A Tough `call`: Mitigating Advanced Code-Reuse Attacks At The Binary Level

Victor van der Veen, Enes Göktaş (joint first author) (VU)

Moritz Contag, Andre Pawlowski, Thorsten Holz (RUB)

Xi Chen, Sanjay Rawat, Herbert Bos, Elias Athanasopoulos, Cristiano Giuffrida (VU)

Control-Flow Integrity

- Promising way to stop code-reuse attacks
- Hard to enforce in practice
- Existing binary-level CFI cannot prevent function-reuse attacks (COOP)

Control-Flow Integrity

- Promising way to stop code-reuse attacks
- Hard to enforce in practice
- Existing binary-level CFI cannot prevent function-reuse attacks (COOP)

TypeArmor

- A more precise binary-level CFI solution
- Acceptable overhead (3% on SPEC)
- Stops all published code-reuse attacks

Running example: normal execution

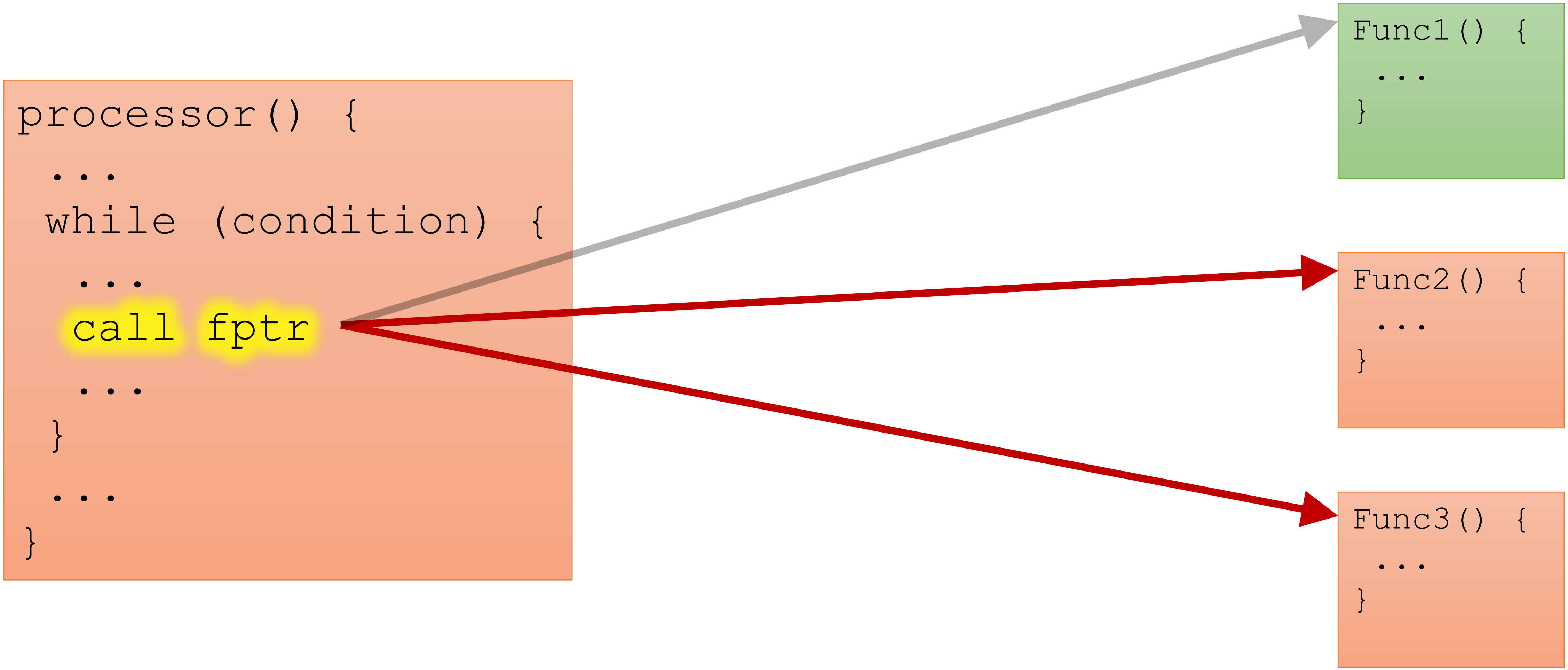
```
processor() {
 ...
 while (condition) {
  ...
  call fptr
  ...
 }
 ...
}
```

```
Func1() {
 ...
}
```

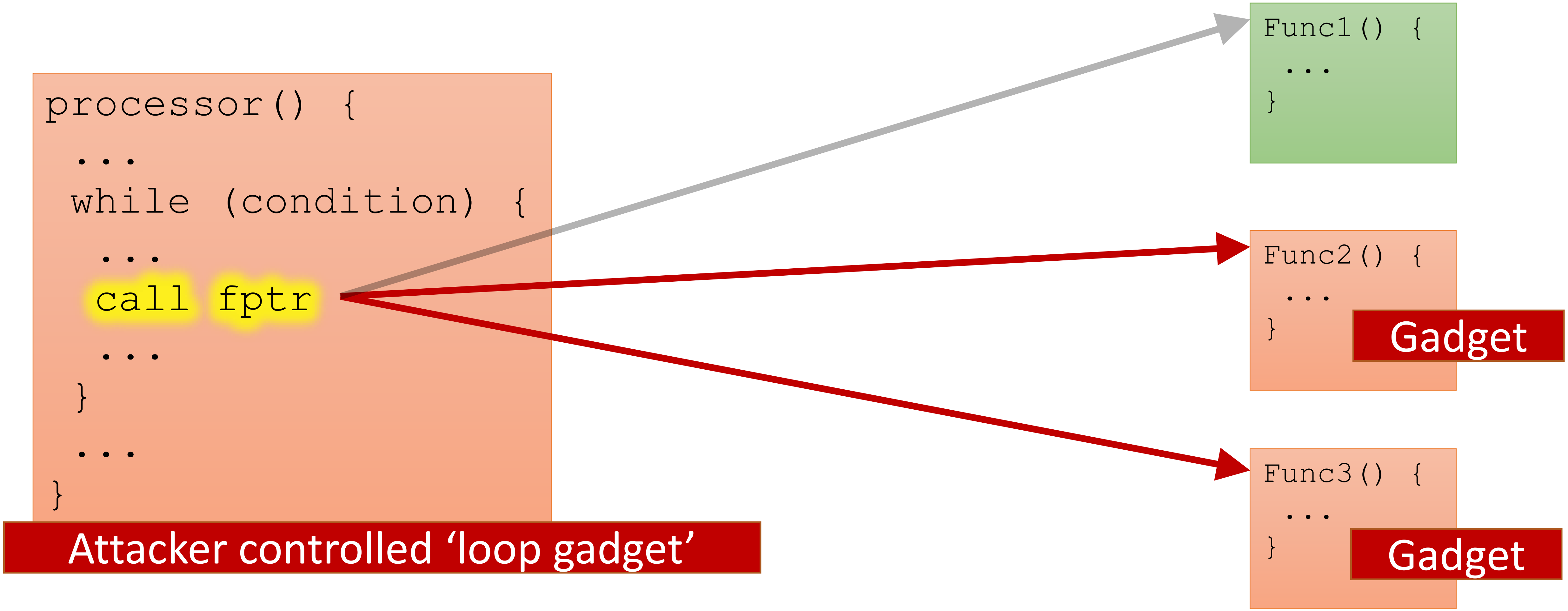
```
Func2() {
 ...
}
```

```
Func3() {
 ...
}
```

Running example: advanced code-reuse



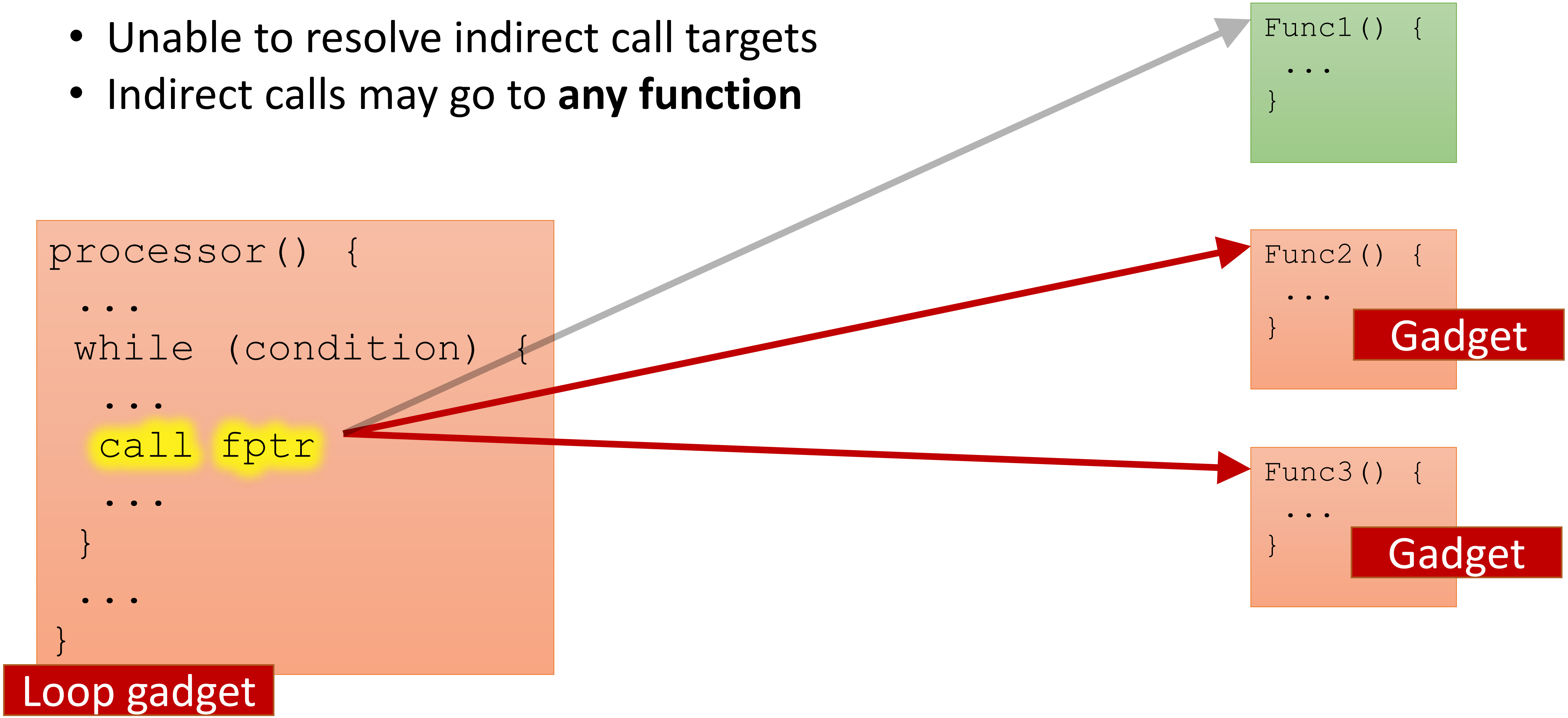
Running example: advanced code-reuse



Function-oriented programming

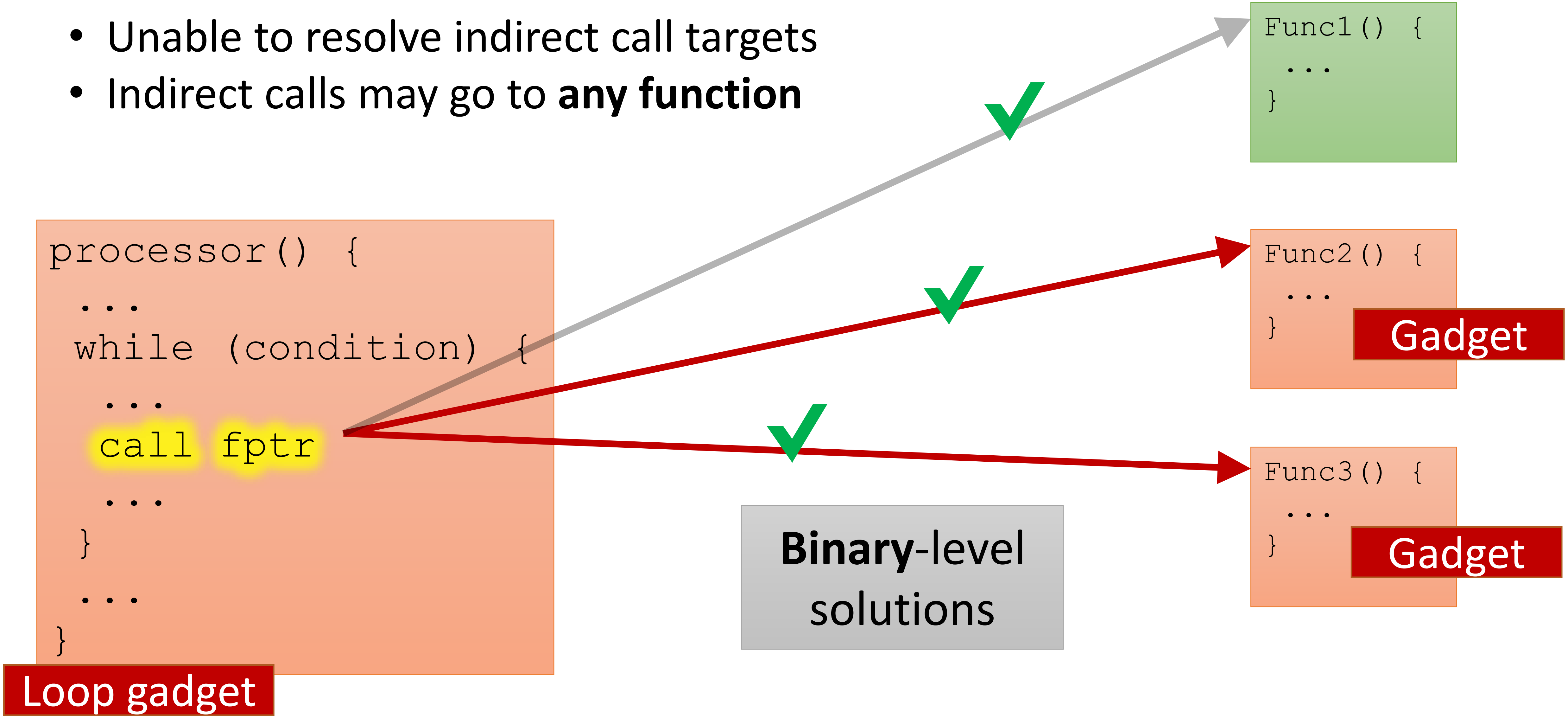
Running example: binary-level CFI

- Unable to resolve indirect call targets
- Indirect calls may go to **any function**



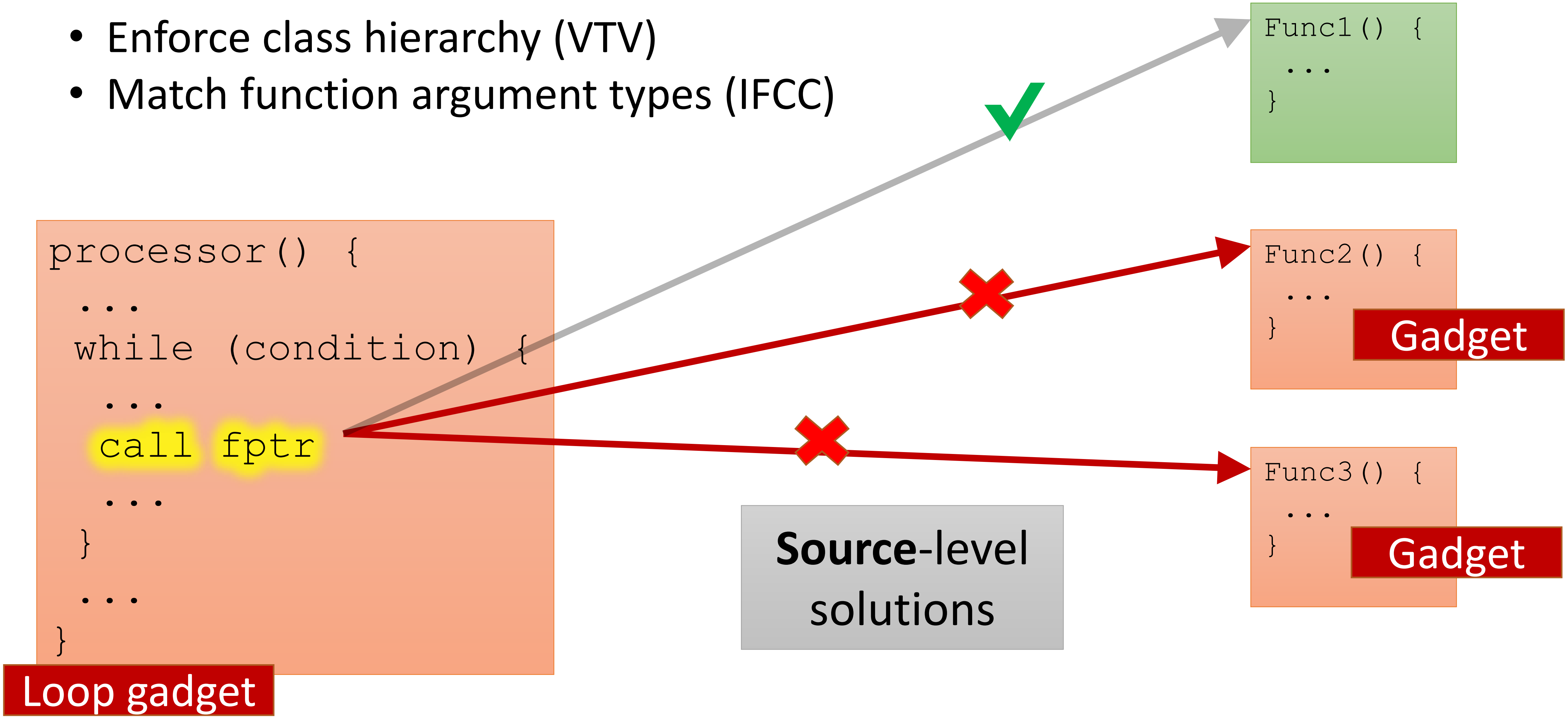
Running example: binary-level CFI

- Unable to resolve indirect call targets
- Indirect calls may go to **any function**



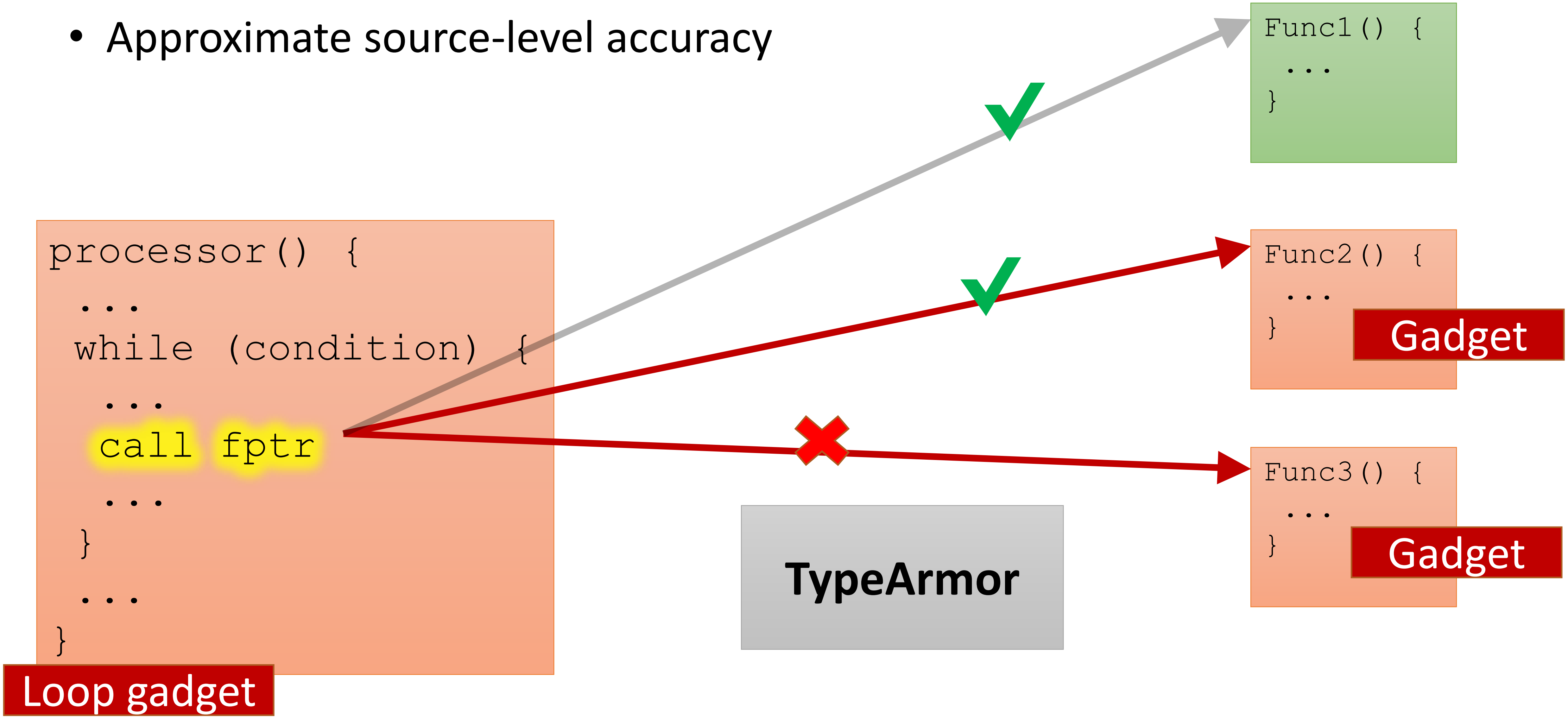
Running example: source-level CFI

- Enforce class hierarchy (VTV)
- Match function argument types (IFCC)



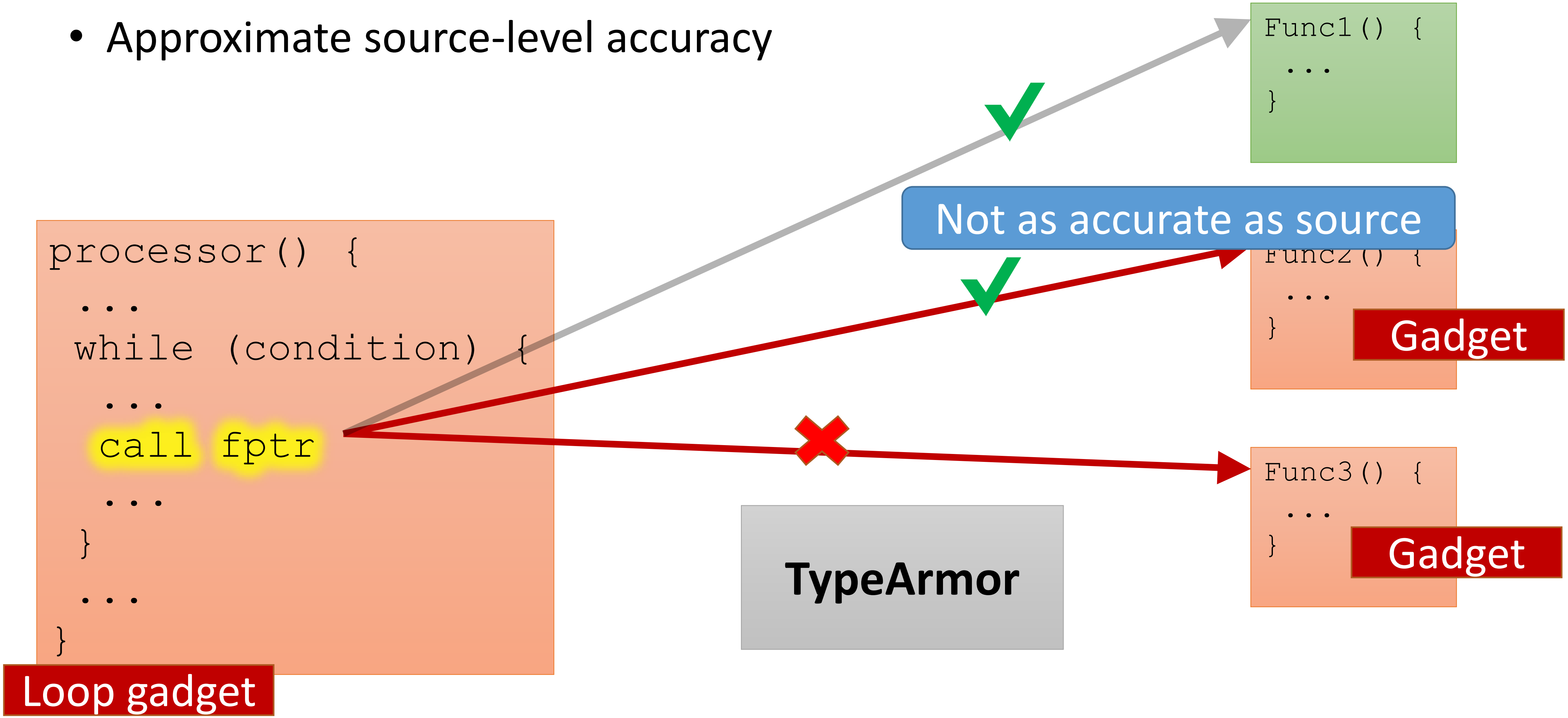
Running example: TypeArmor

- Approximate source-level accuracy



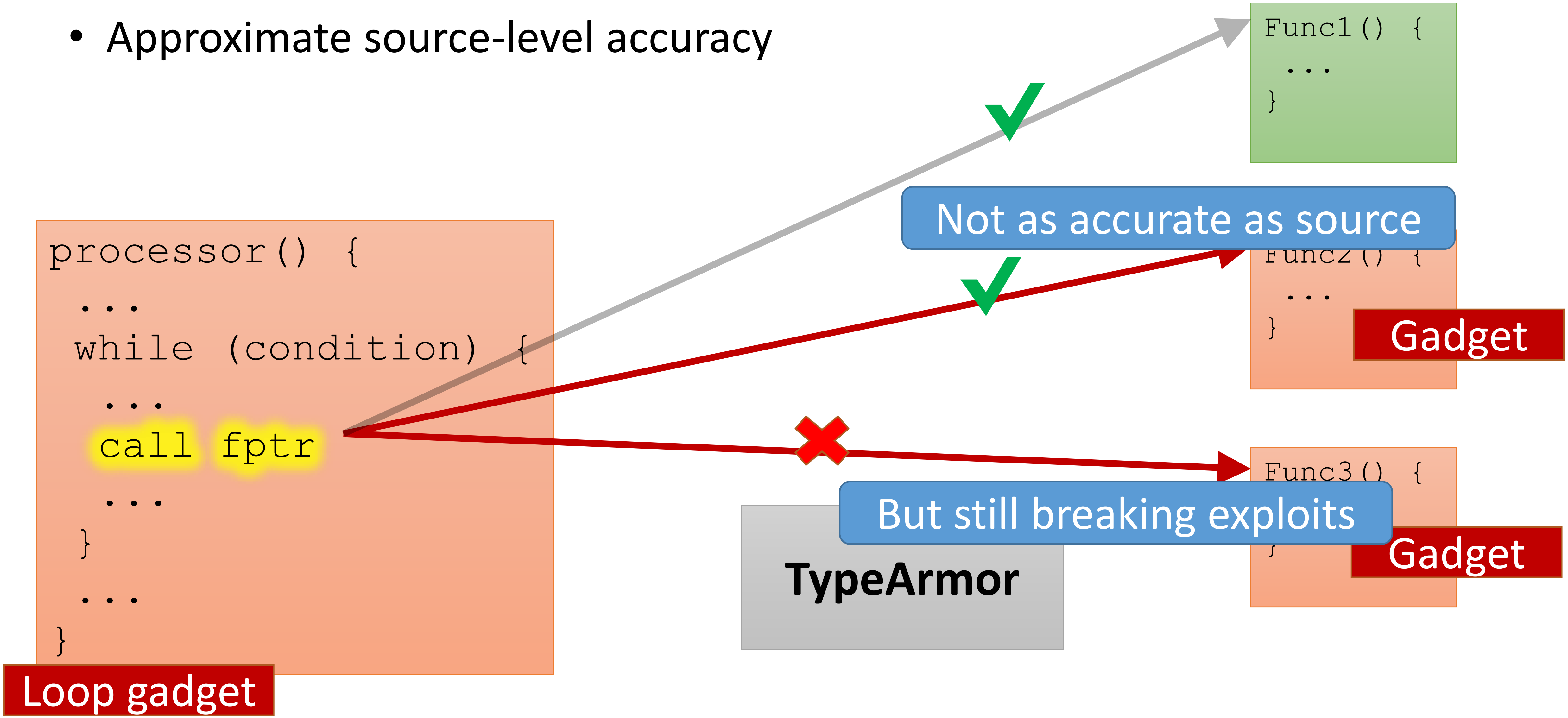
Running example: TypeArmor

- Approximate source-level accuracy



Running example: TypeArmor

- Approximate source-level accuracy



Approximate source-level invariants?

Function signature matching by argcount

- Extract argument count at callsite
- Extract argument usage at callee
- Allow only targets with matching function types

Approximate source-level invariants?

Function signature matching by argcount

- Extract argument count at callsite
- Extract argument usage at callee
- Allow only targets with matching function types

Callsites preparing **two** args should never call functions expecting **three or more**

Approximate source-level invariants?

Function signature matching by argcount

- Extract argument count at callsite
- Extract argument usage at callee
- Allow only targets with matching function types

Callsites preparing **two** args should never call functions expecting **three or more**

Implemented for the x86-64 architecture:

- Calling convention: pass arguments via registers
- Search for write instructions at the callsite
- Search for read-before-write instructions at the callee

Running example: TypeArmor

- Match argument count expectations

```
processor() {
 ...
 while (condition){
  arg1 = x
  arg2 = y
  call fptr(arg1,arg2)
  ...
 }
 ...
```

Loop gadget

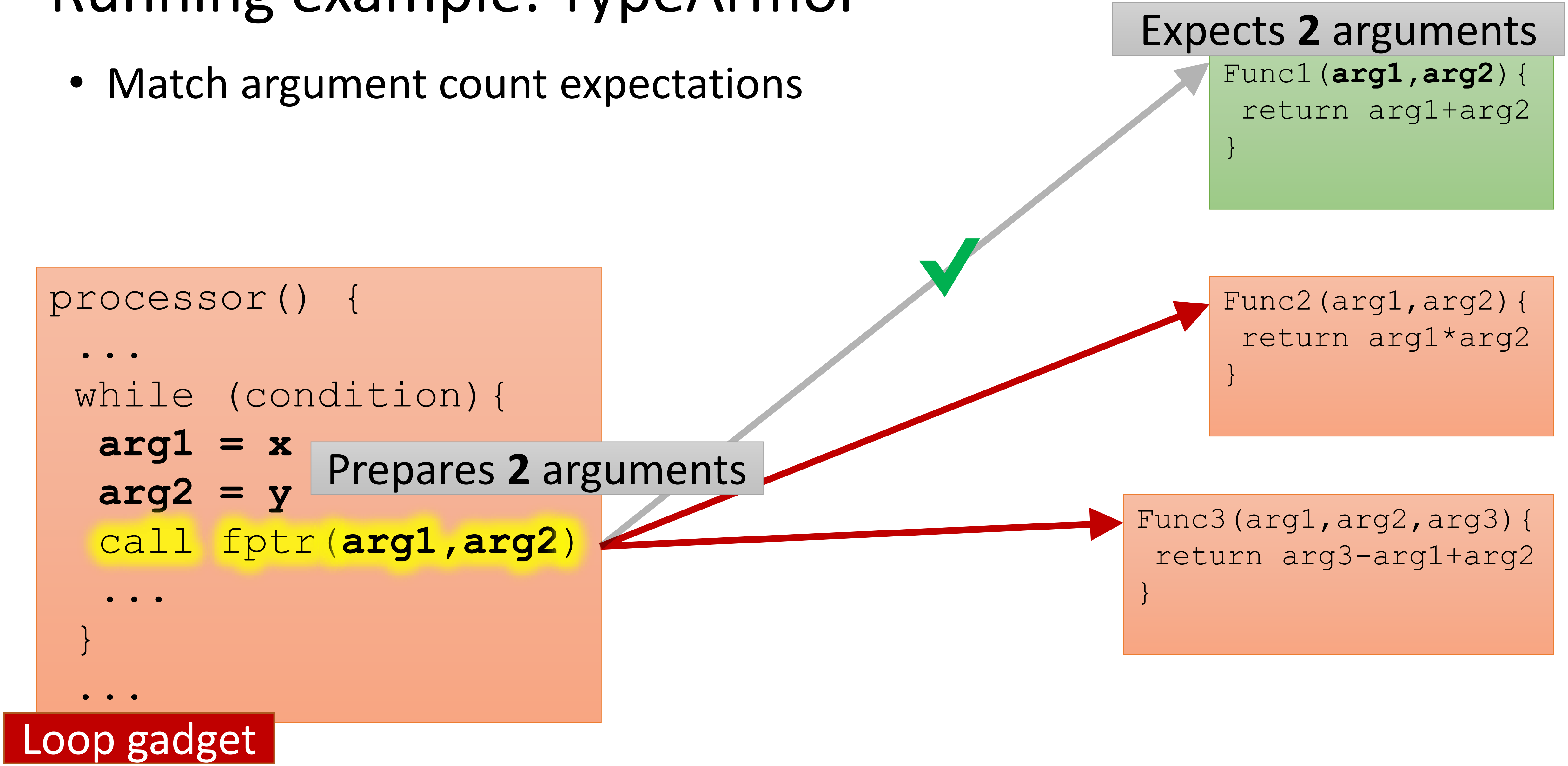
```
Func1(arg1,arg2){
 return arg1+arg2
}
```

```
Func2(arg1,arg2){
 return arg1*arg2
}
```

```
Func3(arg1,arg2,arg3){
 return arg3-arg1+arg2
}
```

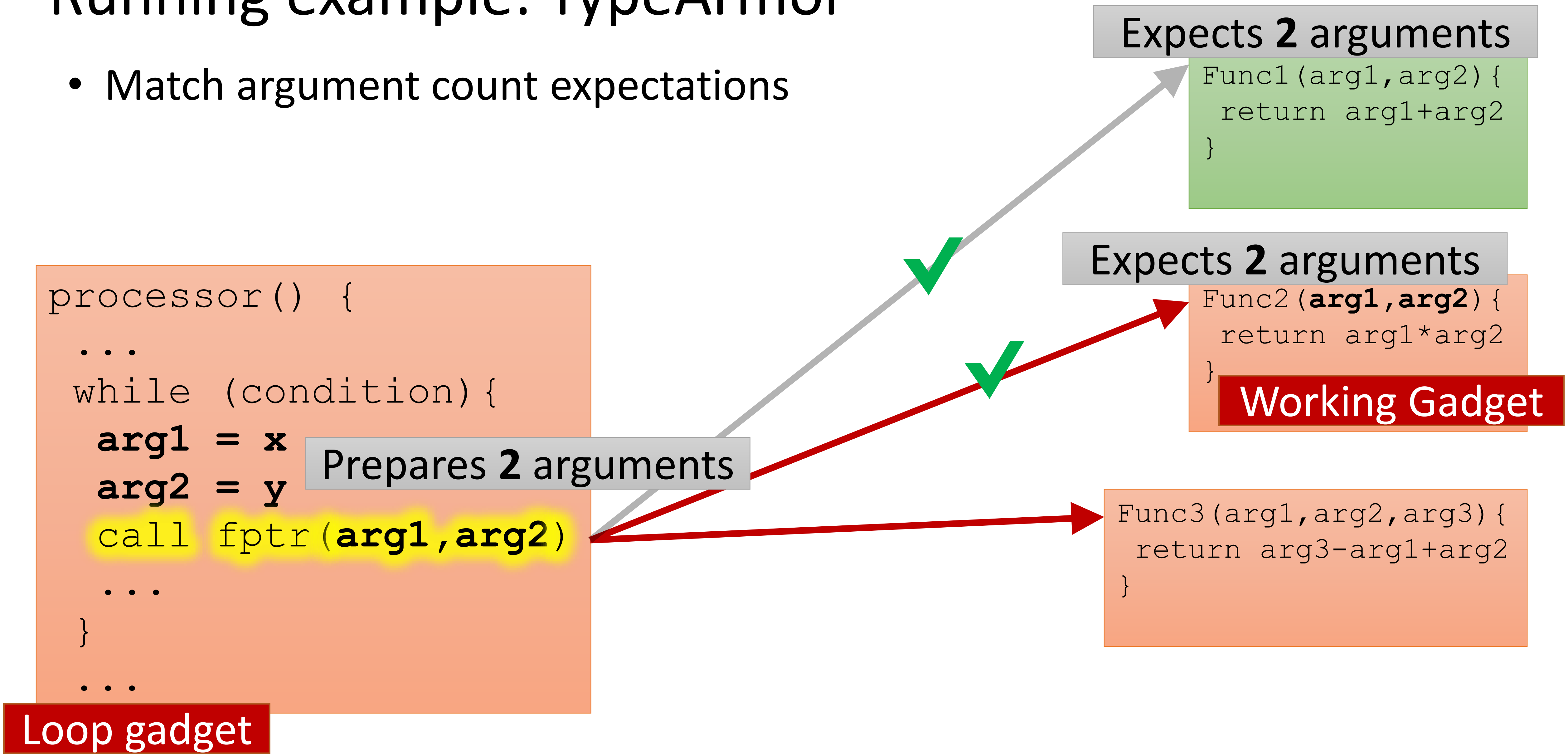
Running example: TypeArmor

- Match argument count expectations



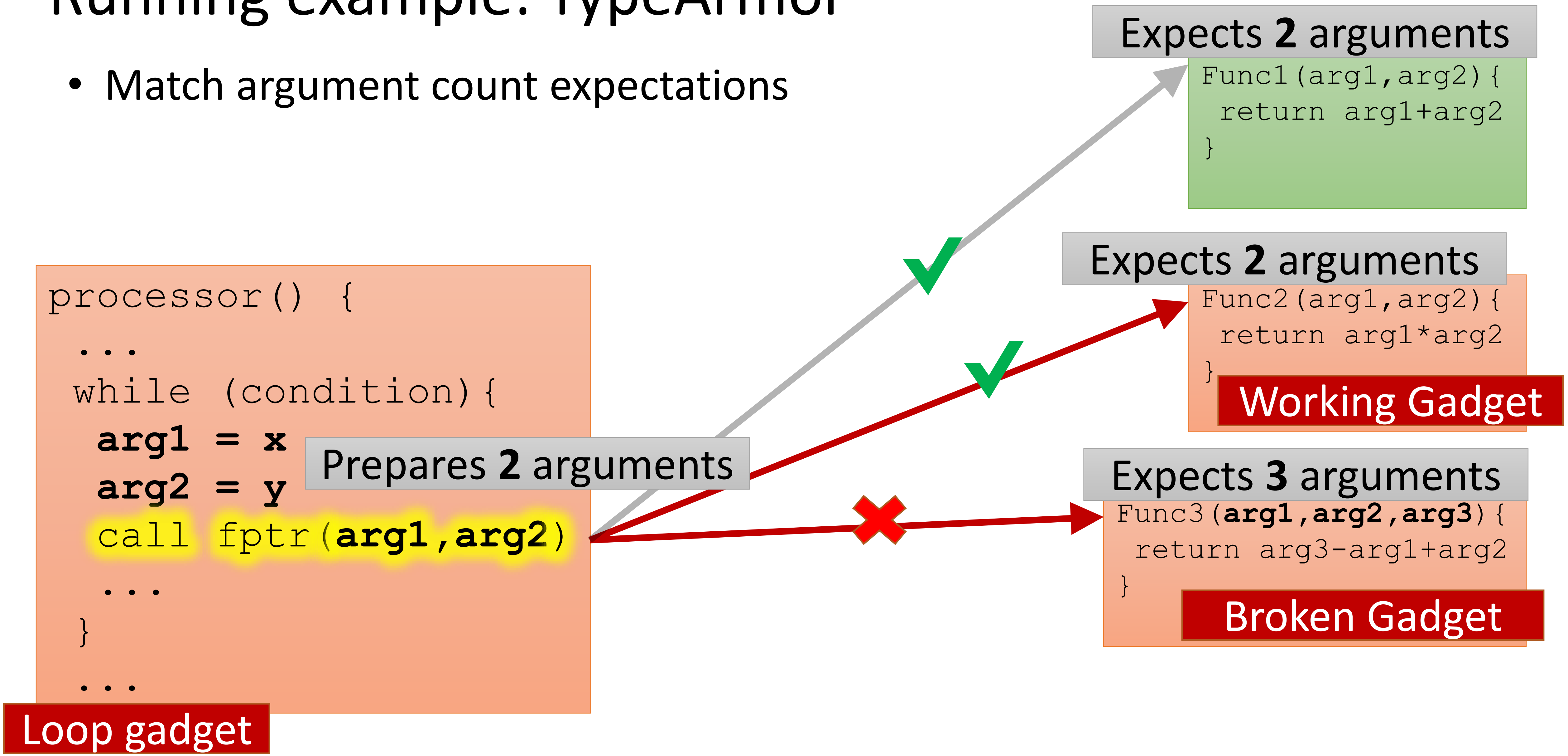
Running example: TypeArmor

- Match argument count expectations



Running example: TypeArmor

- Match argument count expectations



Precision

How accurate can we determine the prepared and used argument count?

	Callsites		Functions	
Server	**#**	**As in source**	**#**	**As in source**
Memcached	48	41 (86%)	236	210 (89%)
lighttpd	54	47 (87%)	353	311 (88%)
Nginx	218	161 (74%)	1,111	869 (78%)
MySQL	7,532	5,771 (77%)	9,961	6,977 (70%)

Precision

How accurate can we determine the prepared and used argument count?

Server	Callsites		Functions	
	#	As in source	#	As in source
Memcached	48	41 (86%)	236	210 (89%)
lighttpd	54	47 (87%)	353	311 (88%)
Nginx	218	161 (74%)	1,111	869 (78%)
MySQL	7,532	5,771 (77%)	9,961	6,977 (70%)

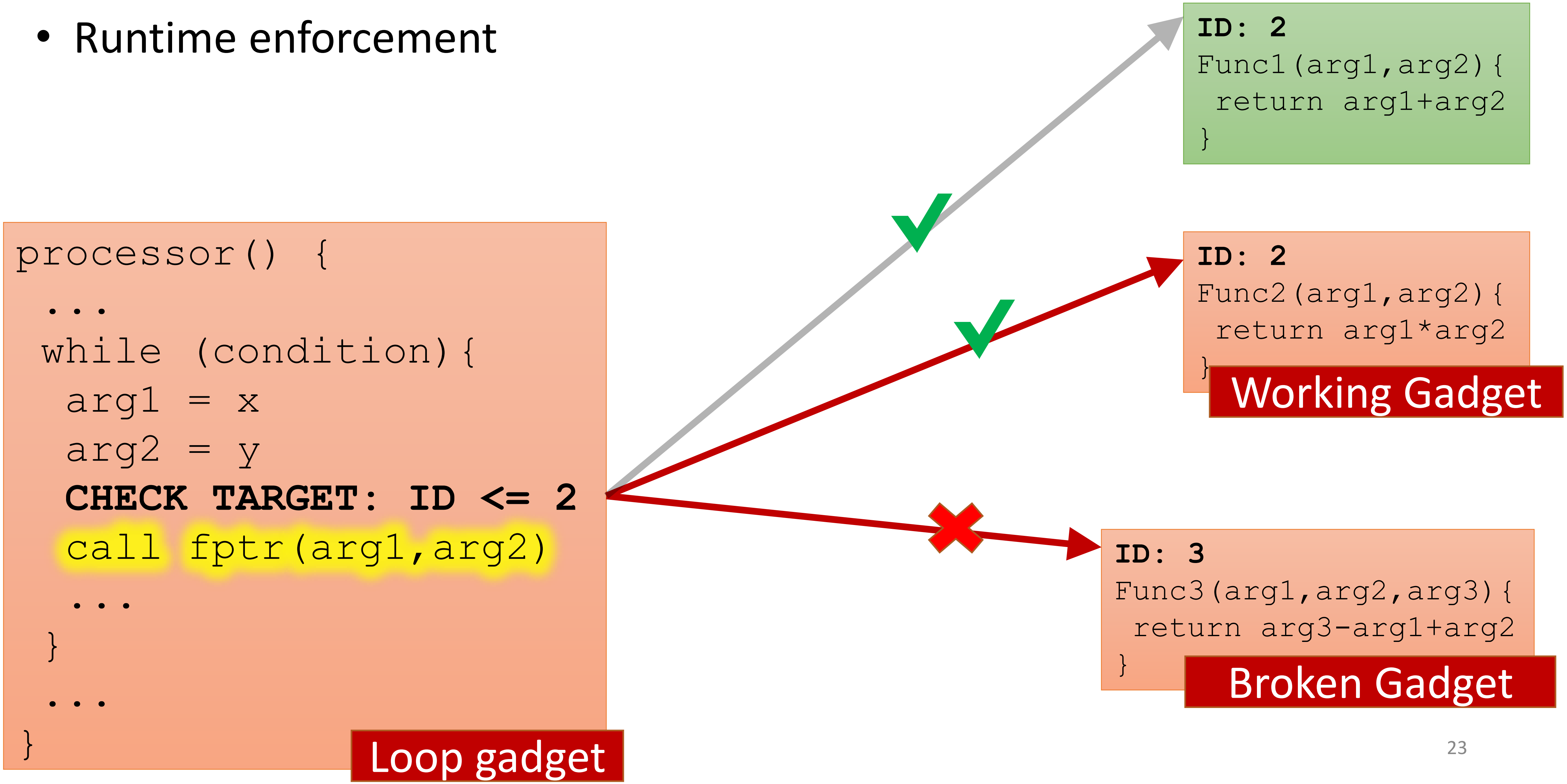
Precision

How accurate can we determine the prepared and used argument count?

Server	Callsites		Functions	
	#	As in source	#	As in source
Memcached	48	41 (86%)	236	210 (89%)
lighttpd	54	47 (87%)	353	311 (88%)
Nginx	218	161 (74%)	1,111	869 (78%)
MySQL	7,532	5,771 (77%)	9,961	6,977 (70%)

Running example: TypeArmor

- Runtime enforcement



Performance

SPEC CPU2006: less than 3% (geometric mean)

Performance

SPEC CPU2006: less than 3% (geometric mean)

Server	Overhead	Language
Memcached	1.4%	C
lighttpd	11.6%	C
Nginx	13.2%	C
MySQL	23.9%	C++

Conclusion

- Extract new invariants from binaries
- Enforce strictest security policy at binary-level to date
- Binary-level CFI solutions **can** mitigate sophisticated code-reuse attacks
- Keep an eye on http://www.vusec.net

VRIJE UNIVERSITEIT AMSTERDAM

RUHR UNIVERSITÄT BOCHUM