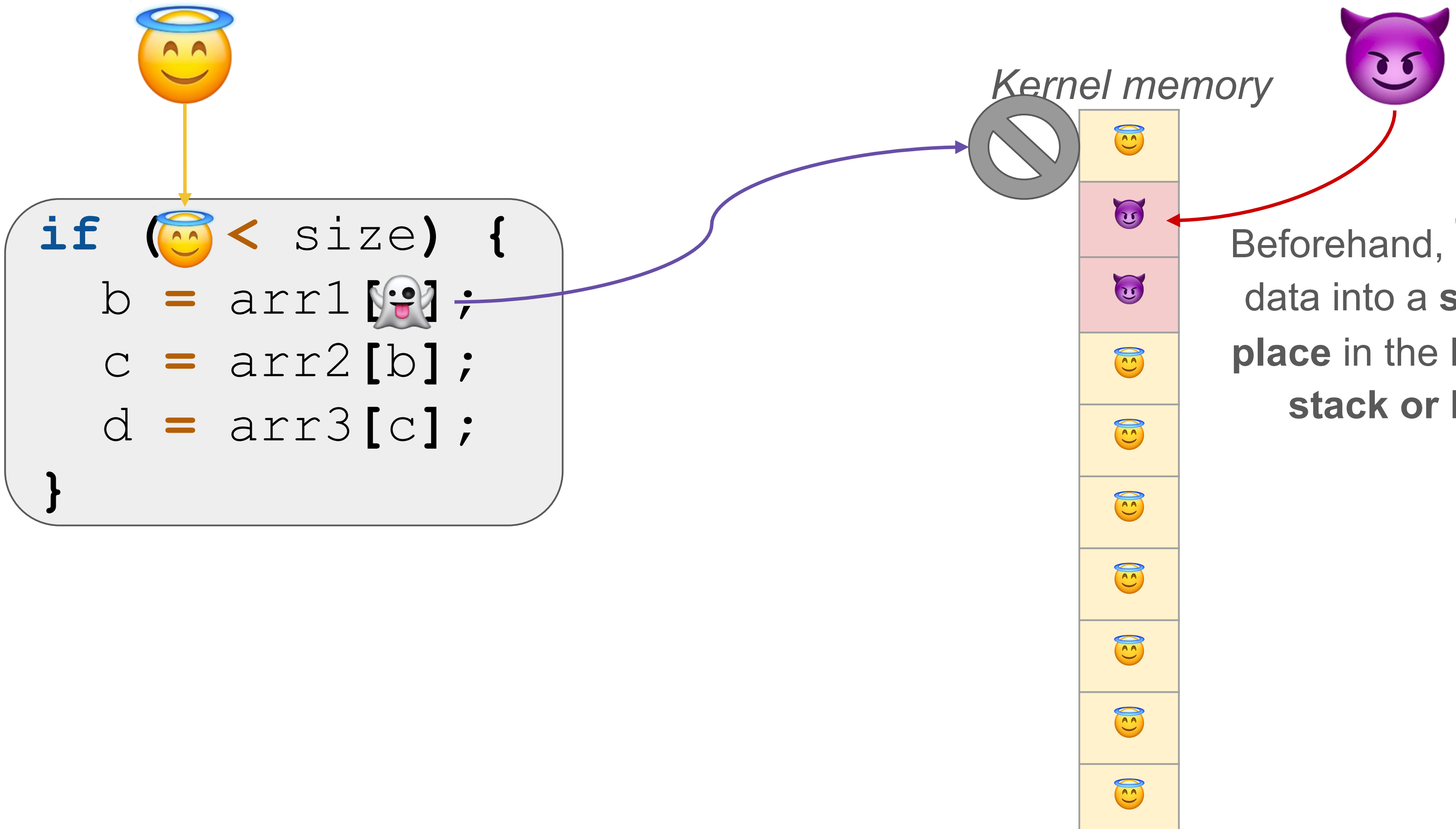
Background: Attacker injection via memory massaging



```
if (😇 < size) {
   b = arr1[a];
   c = arr2[b];
   d = arr3[c];
}
```

Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

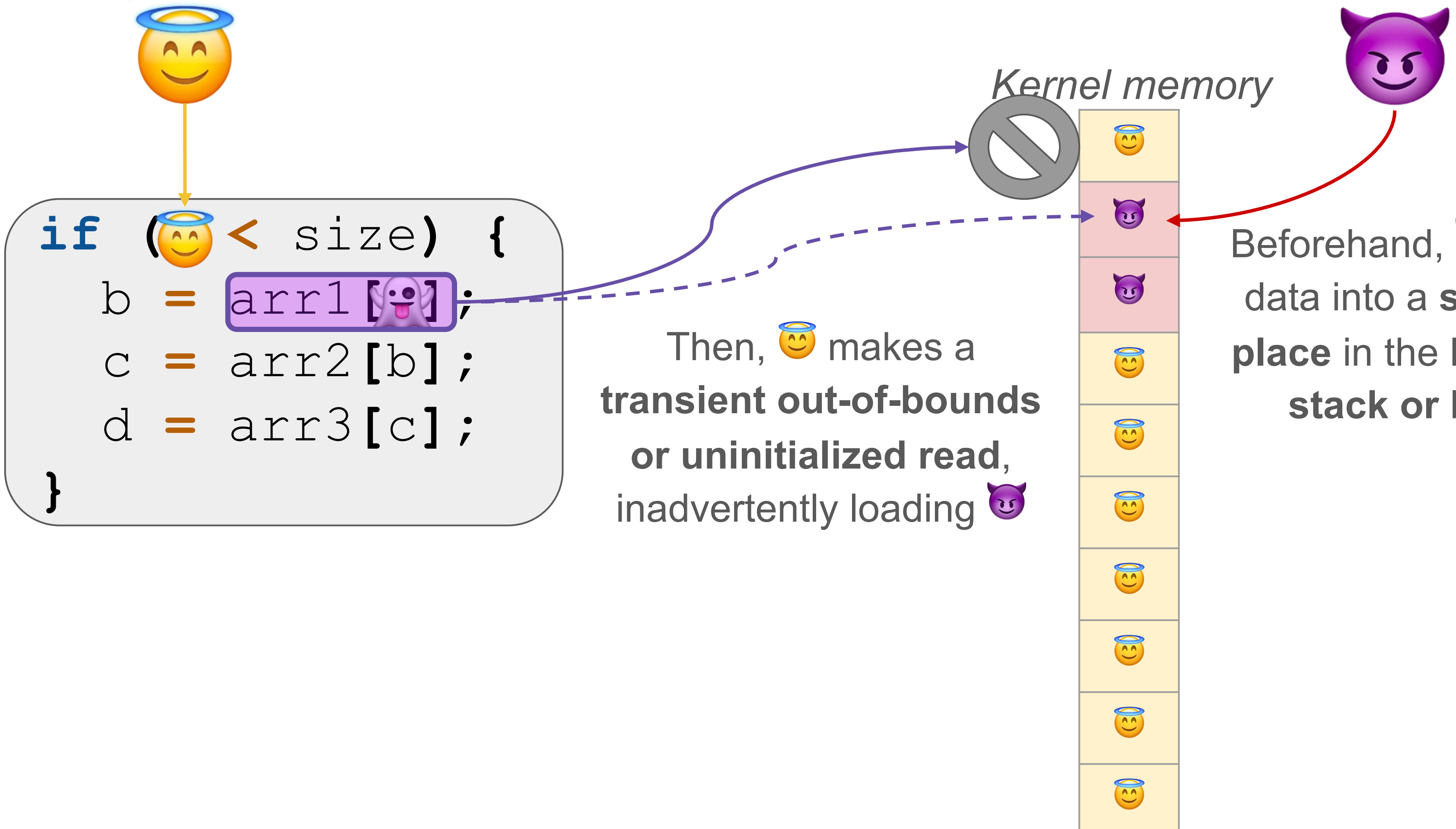
Background: Attacker injection via memory massaging



```
if (😇 < size) {
   b = arr1[👻];
   c = arr2[b];
   d = arr3[c];
}
```

Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

Background: Attacker injection via memory massaging

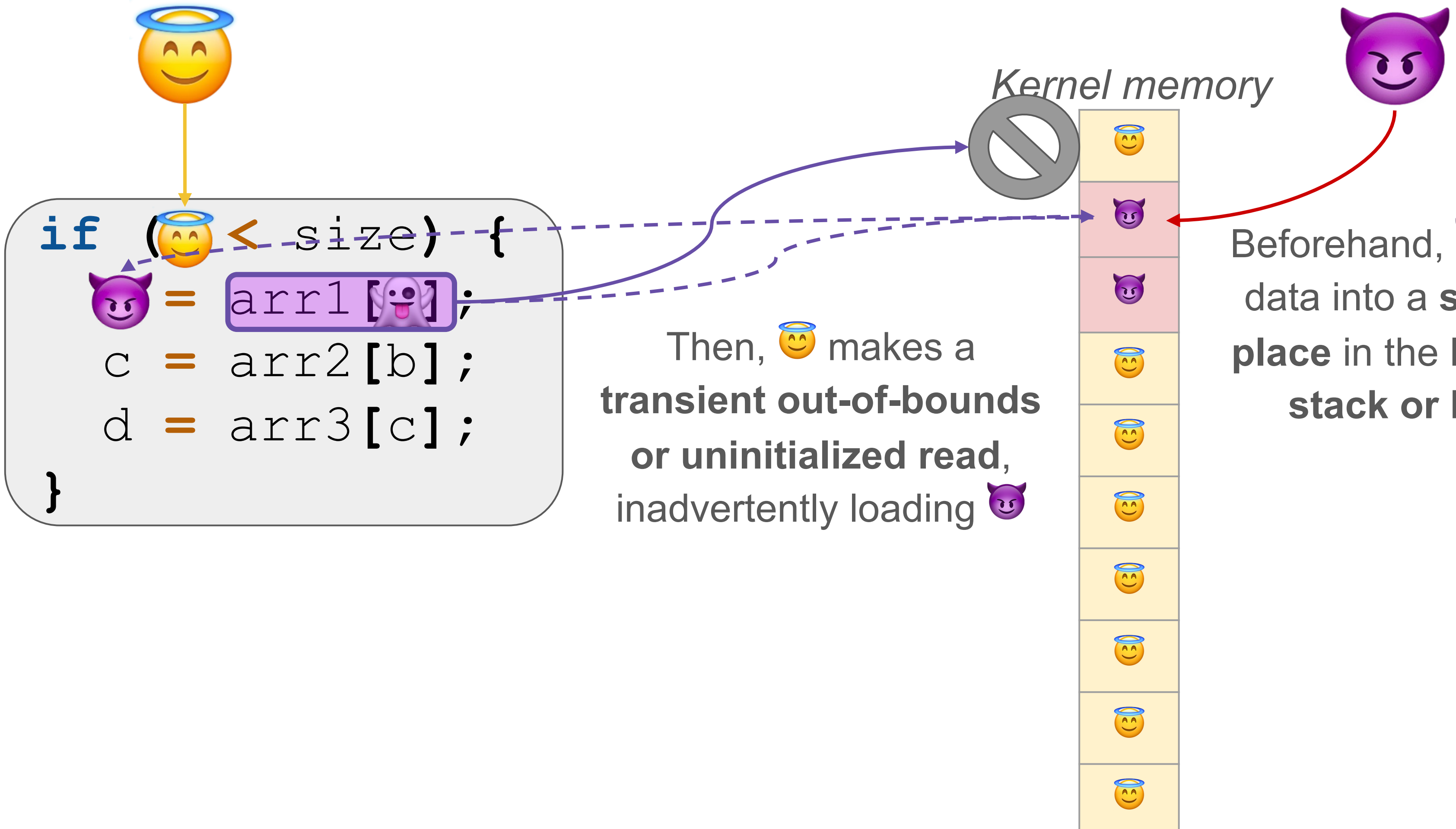


```
if (😇 < size) {
   b = arr1[👻];
   c = arr2[b];
   d = arr3[c];
}
```

Then, 😇 makes a **transient out-of-bounds or uninitialized read**, inadvertently loading 😈

Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

Background: Attacker injection via memory massaging

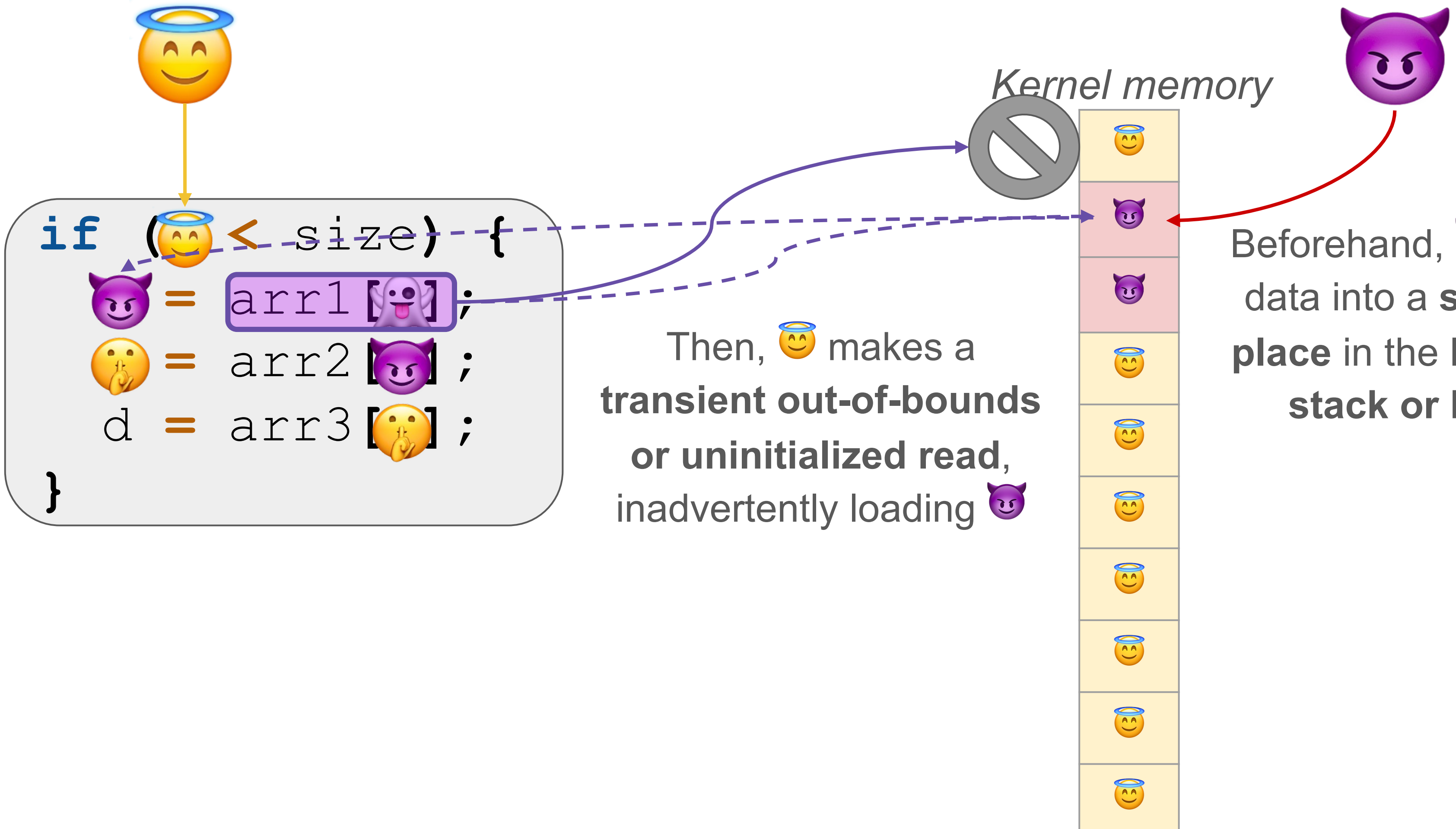


```
if (😇 < size) {
  😈 = arr1[👻];
  c = arr2[b];
  d = arr3[c];
}
```

Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

Then, 😇 makes a **transient out-of-bounds or uninitialized read**, inadvertently loading 😈

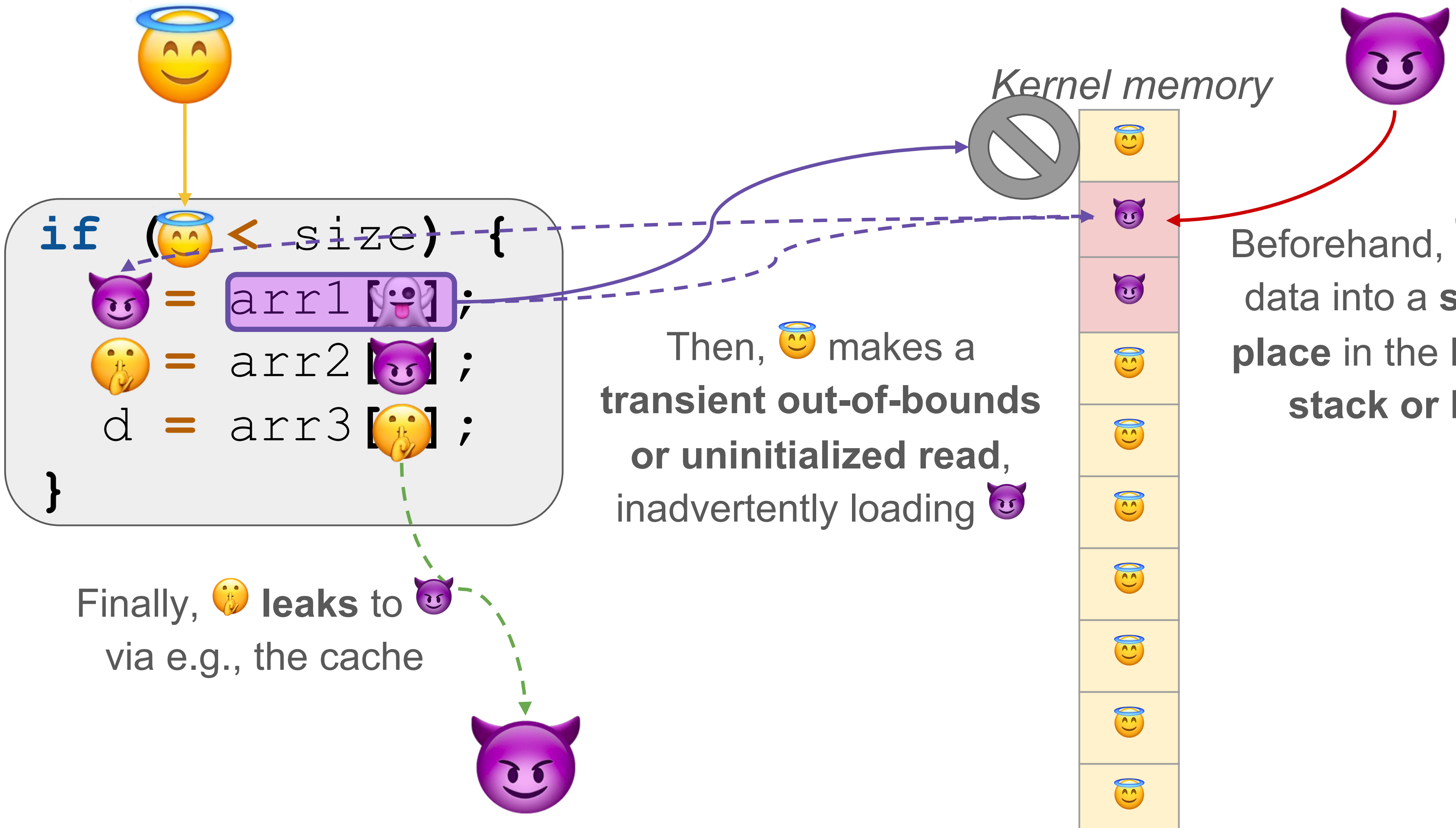
Background: Attacker injection via memory massaging



Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

Then, 😇 makes a **transient out-of-bounds or uninitialized read**, inadvertently loading 😈

Background: Attacker injection via memory massaging



```
if (😇 < size) {
  😈 = arr1[👻];
  🤫 = arr2[😈];
  d = arr3[🤫];
}
```

Beforehand, 😈 lands data into a **specific place** in the **kernel's stack or heap**

Then, 😇 makes a **transient out-of-bounds or uninitialized read**, inadvertently loading 😈

Finally, 🤫 **leaks** to 😈 via e.g., the cache

Background: Attacker injection via load value injection (LVI)

Background: Attacker injection via load value injection (LVI)

```
if (addr_is_mapped(ptr)) {
  x = *ptr;
  y = arr2[x];
  z = arr3[y];
}
```

Background: Attacker injection via load value injection (LVI)

```
if (addr_is_mapped(ptr)) {
   x = *ptr;
   y = arr2[x];
   z = arr3[y];
}
```

😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

Background: Attacker injection via load value injection (LVI)

```
if (addr_is_mapped(ptr)) {
   x = *ptr;
   y = arr2[x];
   z = arr3[y];
}
```

co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

Background: Attacker injection via load value injection (LVI)

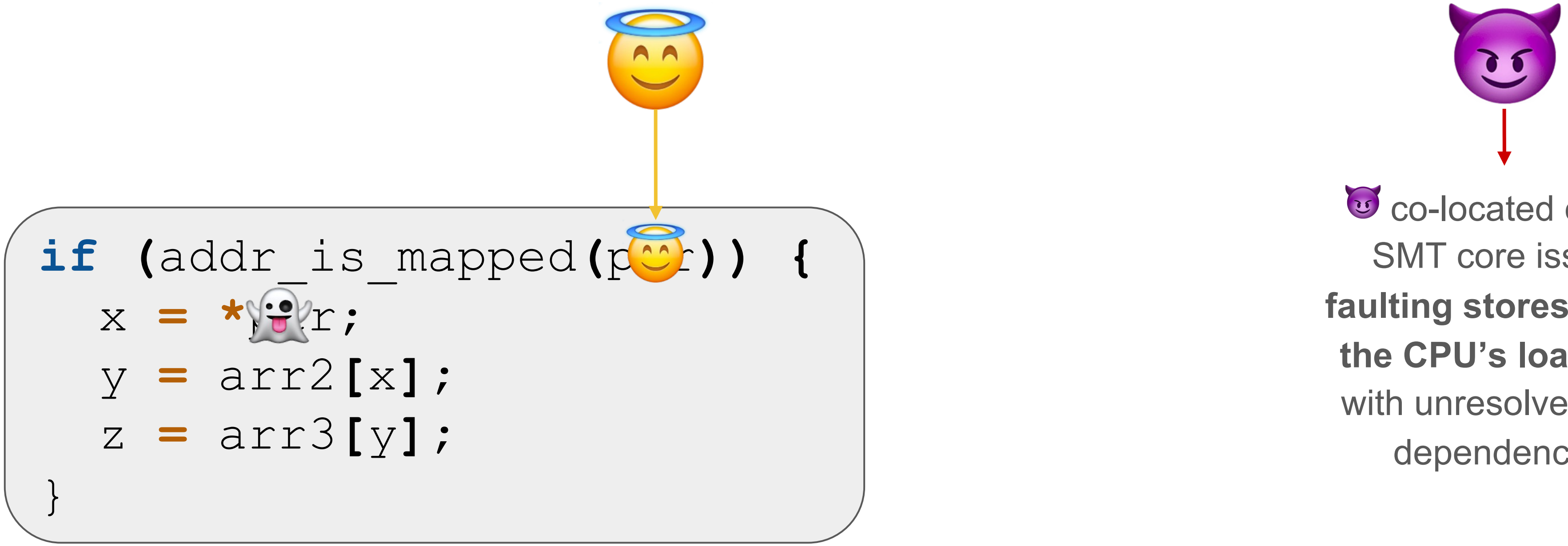
😇

😈

```
if (addr_is_mapped(p😇r)) {
   x = *ptr;
   y = arr2[x];
   z = arr3[y];
}
```

😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

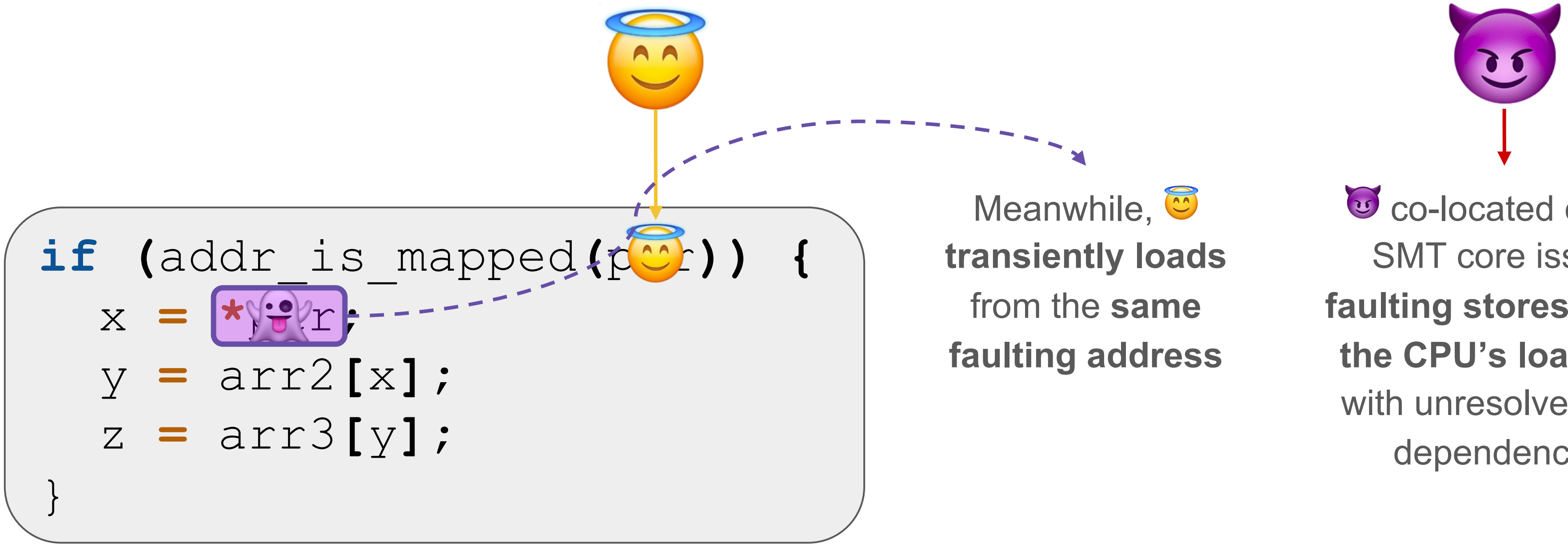
Background: Attacker injection via load value injection (LVI)



```
if (addr_is_mapped(p😇r)) {
   x = *👻r;
   y = arr2[x];
   z = arr3[y];
}
```

😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

Background: Attacker injection via load value injection (LVI)

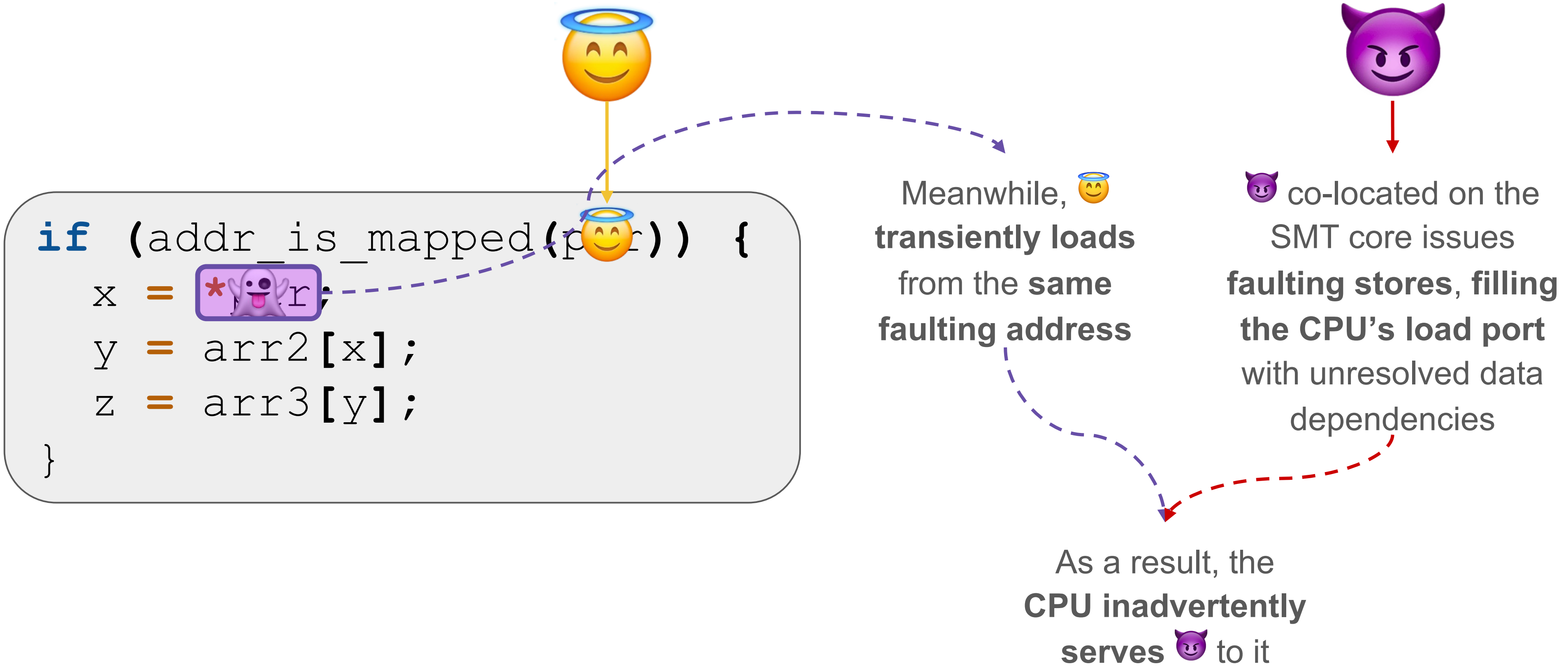


```
if (addr_is_mapped(ptr)) {
   x = *ptr;
   y = arr2[x];
   z = arr3[y];
}
```

Meanwhile, 😇 **transiently loads** from the **same faulting address**

😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

Background: Attacker injection via load value injection (LVI)



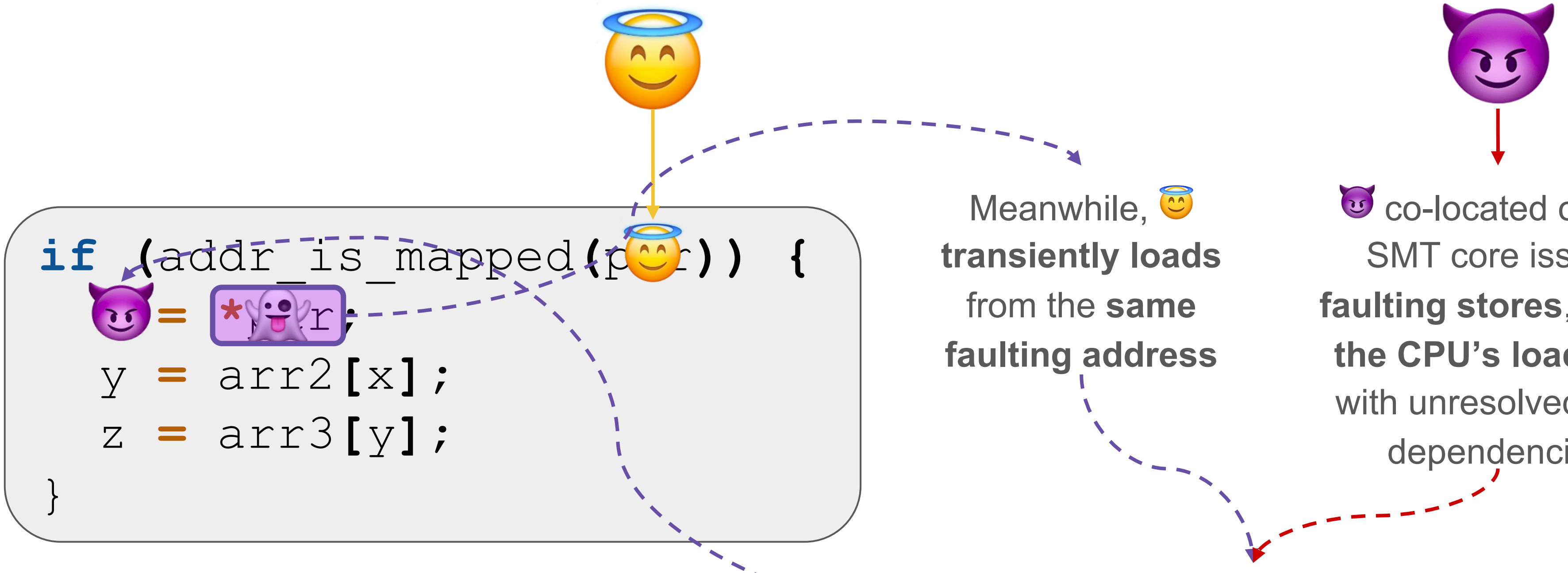
```
if (addr_is_mapped(ptr)) {
   x = *ptr;
   y = arr2[x];
   z = arr3[y];
}
```

Meanwhile, 😇 **transiently loads** from the **same faulting address**

😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

As a result, the **CPU inadvertently serves** 😈 to it

Background: Attacker injection via load value injection (LVI)



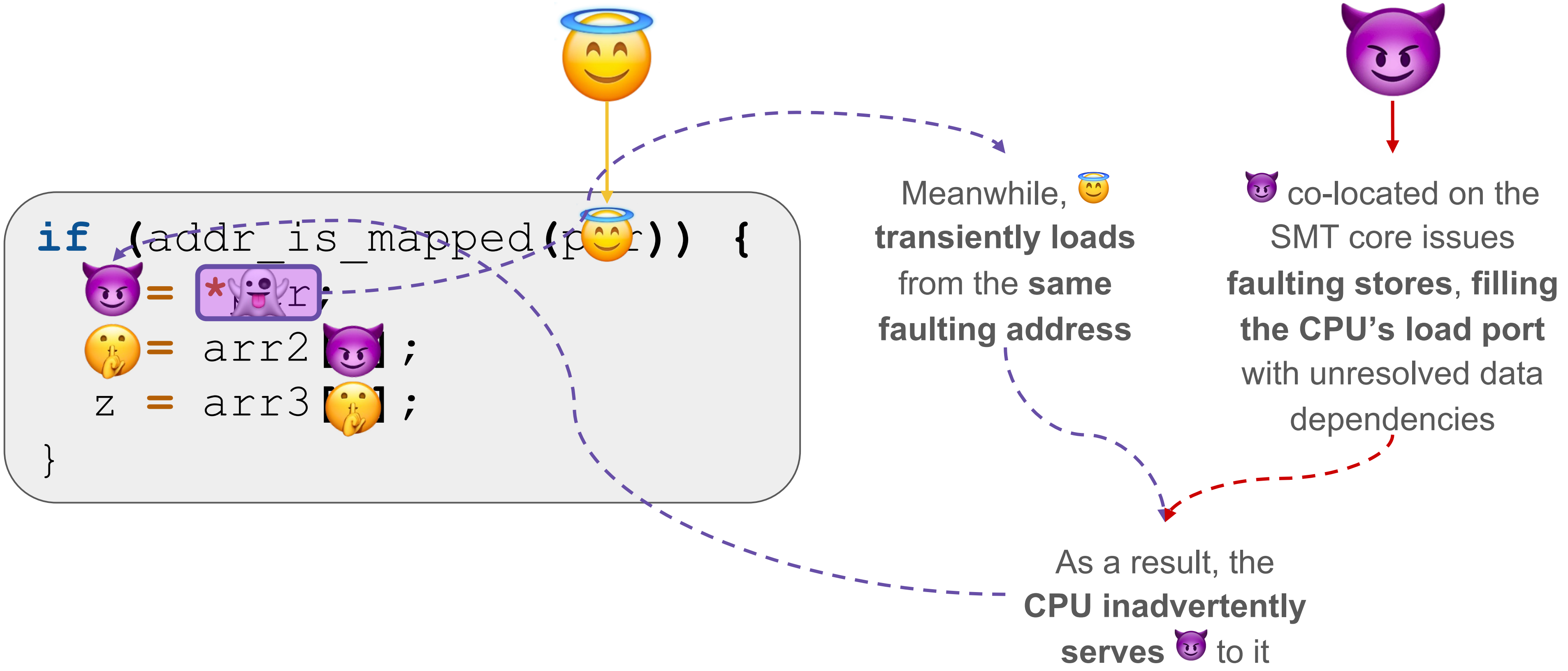
```
if (addr_is_mapped(ptr)) {
  😈= *👻r;
  y = arr2[x];
  z = arr3[y];
}
```

Meanwhile, 😇 **transiently loads** from the **same faulting address**

😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

As a result, the **CPU inadvertently serves** 😈 to it

Background: Attacker injection via load value injection (LVI)



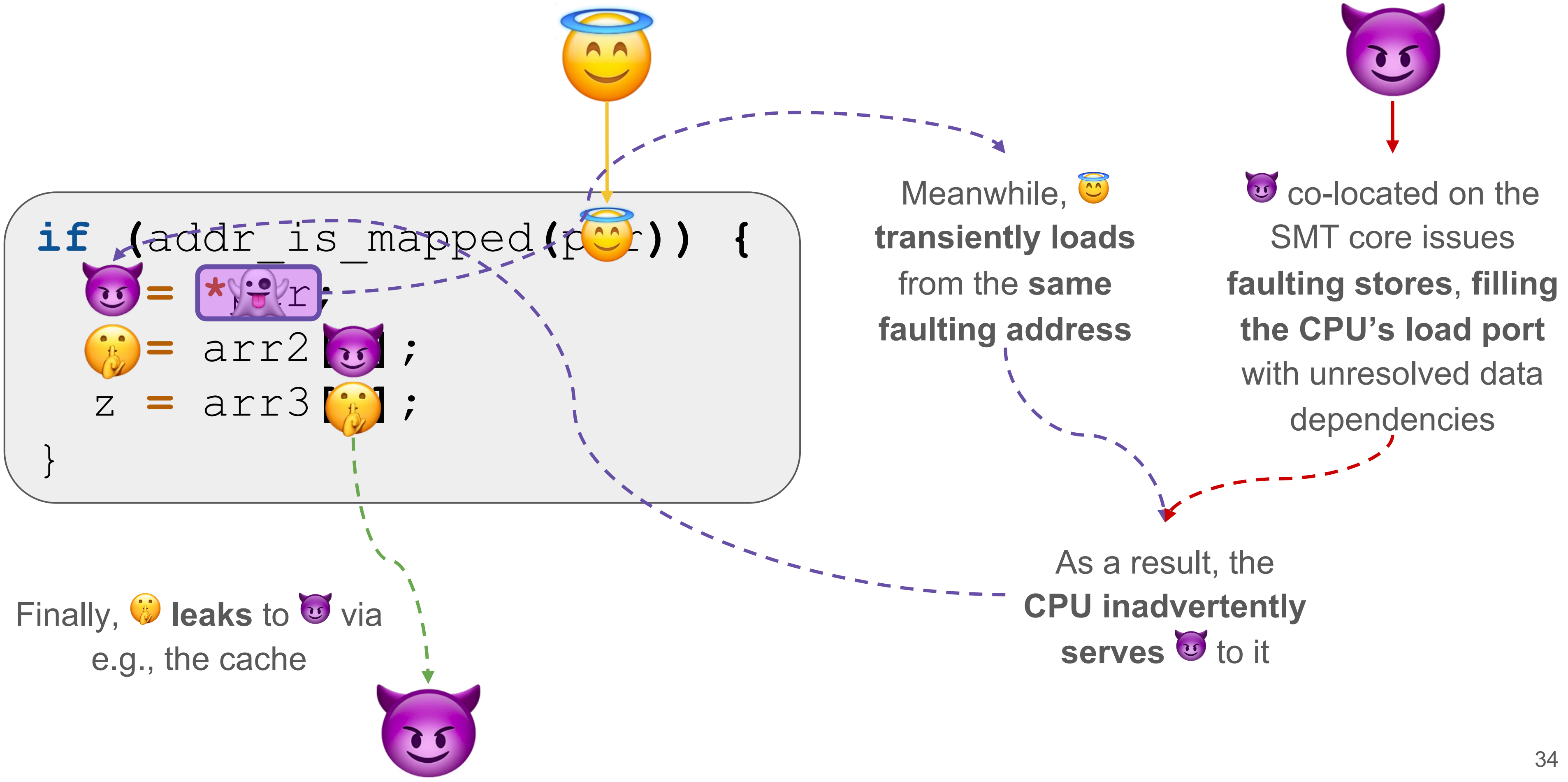
```
if (addr_is_mapped(p😇r)) {
   😈 = *👻r;
   🤫 = arr2[😈];
   z = arr3[🤫];
}
```

Meanwhile, 😇 **transiently loads** from the **same faulting address**

😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies

As a result, the **CPU inadvertently serves** 😈 to it

Background: Attacker injection via load value injection (LVI)



if (addr_is_mapped(p😇r)) {
😈 = *👻r;
🤫 = arr2[😈];
z = arr3[🤫];
}
Meanwhile, 😇 **transiently loads** from the **same faulting address**
😈 co-located on the SMT core issues **faulting stores**, **filling the CPU's load port** with unresolved data dependencies
As a result, the **CPU inadvertently serves** 😈 to it
Finally, 🤫 **leaks** to 😈 via e.g., the cache

Background: Secret leakage via MDS

Background: Secret leakage via MDS

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
}
```

Background: Secret leakage via MDS

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
}
```

Background: Secret leakage via MDS

```
😈 = get_user(ptr);
if (x < size) {
   y = arr1[x];
}
```

Background: Secret leakage via MDS

😈

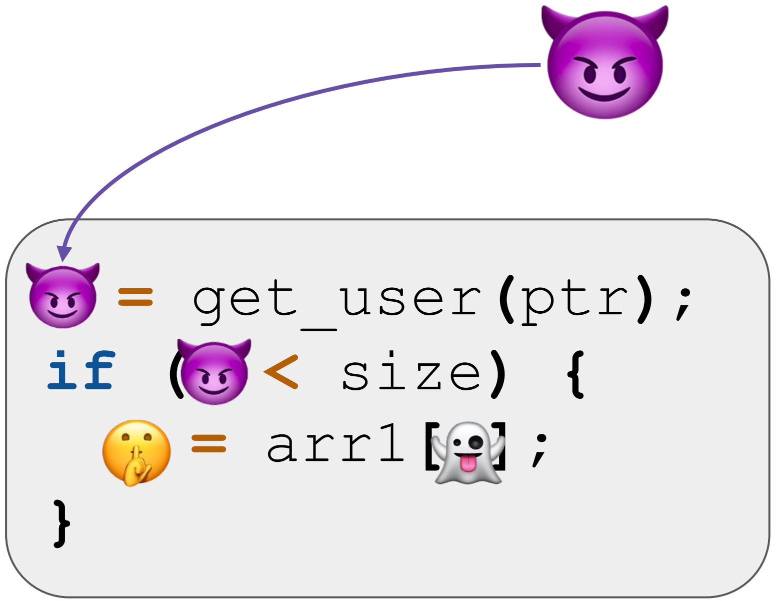
```
😈 = get_user(ptr);
if (😈 < size) {
   y = arr1[x];
}
```

Background: Secret leakage via MDS

😈

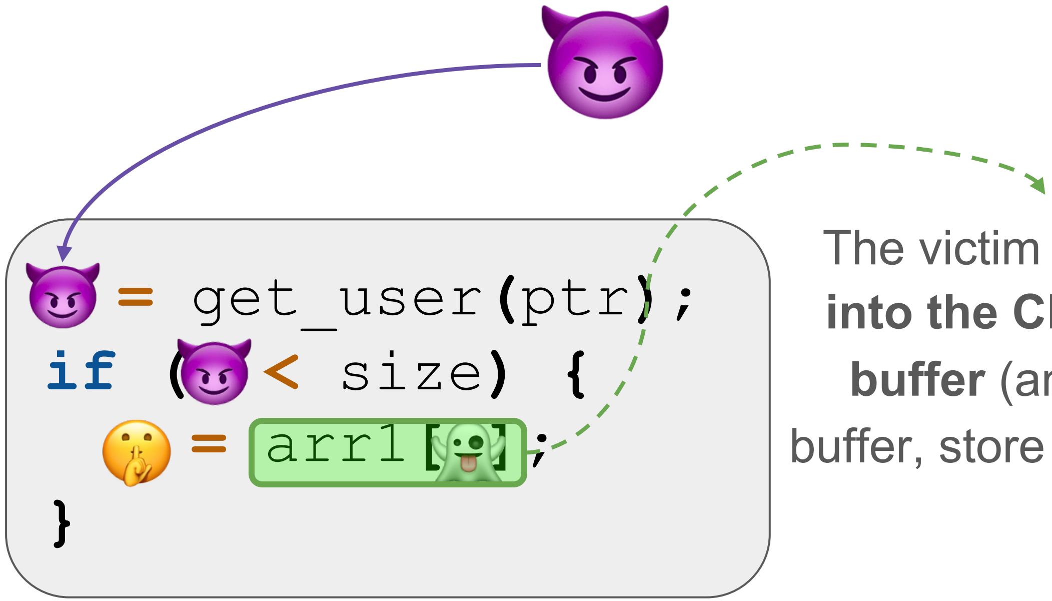
```
😈 = get_user(ptr);
if (😈 < size) {
    y = arr1[👻];
}
```

Background: Secret leakage via MDS



```
😈 = get_user(ptr);
if (😈 < size) {
  🤫 = arr1[👻];
}
```

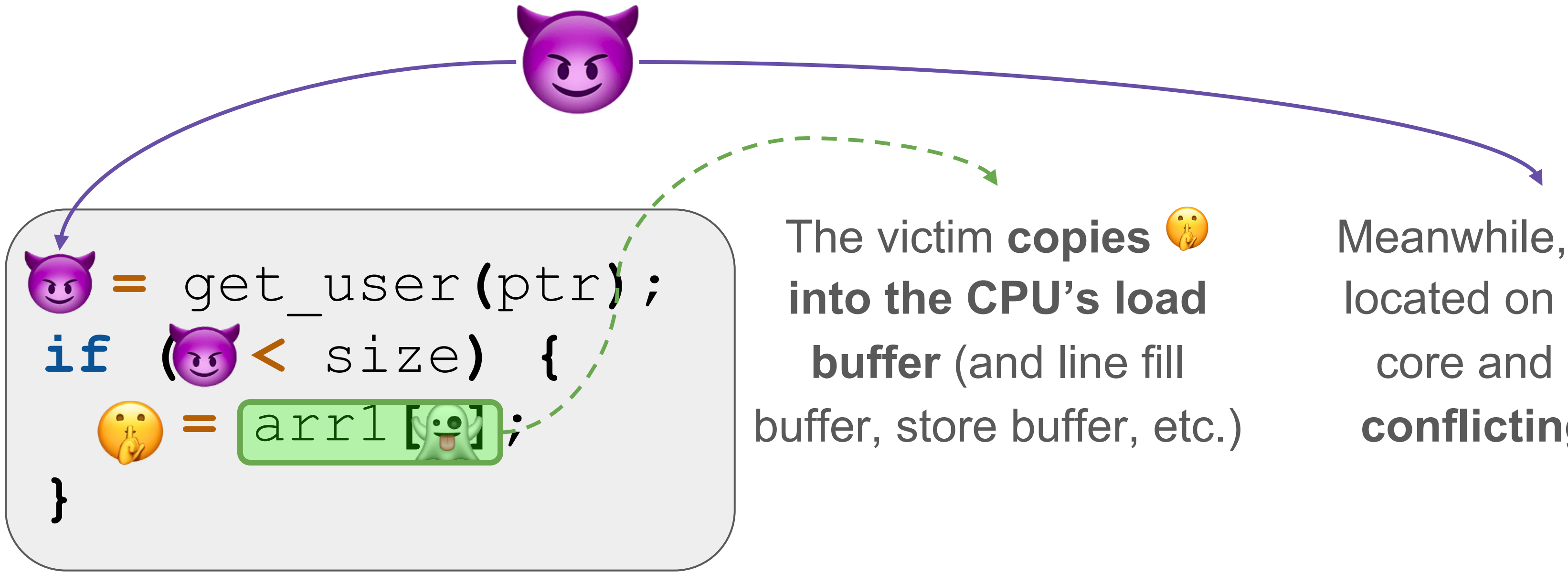
Background: Secret leakage via MDS



```
😈 = get_user(ptr);
if (😈 < size) {
  🤫 = arr1[👻];
}
```

The victim **copies** 🤫 **into the CPU’s load buffer** (and line fill buffer, store buffer, etc.)

Background: Secret leakage via MDS

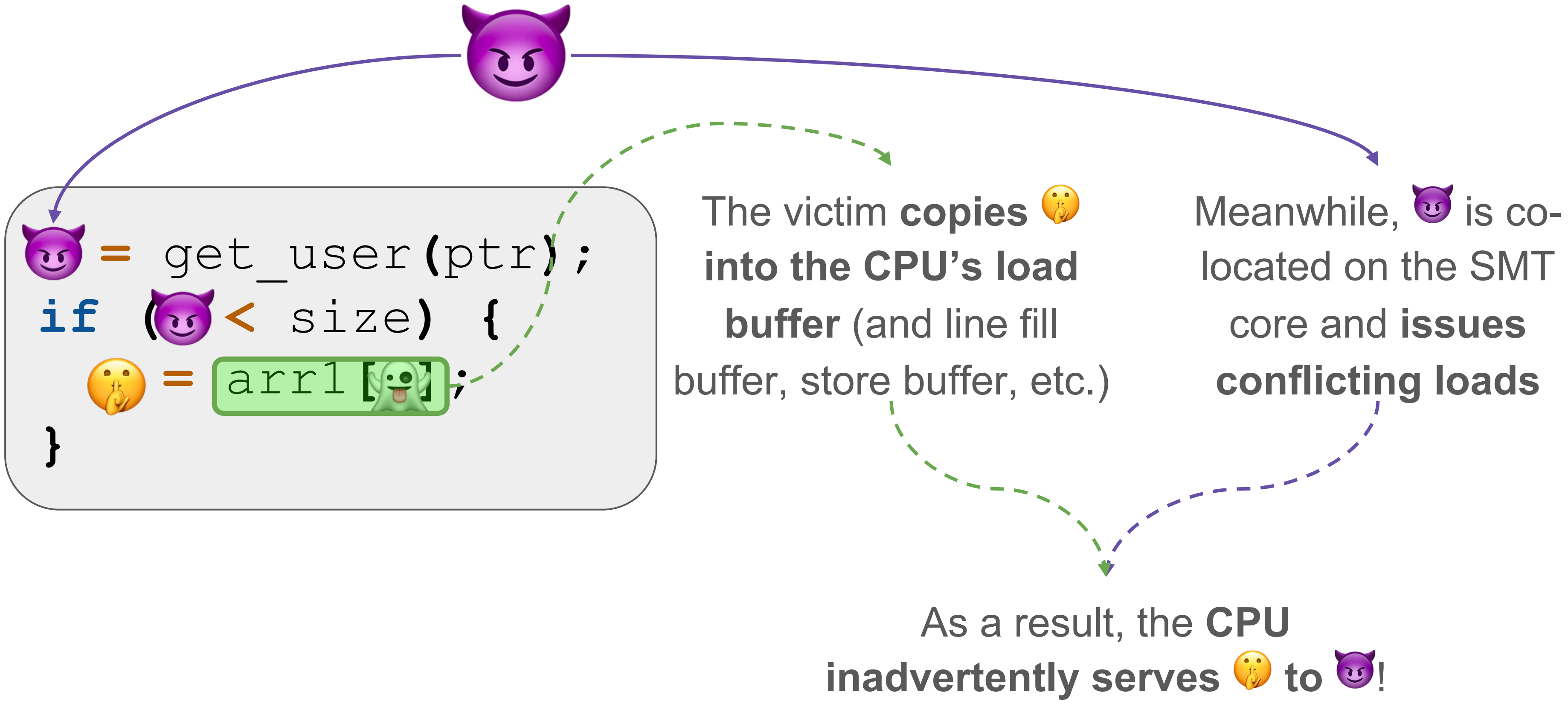


```
😈 = get_user(ptr);
if (😈 < size) {
  🤫 = arr1[👻];
}
```

The victim **copies** 🤫 **into the CPU's load buffer** (and line fill buffer, store buffer, etc.)

Meanwhile, 😈 is co-located on the SMT core and **issues conflicting loads**

Background: Secret leakage via MDS



```
😈 = get_user(ptr);
if (😈 < size) {
  🤫 = arr1[👻];
}
```

The victim **copies** 🤫 **into the CPU's load buffer** (and line fill buffer, store buffer, etc.)

Meanwhile, 😈 is co-located on the SMT core and **issues conflicting loads**

As a result, the **CPU inadvertently serves** 🤫 **to** 😈!

Background: Secret leakage via port contention

Background: Secret leakage via port contention

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  if (y) {
    ...
} }
```

Background: Secret leakage via port contention

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  if (y) {
    ...
} }
```

Background: Secret leakage via port contention

```
😈= get_user(ptr);
if (x < size) {
  y = arr1[x];
  if (y) {
    ...
} }
```

Background: Secret leakage via port contention

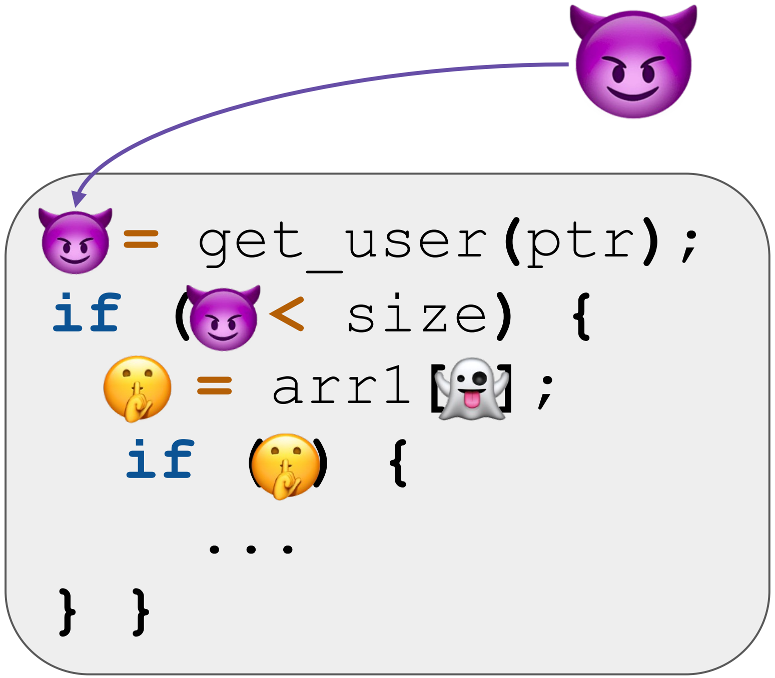
```
😈= get_user(ptr);
if (😈< size) {
   y = arr1[x];
   if (y) {
      ...
} }
```

Background: Secret leakage via port contention

😈

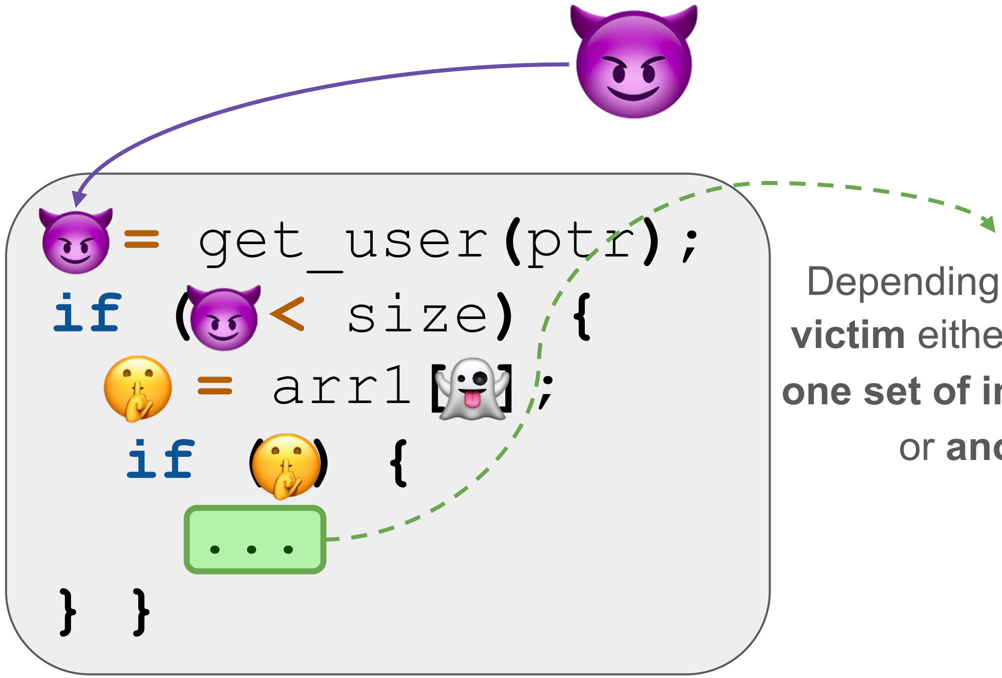
```
😈 = get_user(ptr);
if (😈 < size) {
   y = arr1[👻];
   if (y) {
      ...
} }
```

Background: Secret leakage via port contention



```
😈= get_user(ptr);
if (😈< size) {
  🤫 = arr1[👻];
   if (🤫) {
     ...
} }
```

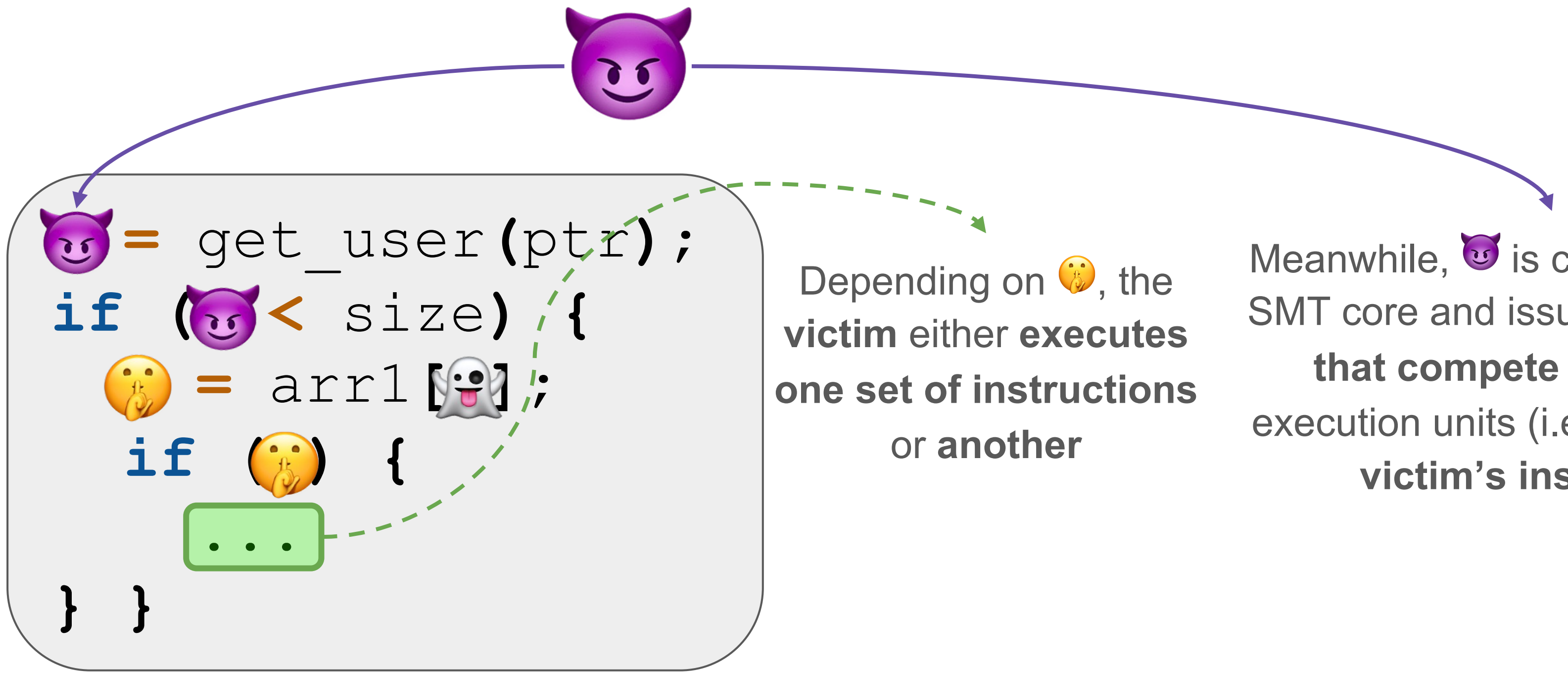
Background: Secret leakage via port contention



```
😈= get_user(ptr);
if (😈< size) {
  🤫 = arr1[👻];
   if (🤫) {
     ...
} }
```

Depending on 🤫, the **victim** either **executes one set of instructions** or **another**

Background: Secret leakage via port contention

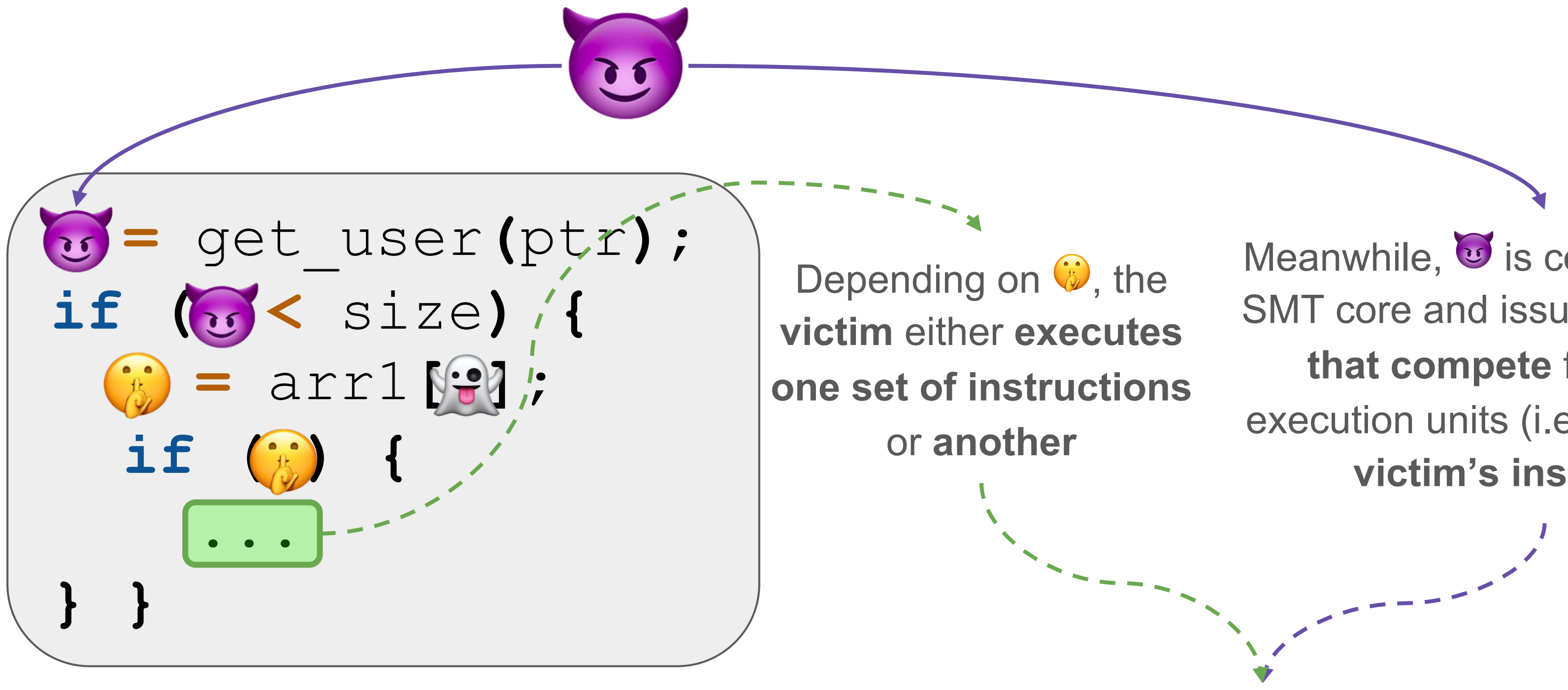


```
😈= get_user(ptr);
if (😈< size) {
  🤫 = arr1[👻];
  if (🤫) {
    ...
} }
```

Depending on 🤫, the **victim** either **executes one set of instructions** or **another**

Meanwhile, 😈 is co-located on the SMT core and issues **instructions that compete** for the same execution units (i.e., **ports**) **as the victim's instructions**

Background: Secret leakage via port contention



```
😈= get_user(ptr);
if (😈< size) {
  🤫 = arr1[👻];
   if (🤫) {
     ...
} }
```

Depending on 🤫, the **victim** either **executes one set of instructions** or **another**

Meanwhile, 😈 is co-located on the SMT core and issues **instructions that compete** for the same execution units (i.e., **ports**) **as the victim's instructions**

As a result, 😈 can use timing information to infer **which instructions the victim executed**, and hence, **learn a bit of** 🤫!

Taint policies: Attacker injection

Taint policies: Attacker injection

Apply an `attacker` label to…

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Data **loaded from out-of-bounds accesses**, i.e., data prone to **memory massaging**

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Data **loaded from out-of-bounds accesses**, i.e., data prone to **memory massaging**

```
if (a < size) {
  b = arr1[a];
  c = arr2[b];
  d = arr3[c];
}
```

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Data **loaded from out-of-bounds accesses**, i.e., data prone to **memory massaging**

```
if (a < size) {
  b = arr1[a];
  c = arr2[b];
  d = arr3[c];
}
```

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Data **loaded from out-of-bounds accesses**, i.e., data prone to **memory massaging**

```
if (a < size) {
  b = arr1[a];
  c = arr2[b];
  d = arr3[c];
}
```

Data **loaded from invalid accesses**, i.e., data prone to **LVI**

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Data **loaded from out-of-bounds accesses**, i.e., data prone to **memory massaging**

```
if (a < size) {
  b = arr1[a];
  c = arr2[b];
  d = arr3[c];
}
```

Data **loaded from invalid accesses**, i.e., data prone to **LVI**

```
if (addr_is_mapped(ptr)) {
  x = *ptr;
  y = arr2[x];
  z = arr3[y];
}
```

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Data **loaded from out-of-bounds accesses**, i.e., data prone to **memory massaging**

```
if (a < size) {
  b = arr1[a];
  c = arr2[b];
  d = arr3[c];
}
```

Data **loaded from invalid accesses**, i.e., data prone to **LVI**

```
if (addr_is_mapped(ptr)) {
  x = *ptr;
  y = arr2[x];
  z = arr3[y];
}
```

Taint policies: Attacker injection

Apply an `attacker` label to…

Data **copied from userspace** (e.g., from syscall arguments, `copy_from_user()`, `get_user()`)

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Data **loaded from out-of-bounds accesses**, i.e., data prone to **memory massaging**

```
if (a < size) {
  b = arr1[a];
  c = arr2[b];
  d = arr3[c];
}
```

Data **loaded from invalid accesses**, i.e., data prone to **LVI**

```
if (addr_is_mapped(ptr)) {
  x = *ptr;
  y = arr2[x];
  z = arr3[y];
}
```

Taint policies: Secret access

Taint policies: Secret access

If a load with an `attacker` pointer is unsafe, then taint the output as a `secret`.

Taint policies: Secret access

If a load with an `attacker` pointer is unsafe, then taint the output as a `secret`.

(There are a few more details here, so if interested, refer to our paper 🤓).

Taint policies: Secret access

If a load with an `attacker` pointer is unsafe, then taint the output as a `secret`.

(There are a few more details here, so if interested, refer to our paper 🤓).

```
x = get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

Taint policies: Secret access

If a load with an `attacker` pointer is unsafe, then taint the output as a `secret`.

(There are a few more details here, so if interested, refer to our paper 🤓).

```
x =
get_user(ptr);
if (x < size) {
   y = arr1[x];
   z = arr2[y];
}
```

Taint policies: Secret leakage

When…

Taint policies: Secret leakage

When…

A memory access has a `secret` **pointer**, report a **CACHE gadget**.

Taint policies: Secret leakage

When…

A memory access has a `secret` **pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Taint policies: Secret leakage

When…

A memory access has a `secret` **pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

Taint policies: Secret leakage

When…

A memory access has a `secret` **pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

A memory access outputs a `secret`, report an **MDS gadget**.

Taint policies: Secret leakage

When…

A memory access has a `secret` **pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

A memory access outputs a `secret`, report an **MDS gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
}
```

Taint policies: Secret leakage

When…

A memory access has a **secret pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

A memory access outputs a **secret**, report an **MDS gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
}
```

Taint policies: Secret leakage

When…

A memory access has a `secret` **pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

A memory access outputs a `secret`, report an **MDS gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
}
```

A branch has a `secret` target, report a **PORT gadget**.

Taint policies: Secret leakage

When…

A memory access has a **secret pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

A memory access outputs a **secret**, report an **MDS gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
}
```

A branch has a **secret** target, report a **PORT gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  if (y) {
    ...
} }
```

Taint policies: Secret leakage

When…

A memory access has a `secret` **pointer**, report a **CACHE gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  z = arr2[y];
}
```

A memory access outputs a `secret`, report an **MDS gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
}
```

A branch has a `secret` target, report a **PORT gadget**.

```
x = get_user(ptr);
if (x < size) {
  y = arr1[x];
  if (y) {
    ...
} }
```