Decompiling for Constant-Time Analysis

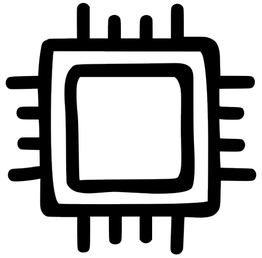
Decompilers disturb Formal Side Channel Security!

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter,
Youcef Bouzid, Sören van der Wall, Zhiyuan Zhang

PriSC 2026
Rennes

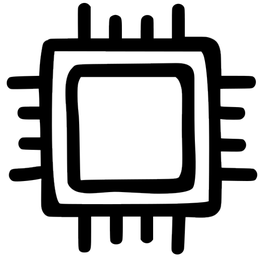
Formal Side-Channel Security 101

Attacker Model



Formal Side-Channel Security 101

Attacker Model

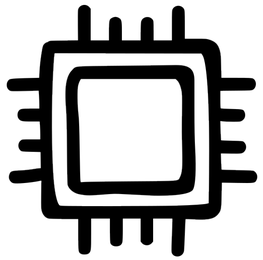


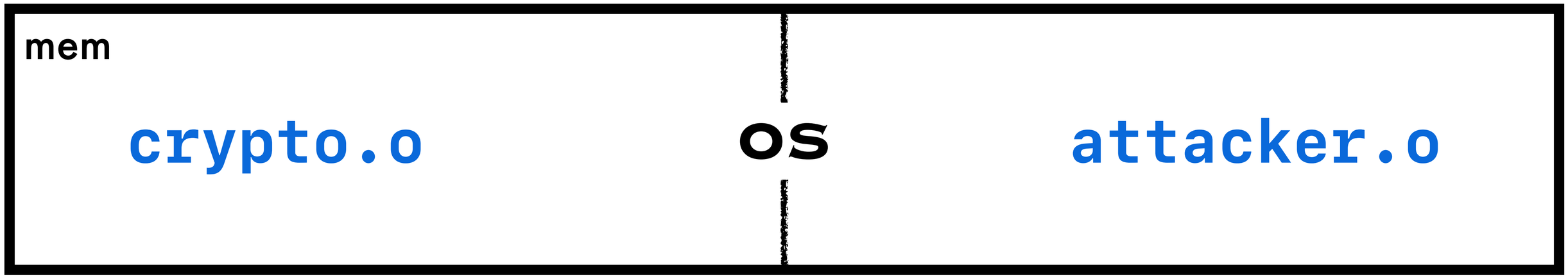
mem

`crypto.o` `attacker.o`

Formal Side-Channel Security 101

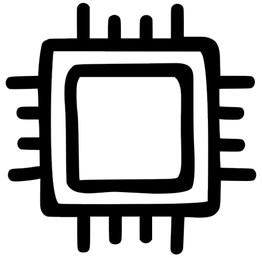
Attacker Model





Formal Side-Channel Security 101

Attacker Model



cache

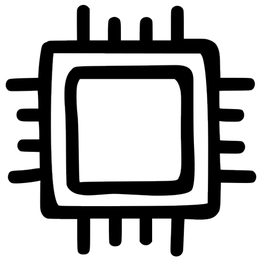
branch predictor

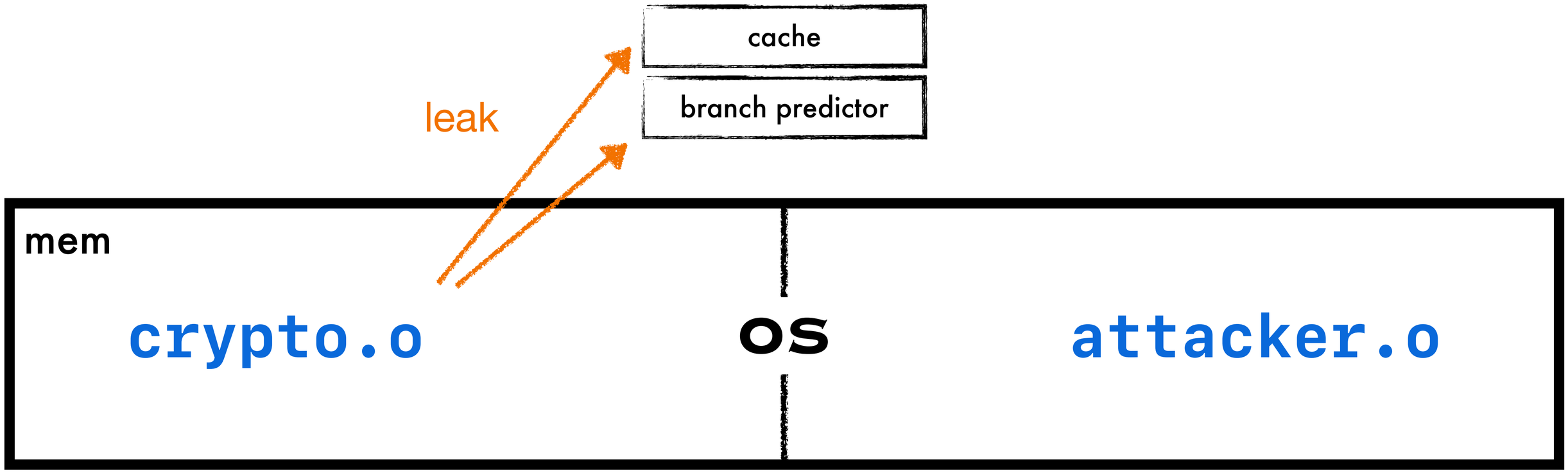
mem

`crypto.o` | OS | `attacker.o`

Formal Side-Channel Security 101

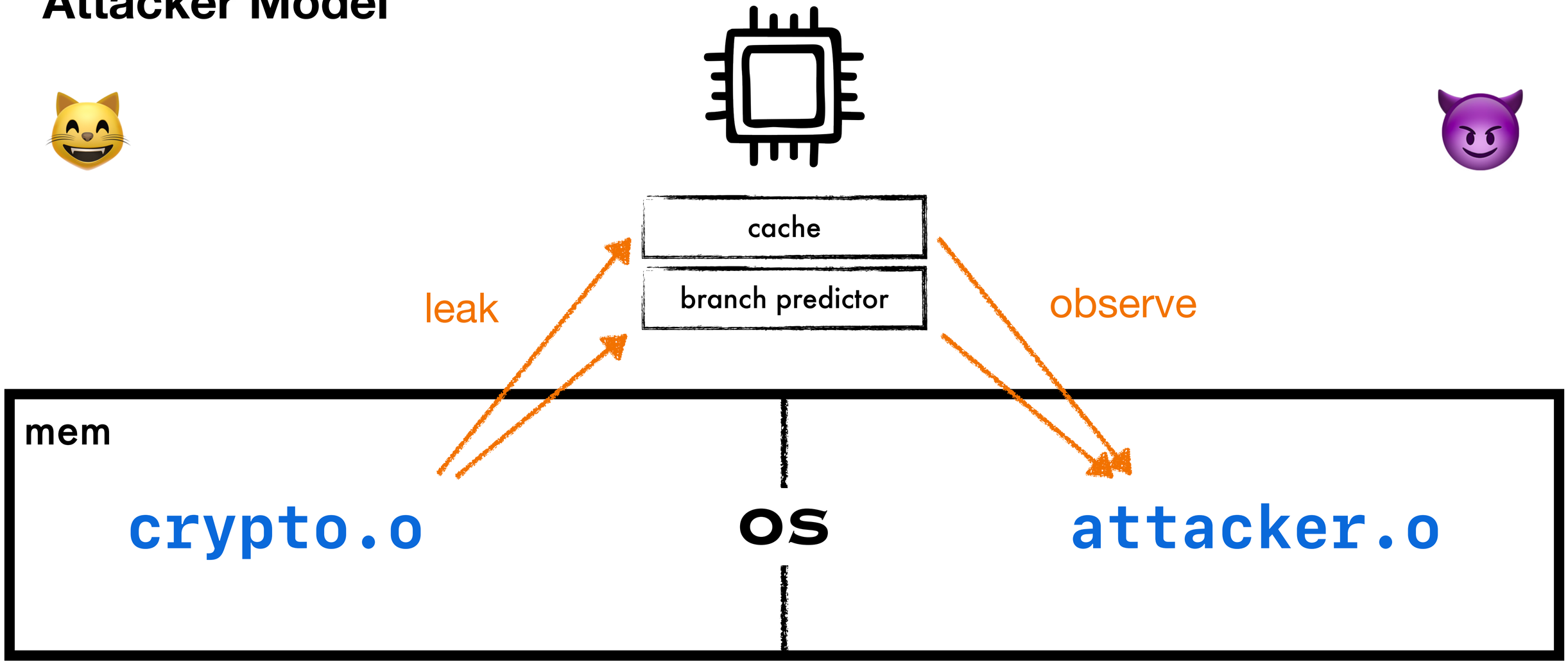
Attacker Model





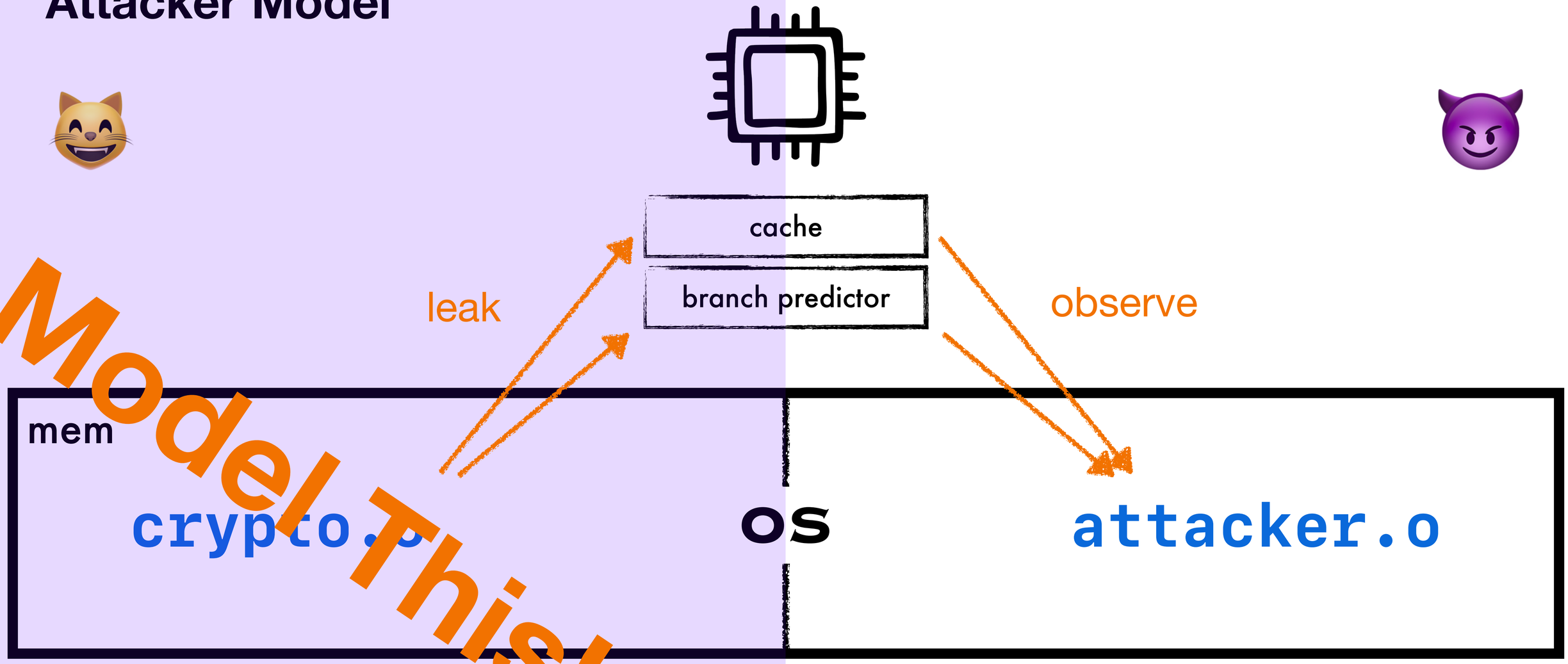
Formal Side-Channel Security 101

Attacker Model

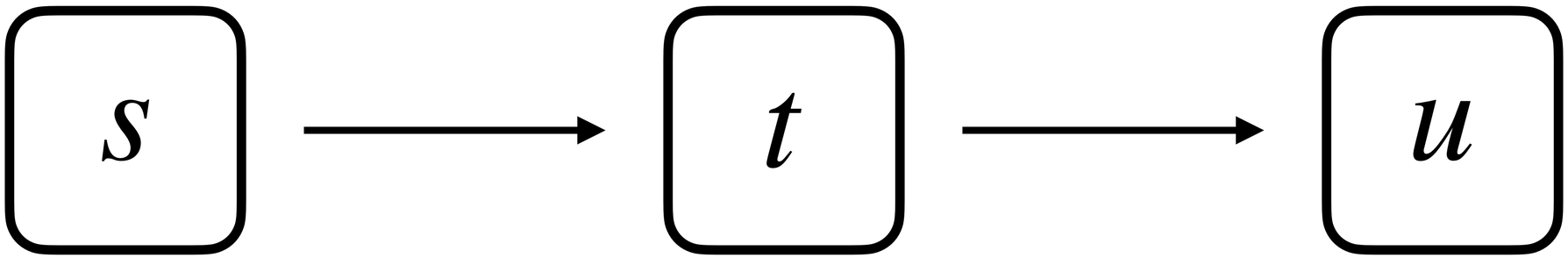


Formal Side-Channel Security 101

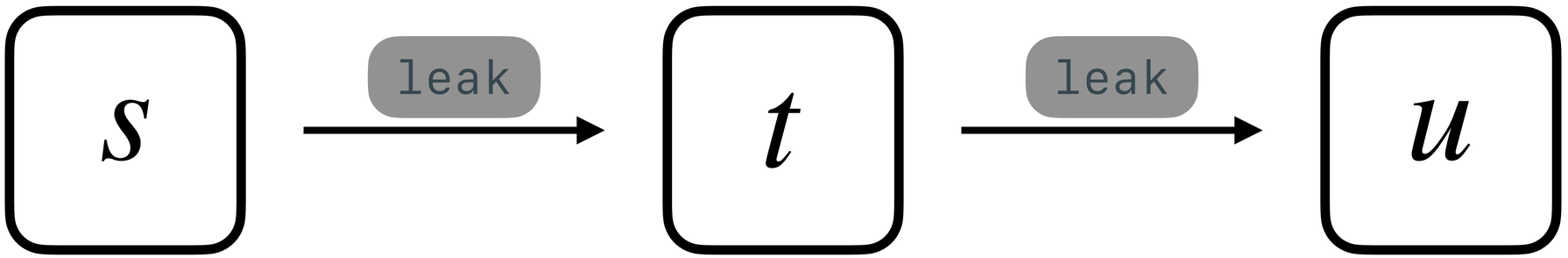
Attacker Model



Model: CT-Leakage Semantics

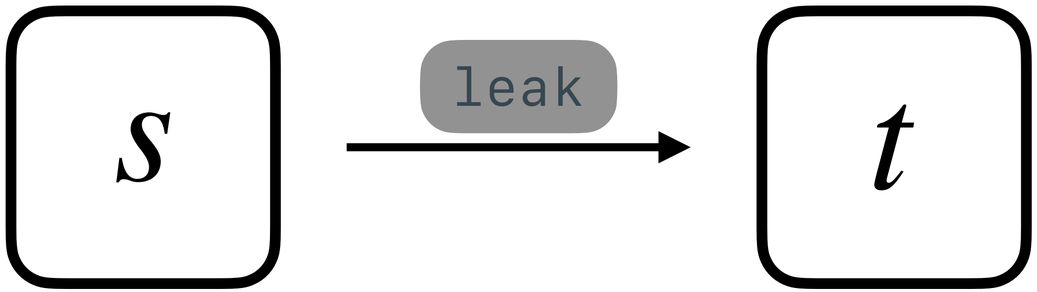


operational semantics



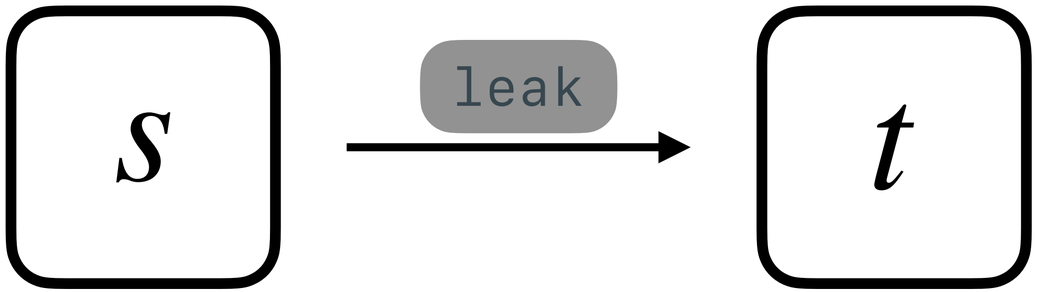
operational semantics + leakage observations

CT-Leakage



`leak` depends on executed instruction

CT-Leakage

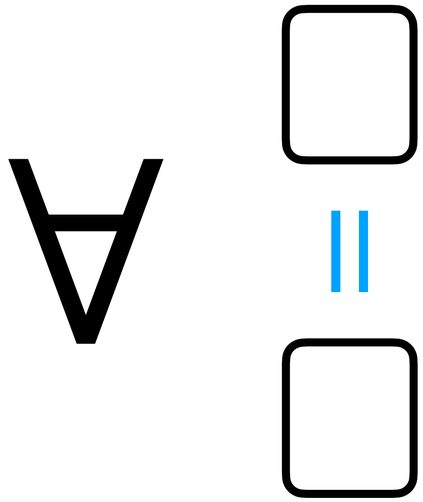


`leak` depends on executed instruction

instruction	`leak`
load / store	address
conditional branch	branch outcome

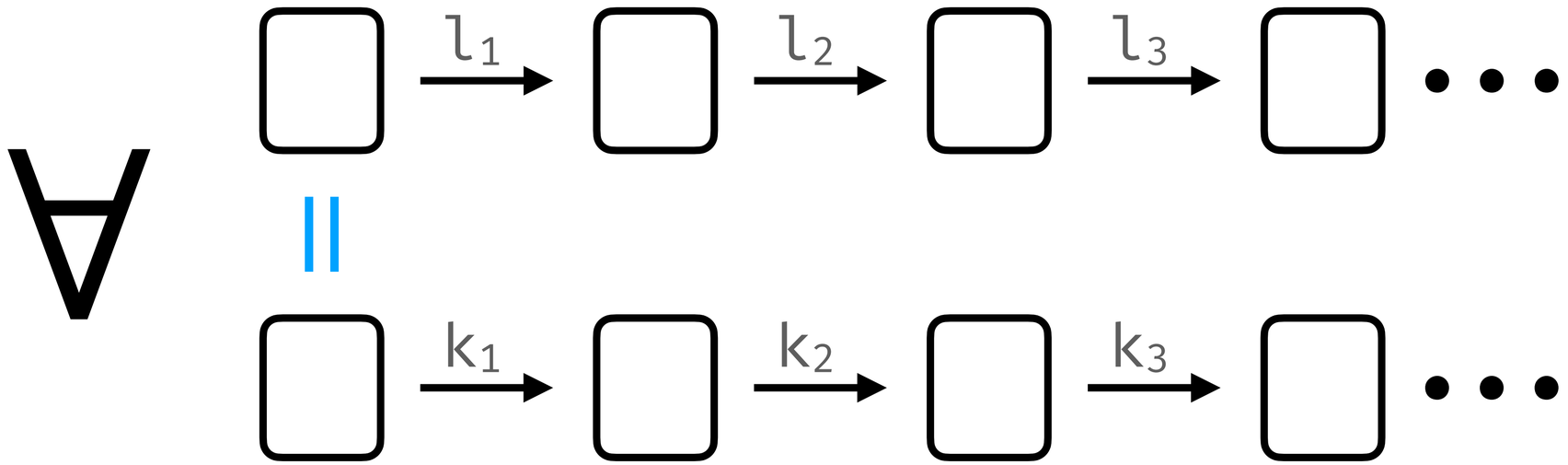
`prog.c` is **CT** **WHEN**

`prog.c` is **CT** **WHEN**



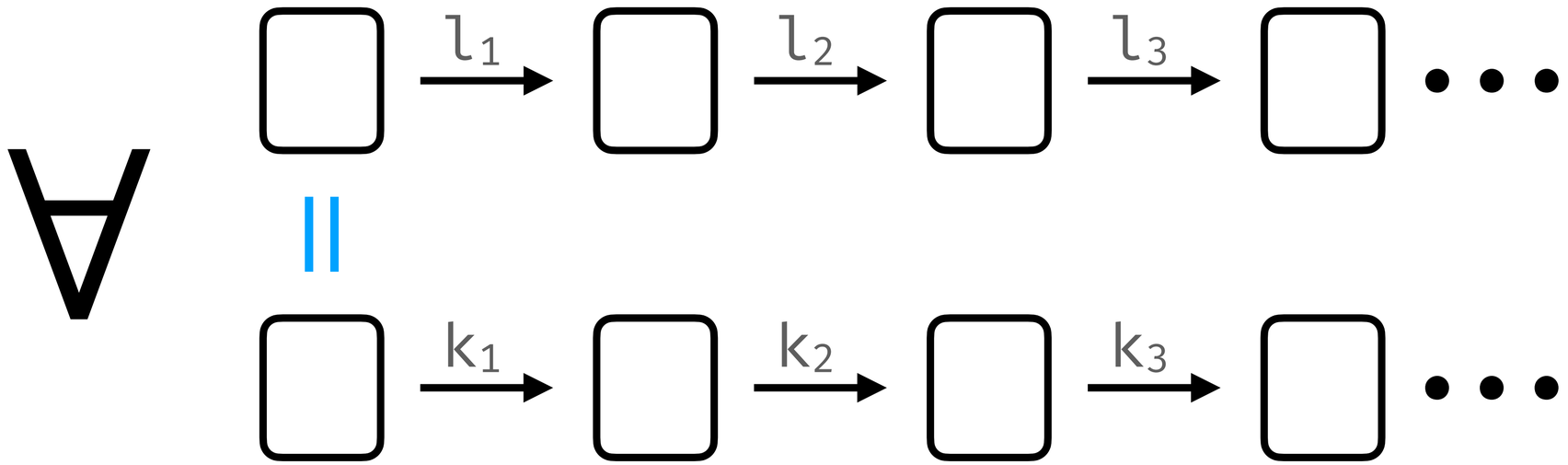
$\overline{=}$ attacker-indistinguishable:
equal except on secrets.

`prog.c` is **CT** **WHEN**



= attacker-indistinguishable:
equal except on secrets.

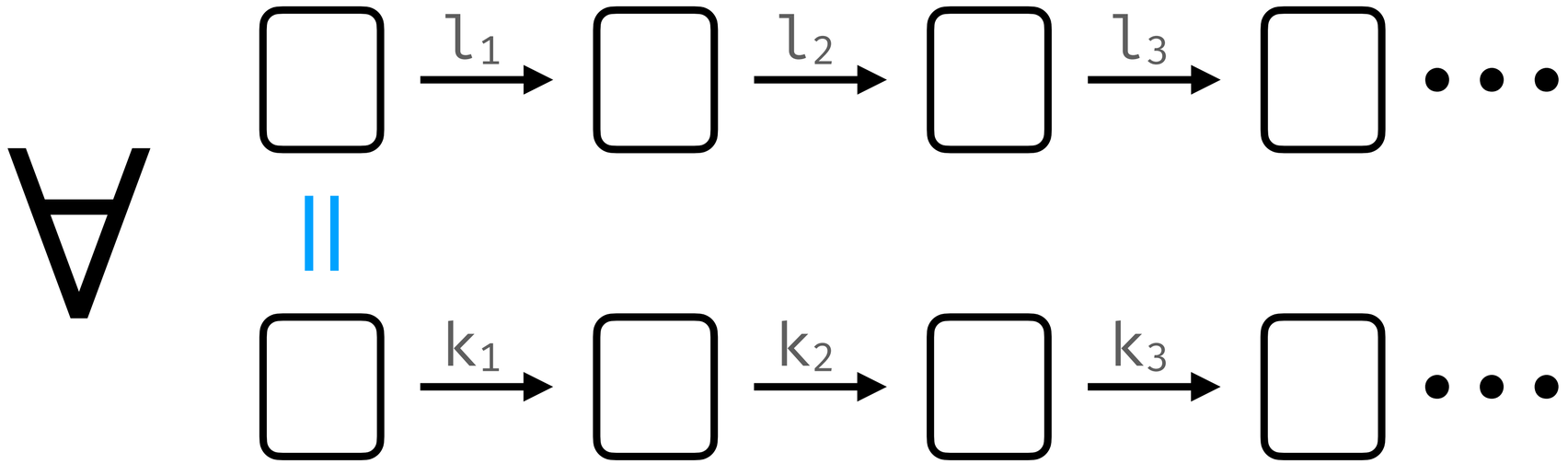
`prog.c` is **CT** **WHEN**



have $k_1k_2k_3\ldots = l_1l_2l_3\ldots$

= attacker-indistinguishable: equal except on secrets.

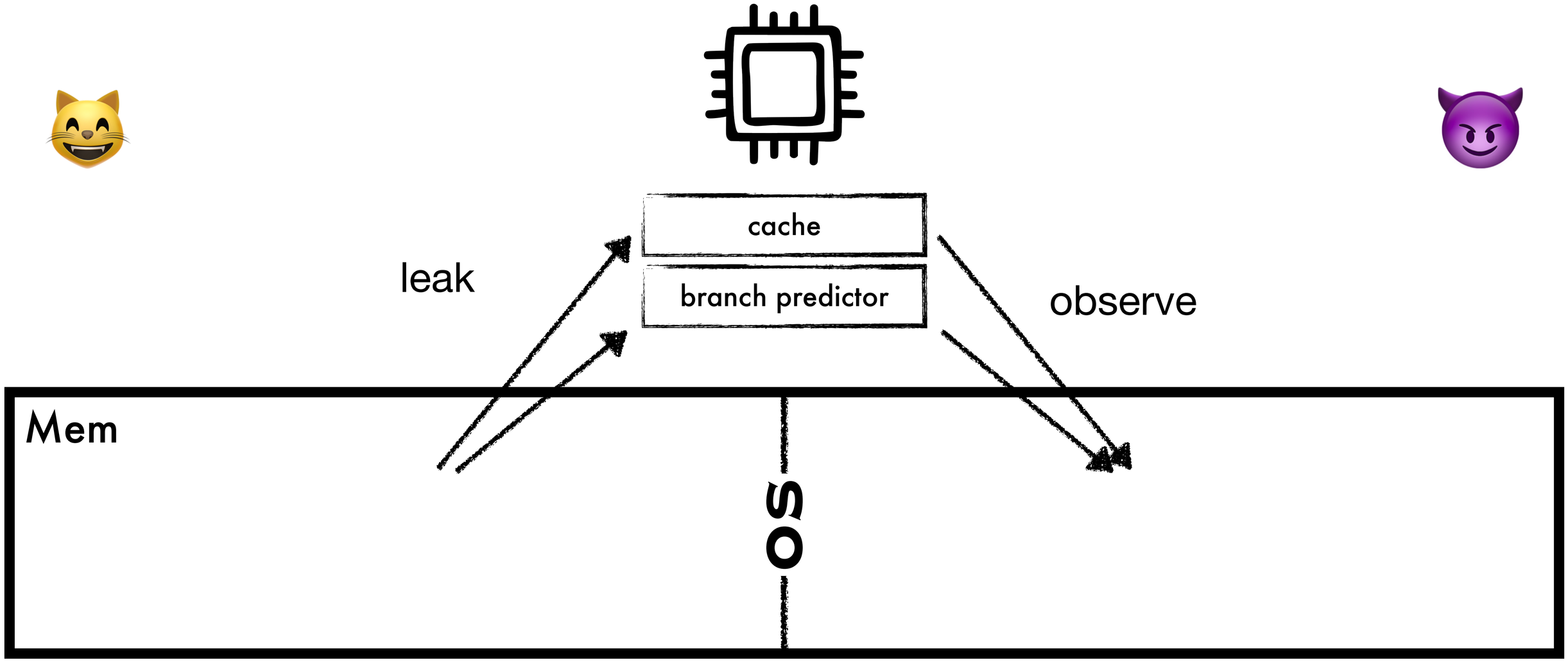
`prog.c` is **CT** **WHEN** “Leakages don’t depend on secrets.”



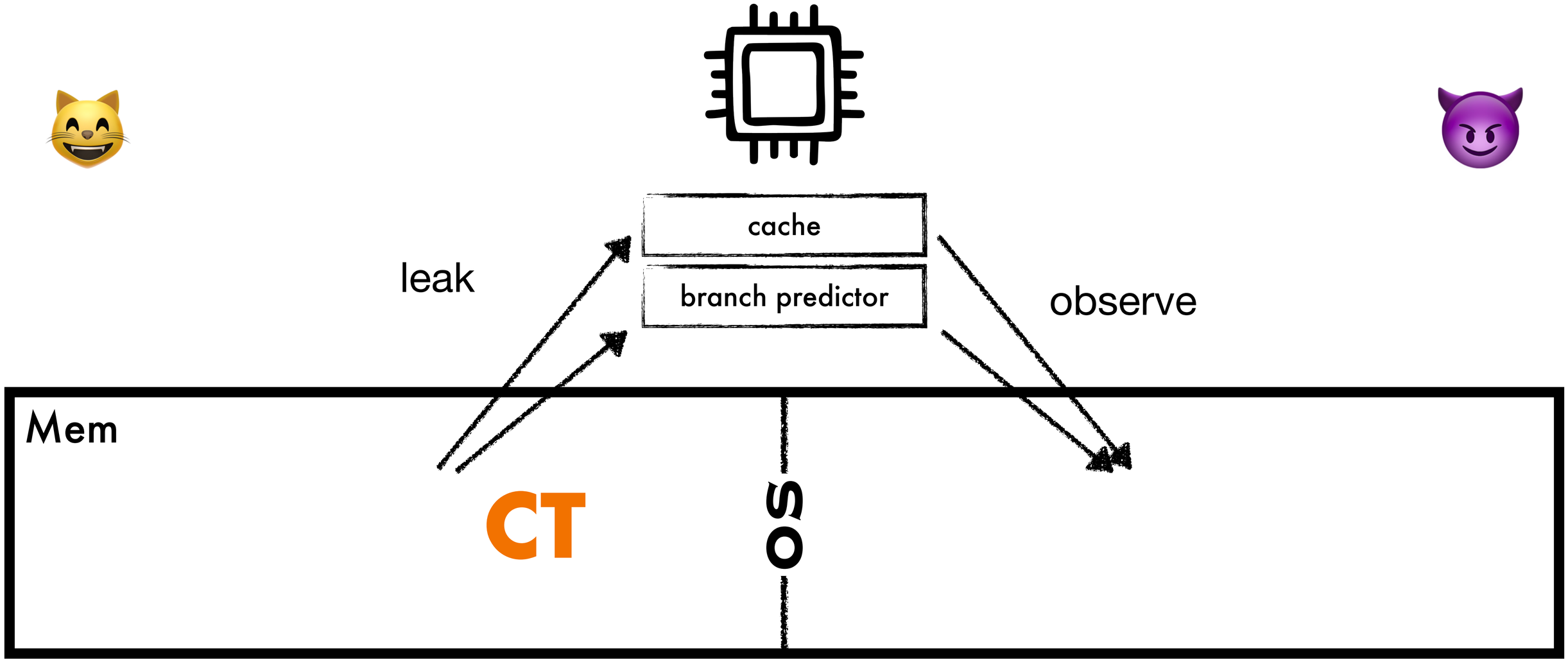
have $k_1k_2k_3\ldots = l_1l_2l_3\ldots$

= attacker-indistinguishable: equal except on secrets.

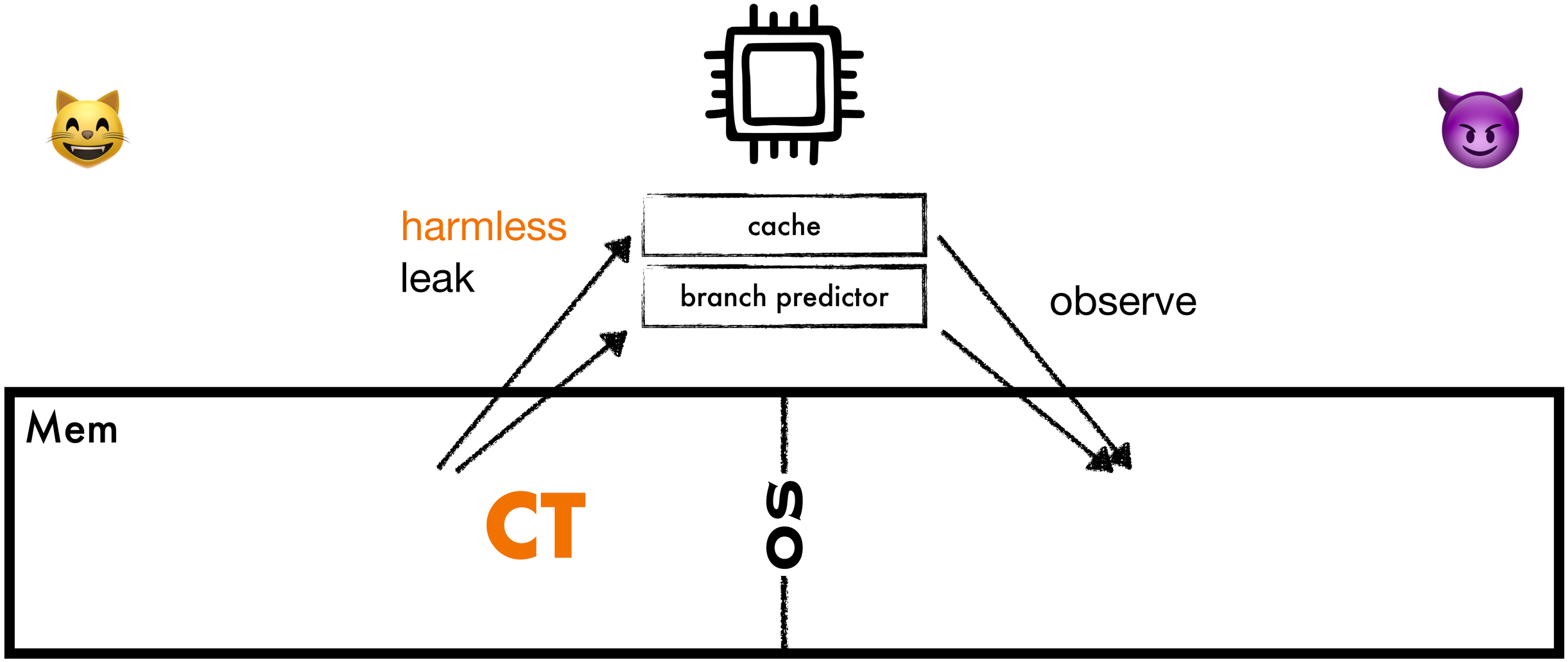
Formal Side Channel Security 101



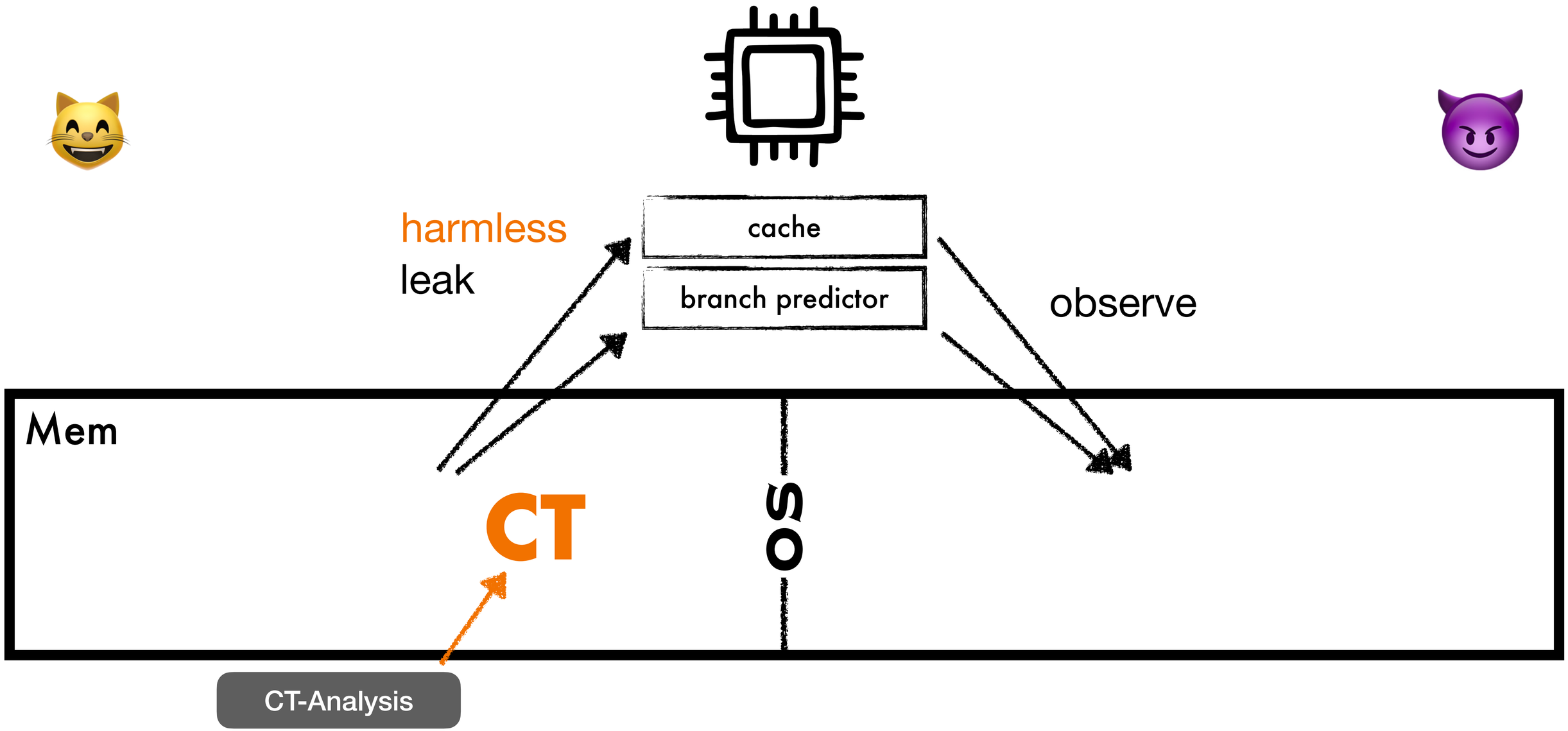
Formal Side Channel Security 101



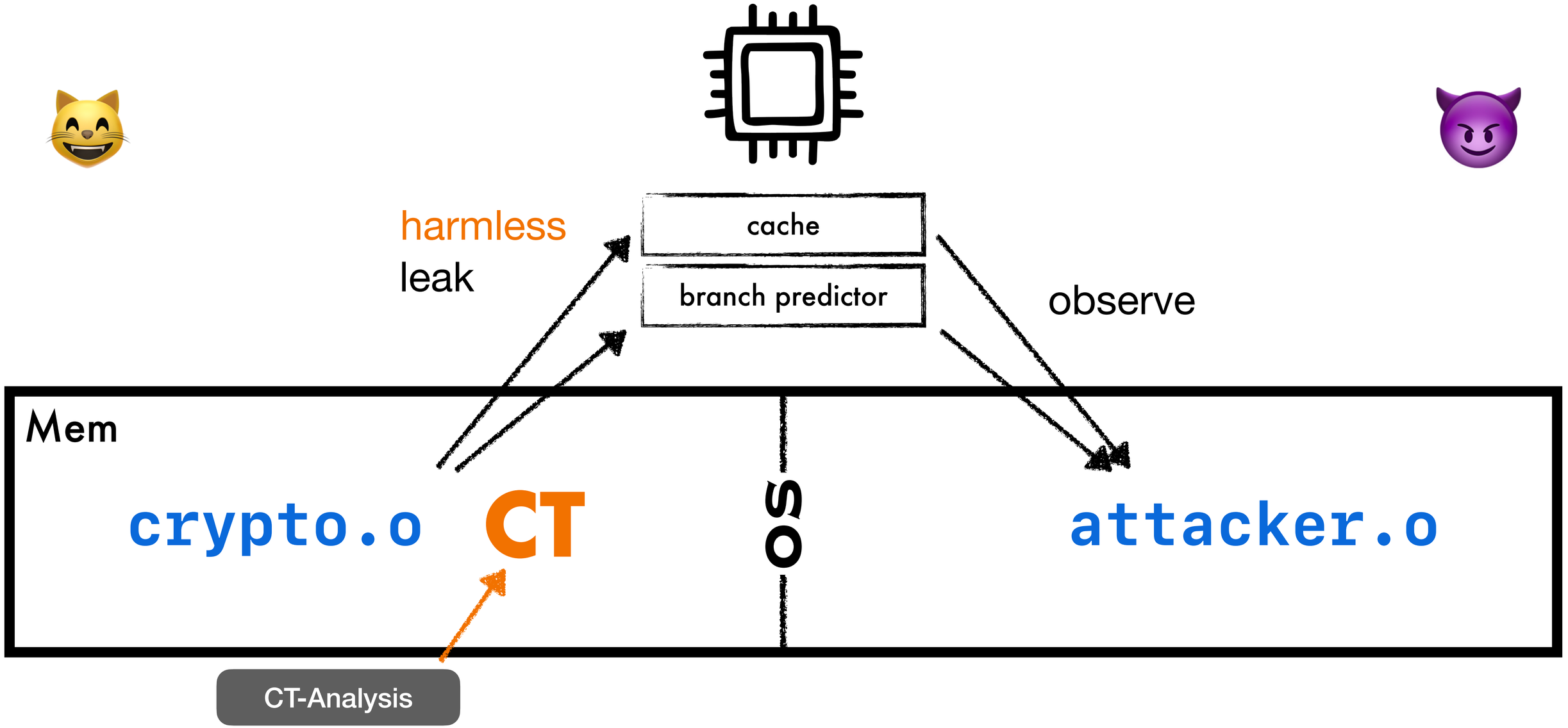
Formal Side Channel Security 101



Formal Side Channel Security 101



Formal Side Channel Security 101

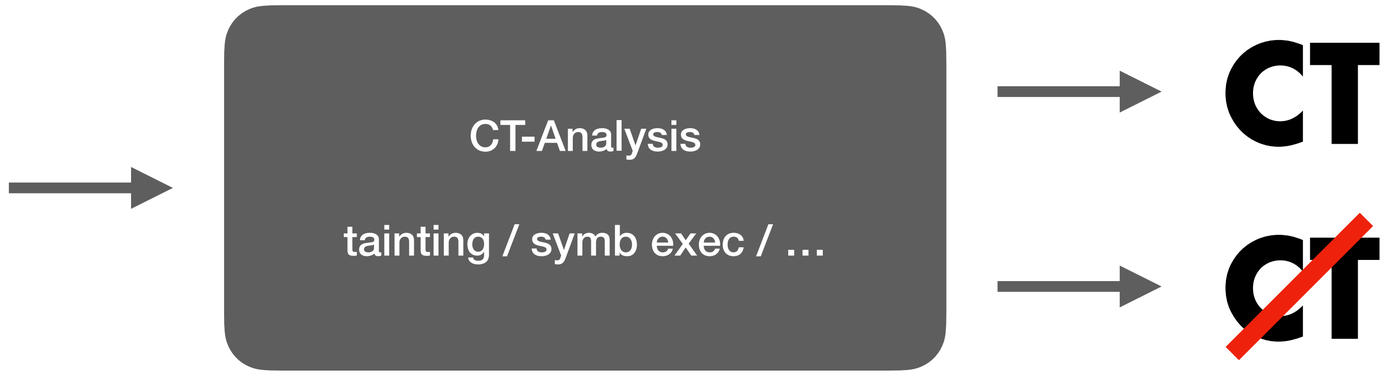


Constant Time Analysis

Remove functions

trusted-crypto-lib.c Raw

```
1  encrypt(secret_data,key);
2  create_hmac(secret_data,key);
3  ...
```



Decompile-then-Analyze

`crypto.o`

CT?

binary level

Decompile-then-Analyze

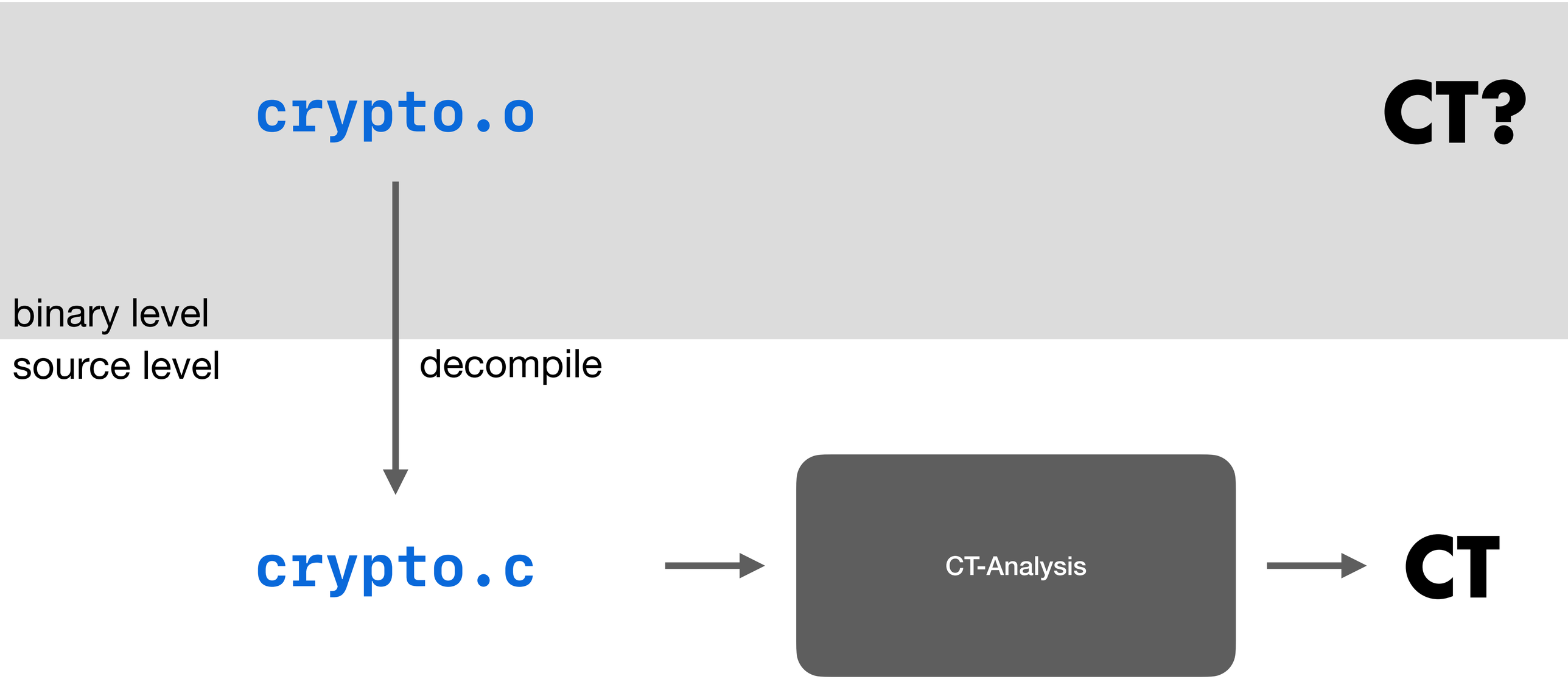
`crypto.o`

CT?

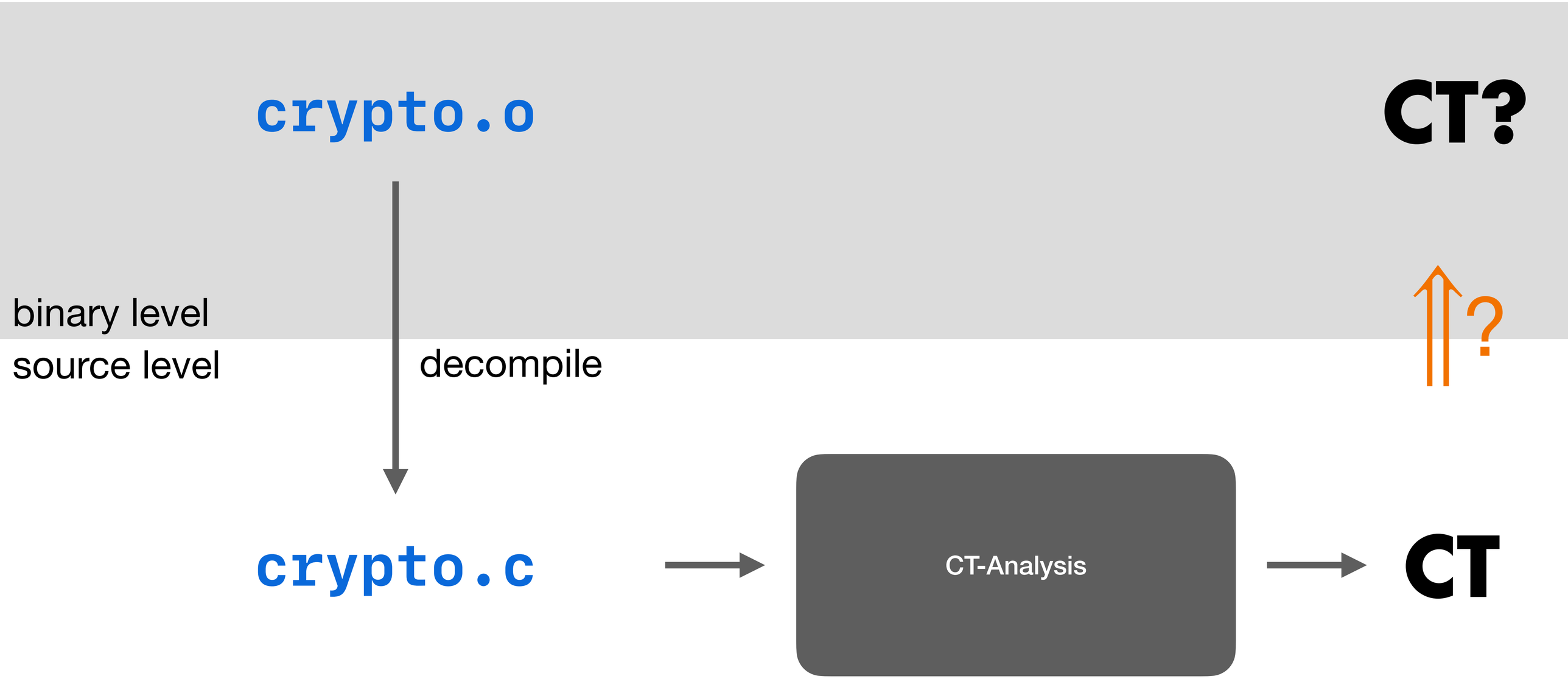
binary level

source level

Decompile-then-Analyze



Decompile-then-Analyze



Decompile-then-Analyze

CLangover (CVE-2024-37880)

ML-KEM.o

```
.LBB0_2: ; for(j = 0; j < 8; j++)
    movzx   r8d, byte ptr [rsi + rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .LBB0_4 ; leaks bit of msg
    mov     edx, 1665
.LBB0_4:
    mov     word ptr [rdi + 2*rcx], dx
    inc     rcx
    cmp     rcx, 8
    jne     .LBB0_2
```

Decompile-then-Analyze

CLangover (CVE-2024-37880)

ML-KEM.o

```
.LBB0_2: ; for(j = 0; j < 8; j++)
    movzx   r8d, byte ptr [rsi + rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .LBB0_4 ; leaks bit of msg
    mov     edx, 1665
.LBB0_4:
    mov     word ptr [rdi + 2*rcx], dx
    inc     rcx
    cmp     rcx, 8
    jne     .LBB0_2
```

Decompile-then-Analyze

CLangover (CVE-2024-37880)

ML-KEM.o —RetDec→ **ML-KEM.c**

```
.LBB0_2: ; for(j = 0; j < 8; j++)
    movzx   r8d, byte ptr [rsi + rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .LBB0_4 ; leaks bit of msg
    mov     edx, 1665
.LBB0_4:
    mov     word ptr [rdi + 2*rcx], dx
    inc     rcx
    cmp     rcx, 8
    jne     .LBB0_2
```

Decompile-then-Analyze

CLangover (CVE-2024-37880)

ML-KEM.o → RetDec → **ML-KEM.c**

```
.LBB0_2:  ; for(j = 0; j < 8; j++)
    movzx   r8d, byte ptr [rsi + rax]
    xor     edx, edx
    bt      r8d, ecx
    jae     .LBB0_4 ; leaks bit of msg
    mov     edx, 1665
.LBB0_4:
    mov     word ptr [rdi + 2*rcx], dx
    inc     rcx
    cmp     rcx, 8
    jne     .LBB0_2
```

```
for (int64_t j = 0; j < 8; j++) {
  byte = *(char *)(msg_offset + msg);

  coef = (1 << (int32_t)j % 32 & (int32_t)byte) == 0 ?
             0 : 1665;        // no leak

  *(int16_t *)(2 * j + poly_offset) = coef;

}
```

Decompile-then-Analyze

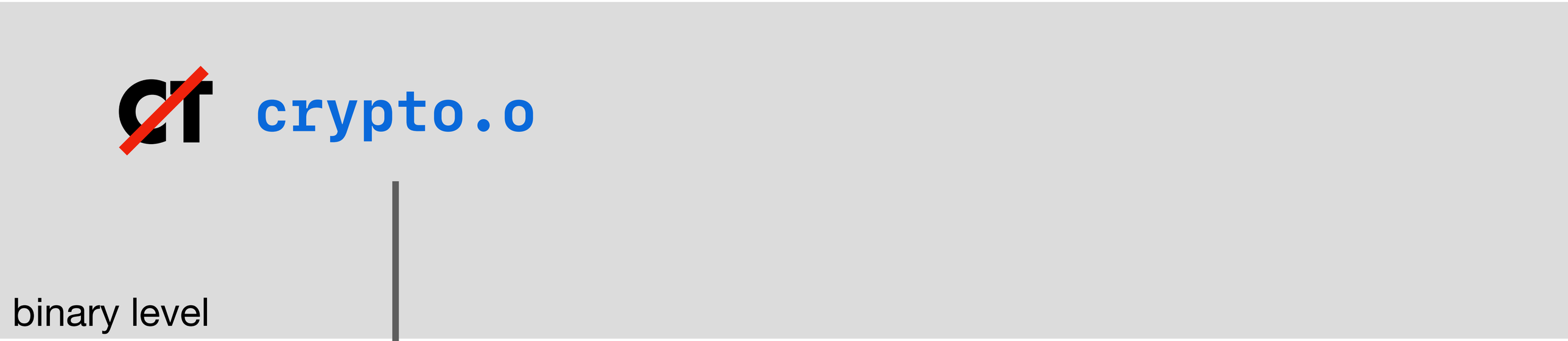
`crypto.o`

binary level

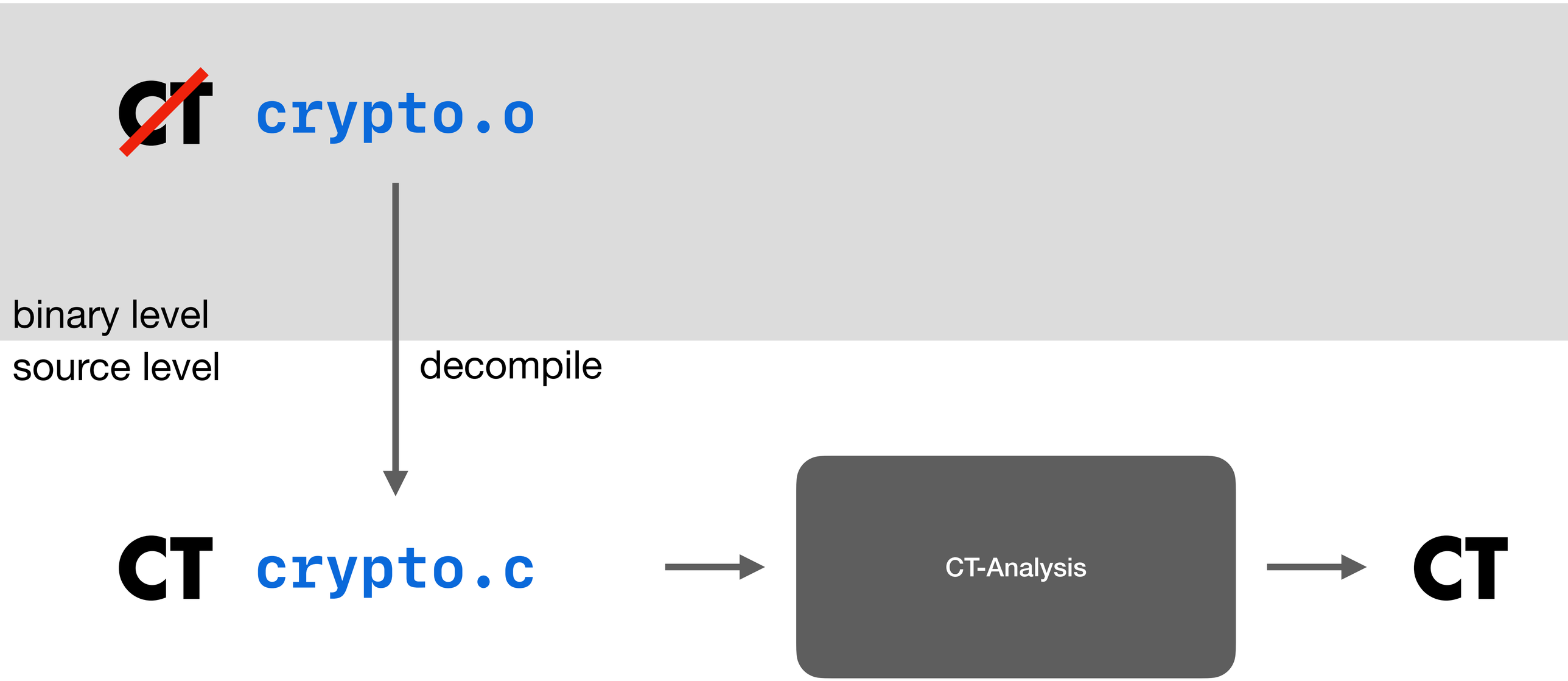
source level

CT-Analysis

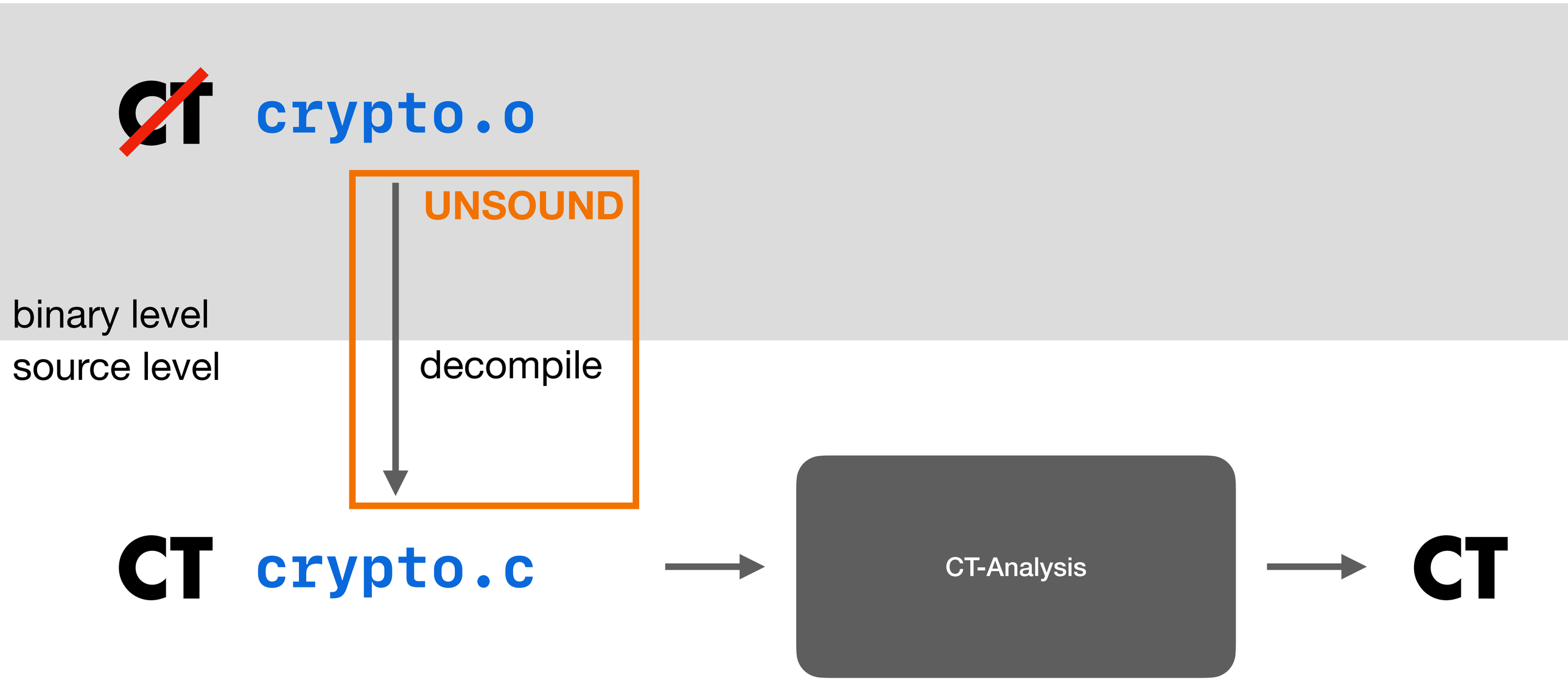
Decompile-then-Analyze



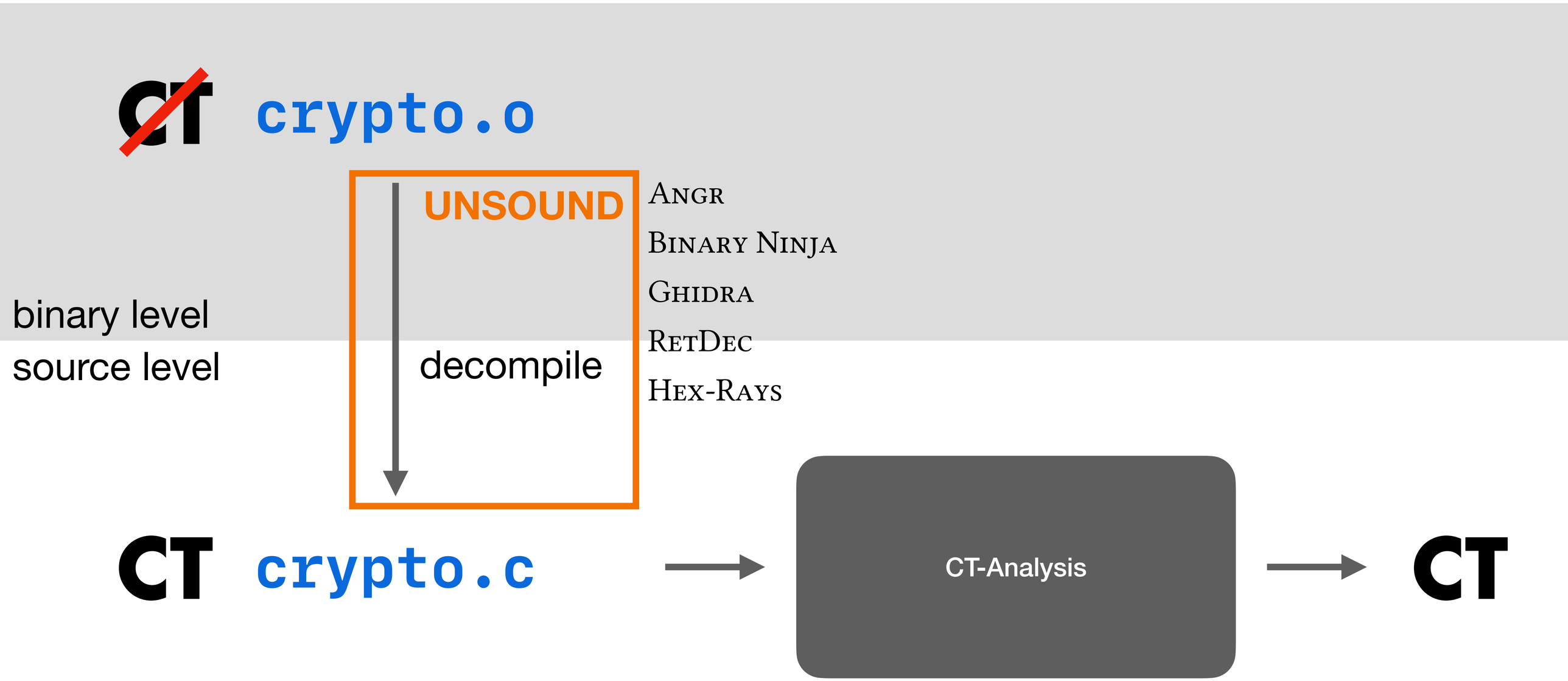
Decompile-then-Analyze



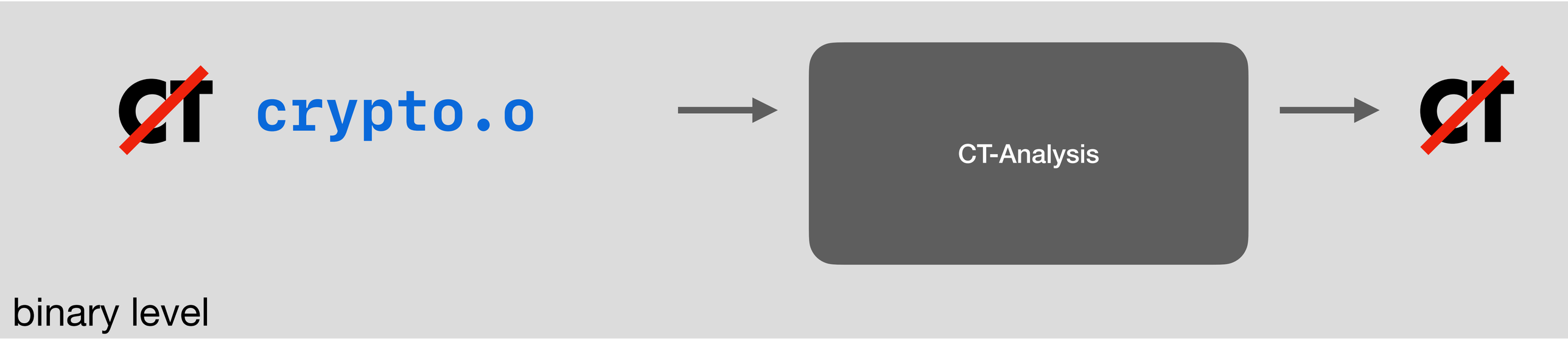
Decompile-then-Analyze



Decompile-then-Analyze



Decompile-then-Analyze



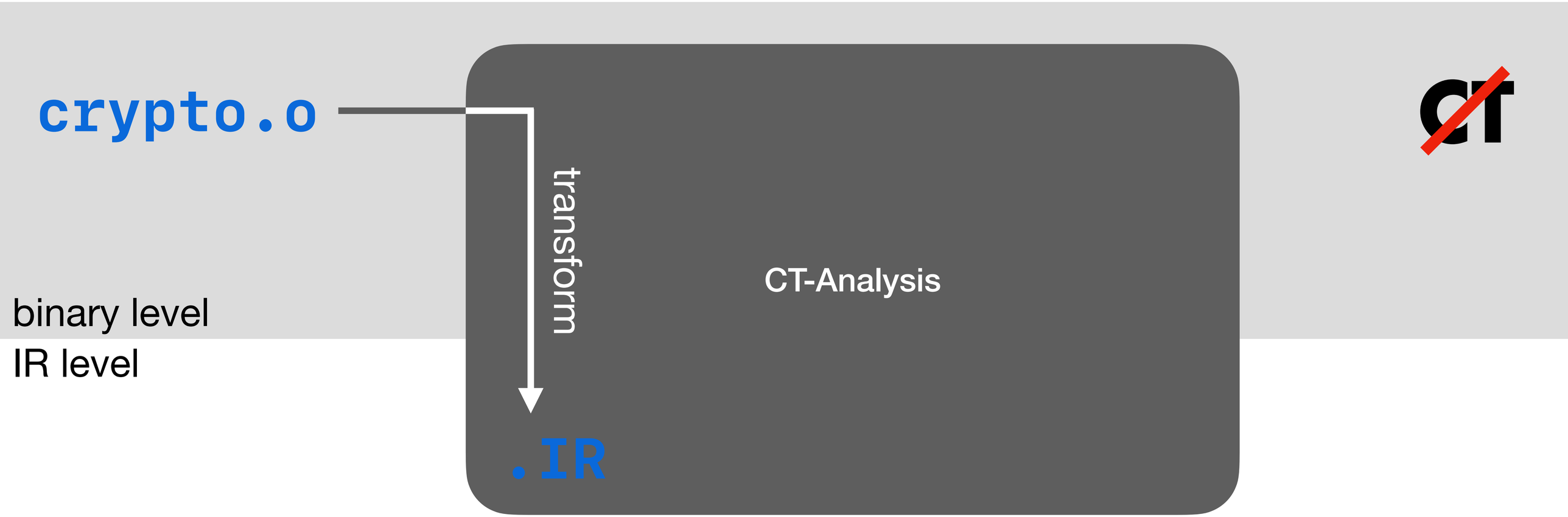
binary level

Decompile-then-Analyze

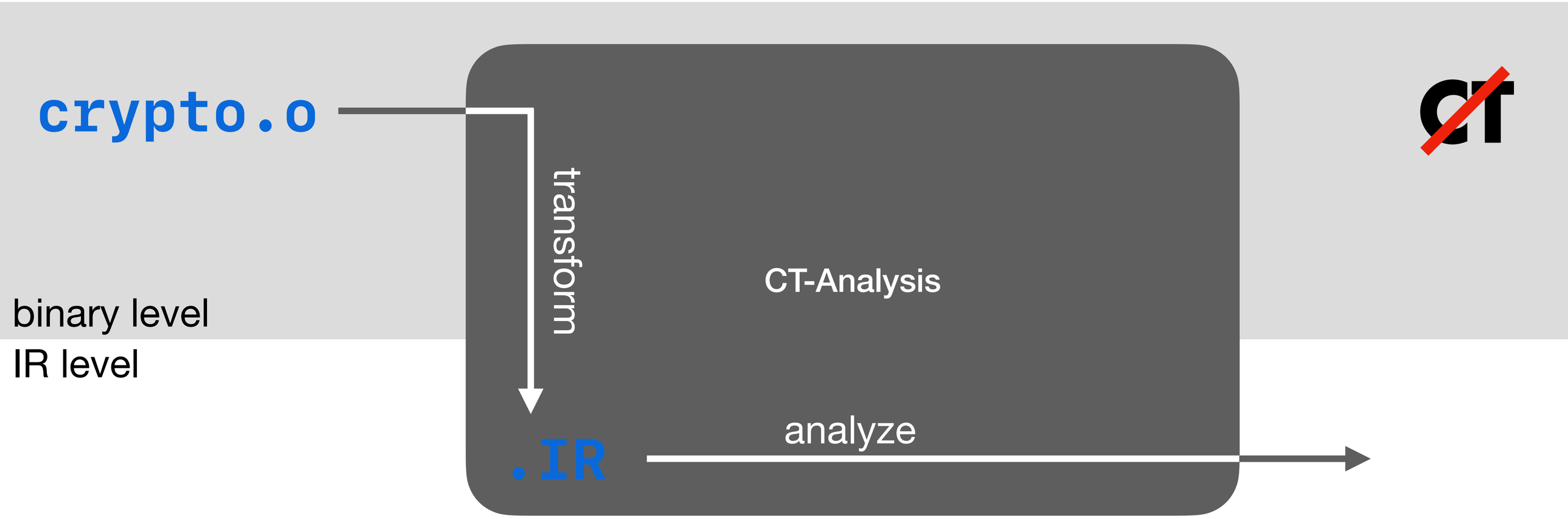
`crypto.o`

binary level

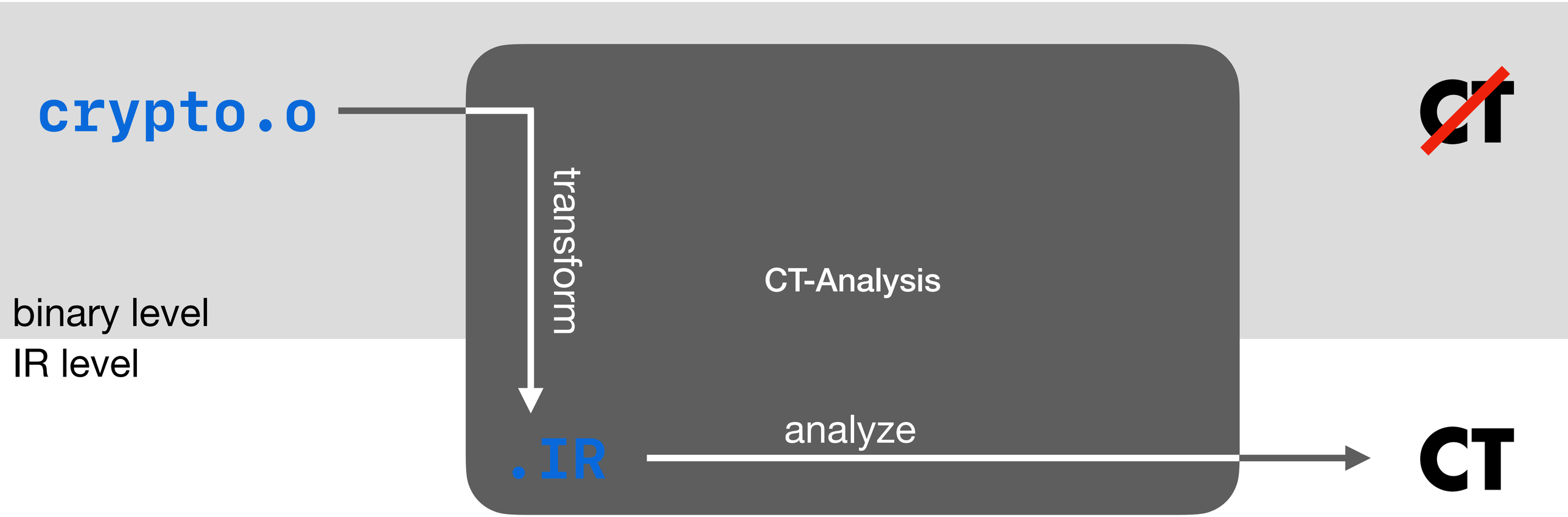
Decompile-then-Analyze



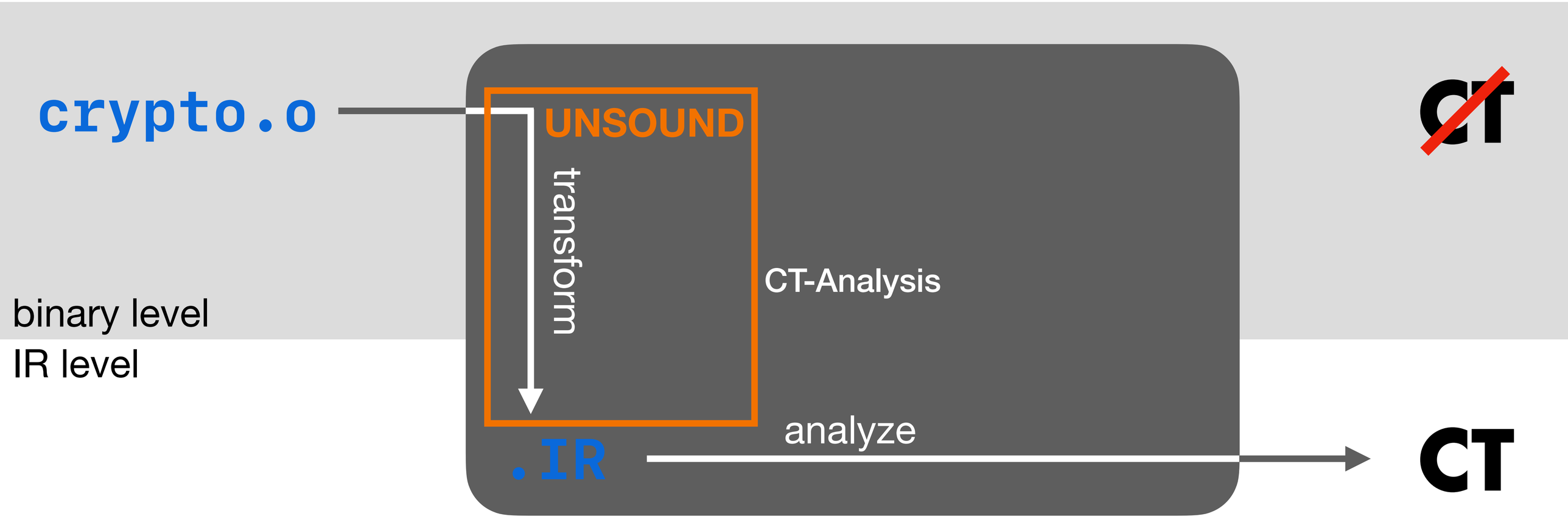
Decompile-then-Analyze



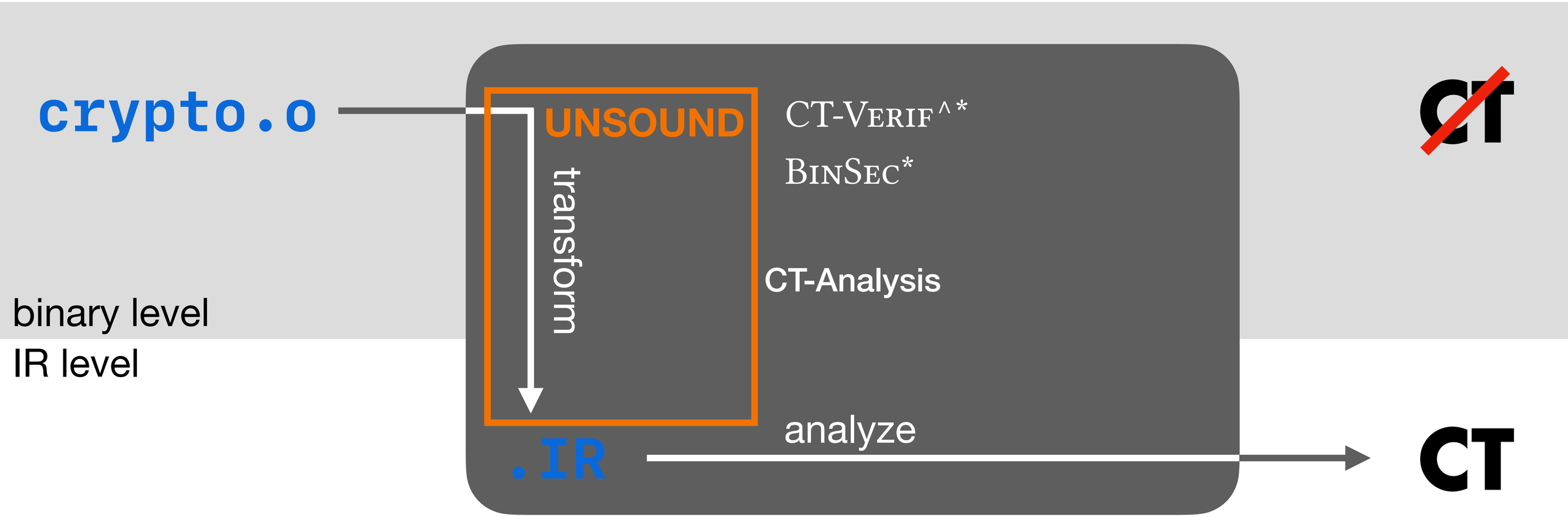
Decompile-then-Analyze



Decompile-then-Analyze



Decompile-then-Analyze



^: input is LLVM-IR
*: on handcrafted `example.o`

Properties for Sound Decompilation

Properties for Sound Decompilation

Properties for Sound Decompilation

[ ] reflects **CT**

[ ] preserves **CT**

[ ] **CT** transparent

Properties for Sound Decompilation

[ ] reflects **CT**

WHEN

prog ~~CT~~ $\Longrightarrow$ [prog] ~~CT~~

“[ ] **removes no vulnerabilities”**

[ ] preserves **CT**

[ ] **CT** transparent

Properties for Sound Decompilation

[ ] reflects **CT** **WHEN** **prog** ~~**CT**~~ $\Longrightarrow$ **[prog]** ~~**CT**~~

“[ ] removes no vulnerabilities”

[ ] preserves **CT** **WHEN** **prog** **CT** $\Longrightarrow$ **[prog]** **CT**

“[ ] introduces no vulnerabilities”

[ ] **CT** transparent

Properties for Sound Decompilation

[ ] reflects **CT** **WHEN** `prog` ~~**CT**~~ $\Longrightarrow$ `[prog]` ~~**CT**~~

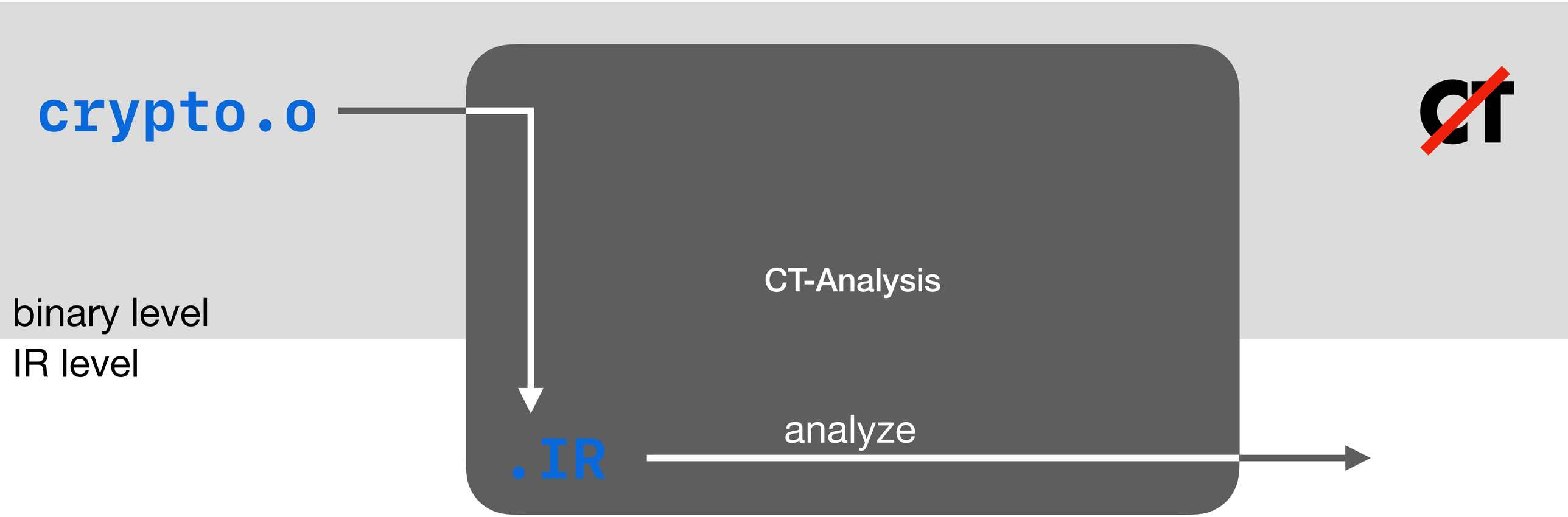
"[ ] removes no vulnerabilities"

[ ] preserves **CT** **WHEN** `prog` **CT** $\Longrightarrow$ `[prog]` **CT**

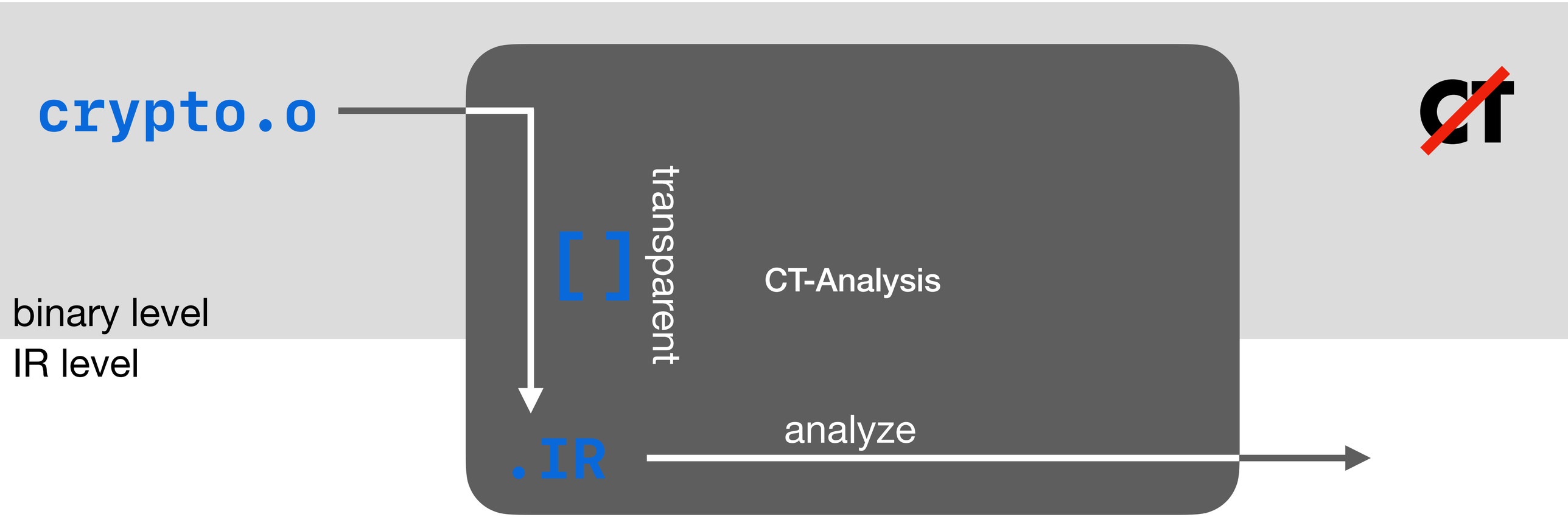
"[ ] introduces no vulnerabilities"

[ ] **CT** transparent **WHEN** `prog` **CT** $\Longleftrightarrow$ `[prog]` **CT**

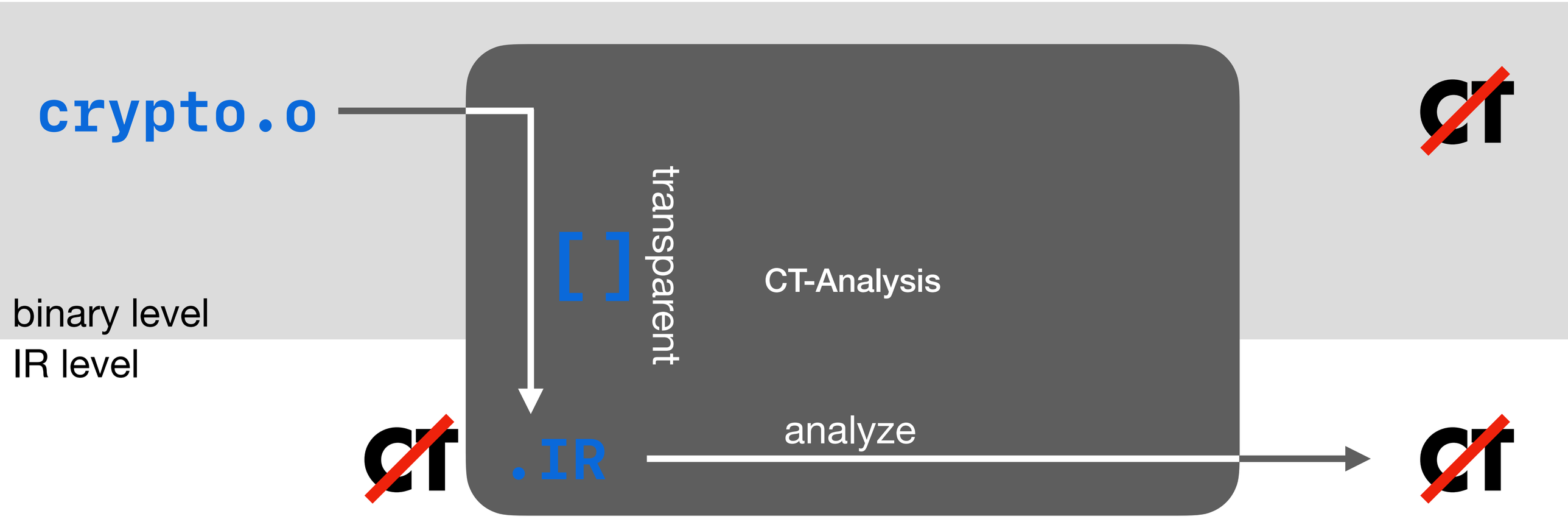
Decompile-then-Analyze



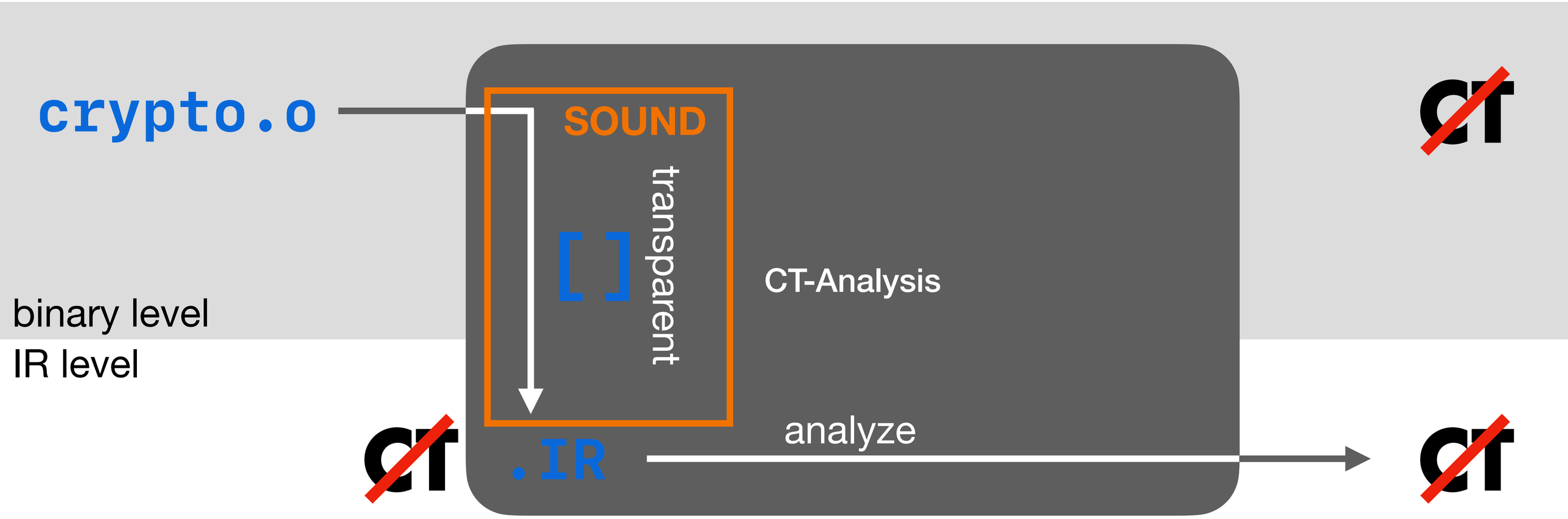
Decompile-then-Analyze



Decompile-then-Analyze



Decompile-then-Analyze



Proving “[ ] is **CT** transparent”

Proving “[ ] is **CT** transparent”

start with [ ] preserves **CT** and [ ] reflects **CT**

`prog` CT $\Longrightarrow$ `[prog]` CT

Proving "[ ] preserves CT"

Proving "`[]` preserves CT"

`prog` **CT** $\Longrightarrow$ `[prog]` **CT**

`prog`

`[prog]`

Cites!

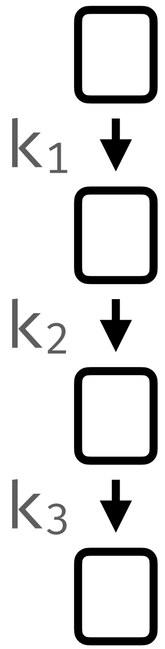
Proving "[] preserves CT"

`prog` **CT** $\Longrightarrow$ `[prog]` **CT**

`prog`

`[prog]`

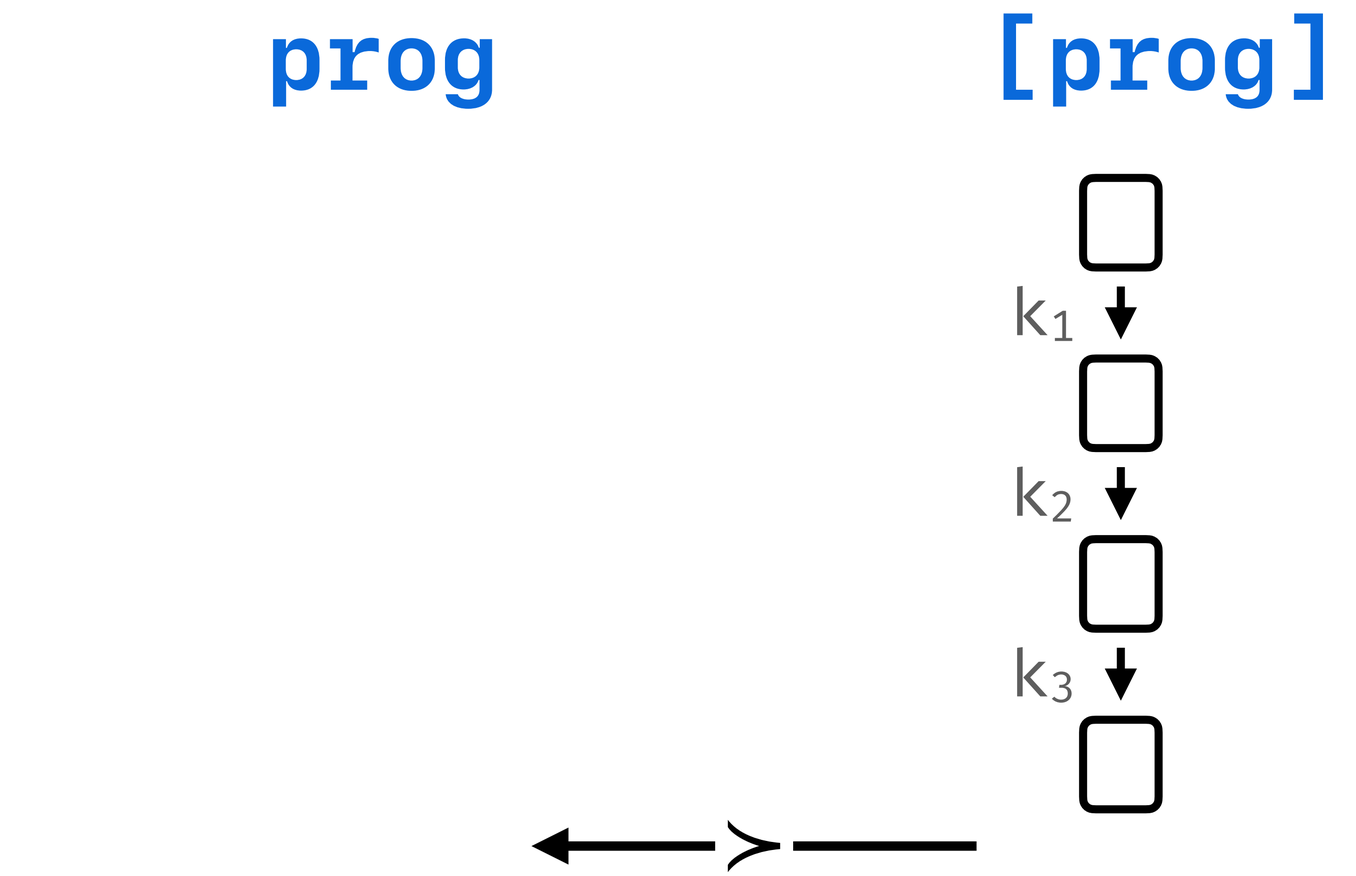
Cites!



Proving "[] preserves CT"

`prog` **CT** $\Longrightarrow$ `[prog]` **CT**

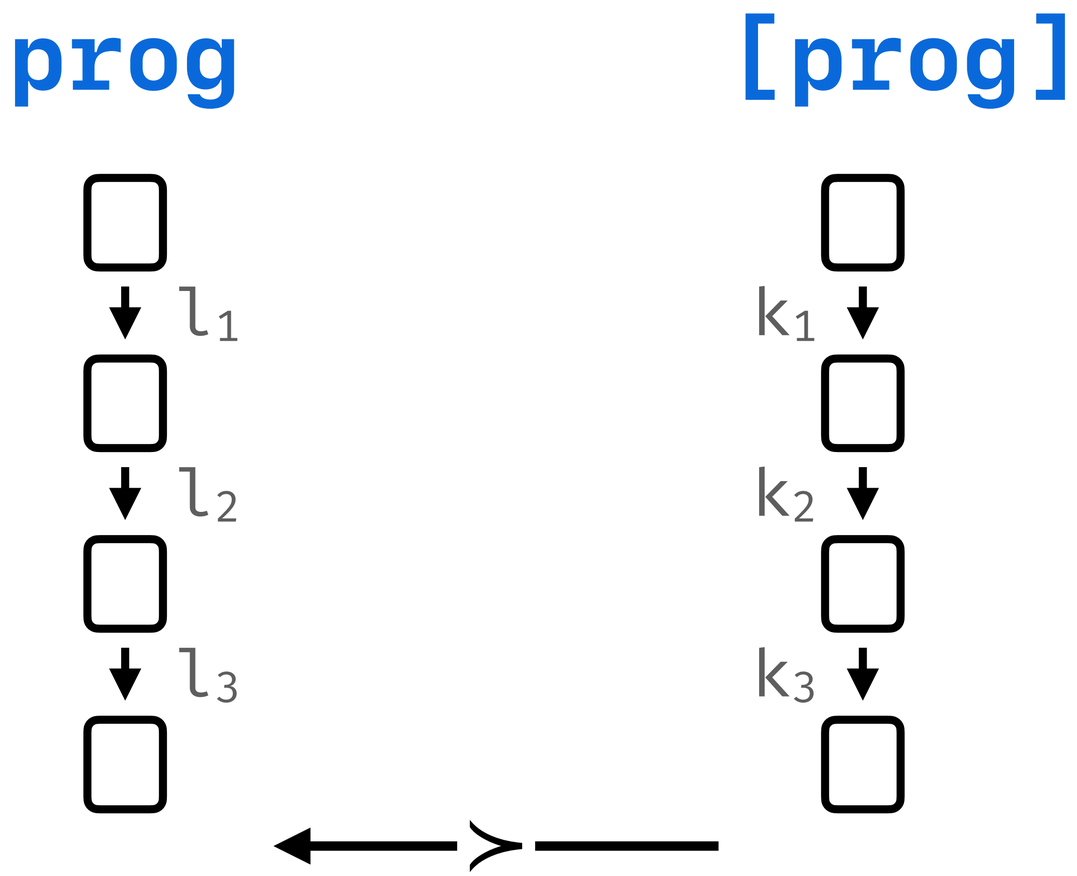
Cites!



Proving "`[]` preserves CT"

`prog` CT $\Longrightarrow$ `[prog]` CT

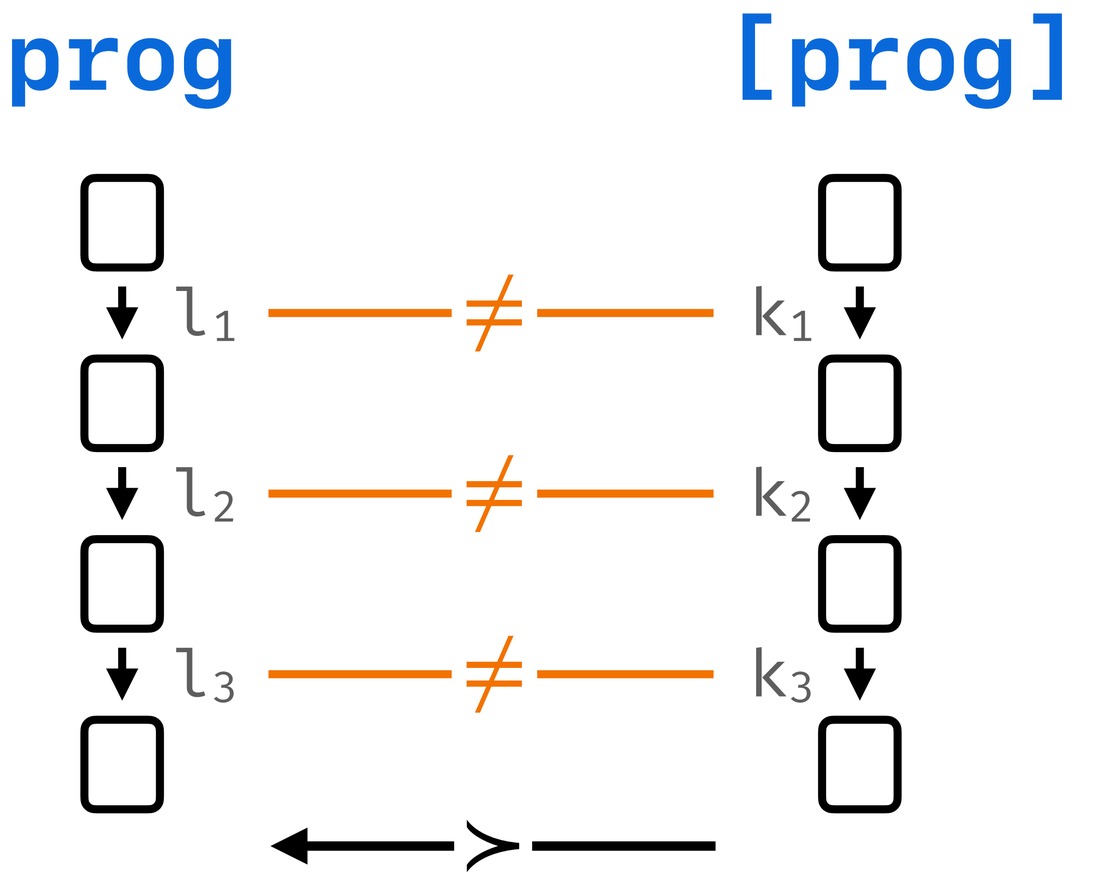
Cites!



Proving "[] preserves CT"

`prog` **CT** $\Longrightarrow$ `[prog]` **CT**

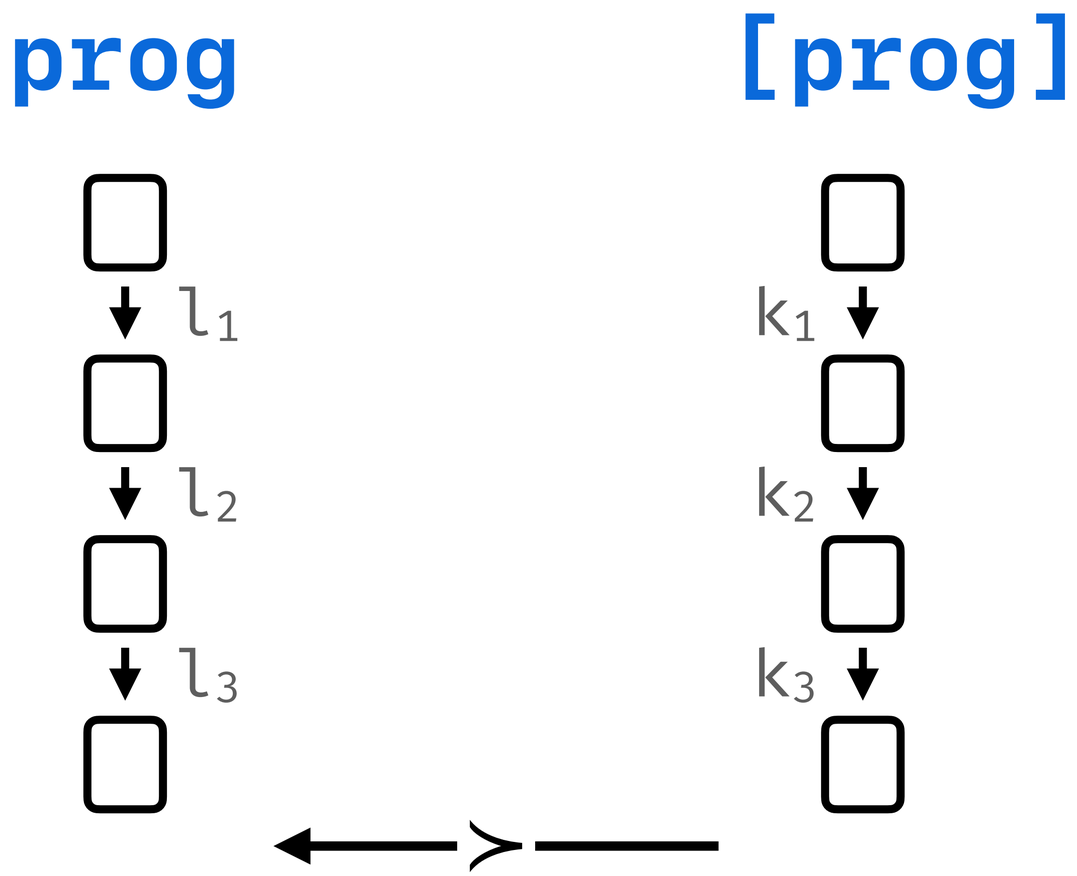
Cites!



Proving "`[]` preserves CT"

`prog` CT $\Longrightarrow$ `[prog]` CT

Cites!

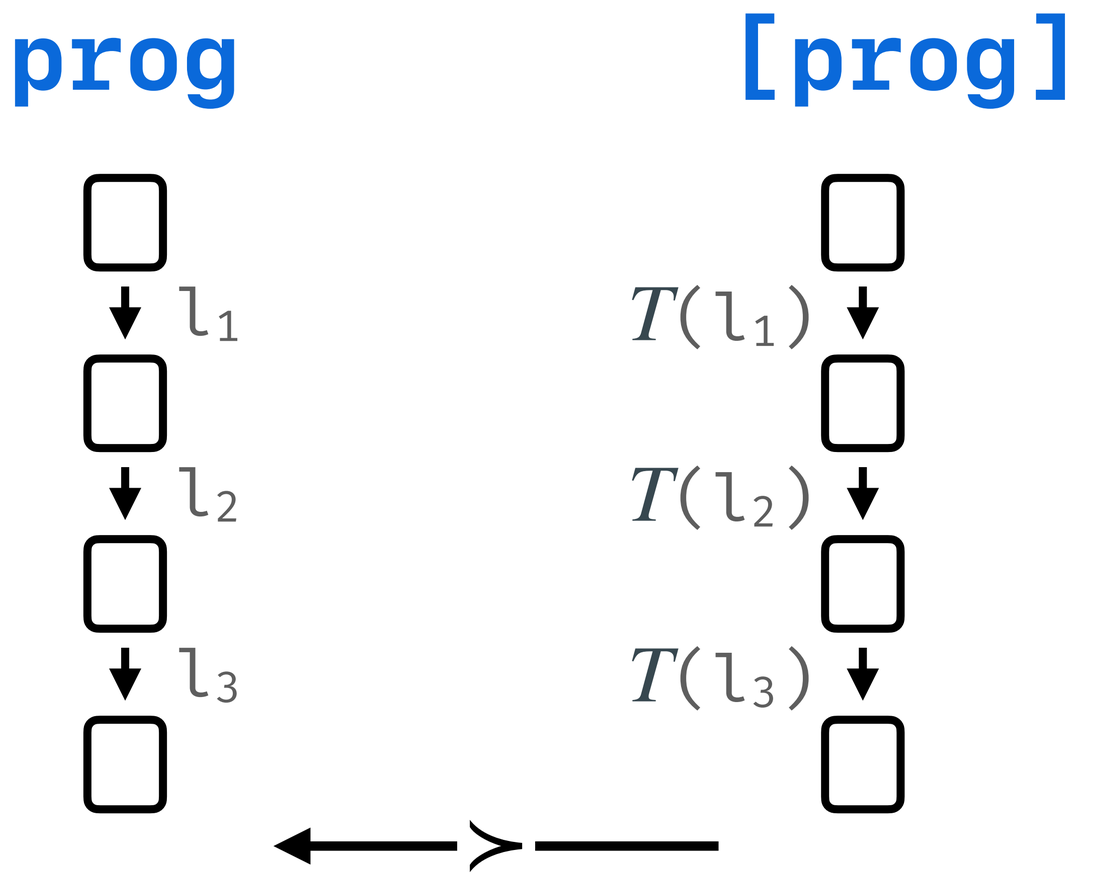


Proving "[] preserves CT"

prog CT $\Longrightarrow$ [prog] CT

T : leakage transformer

Cites!

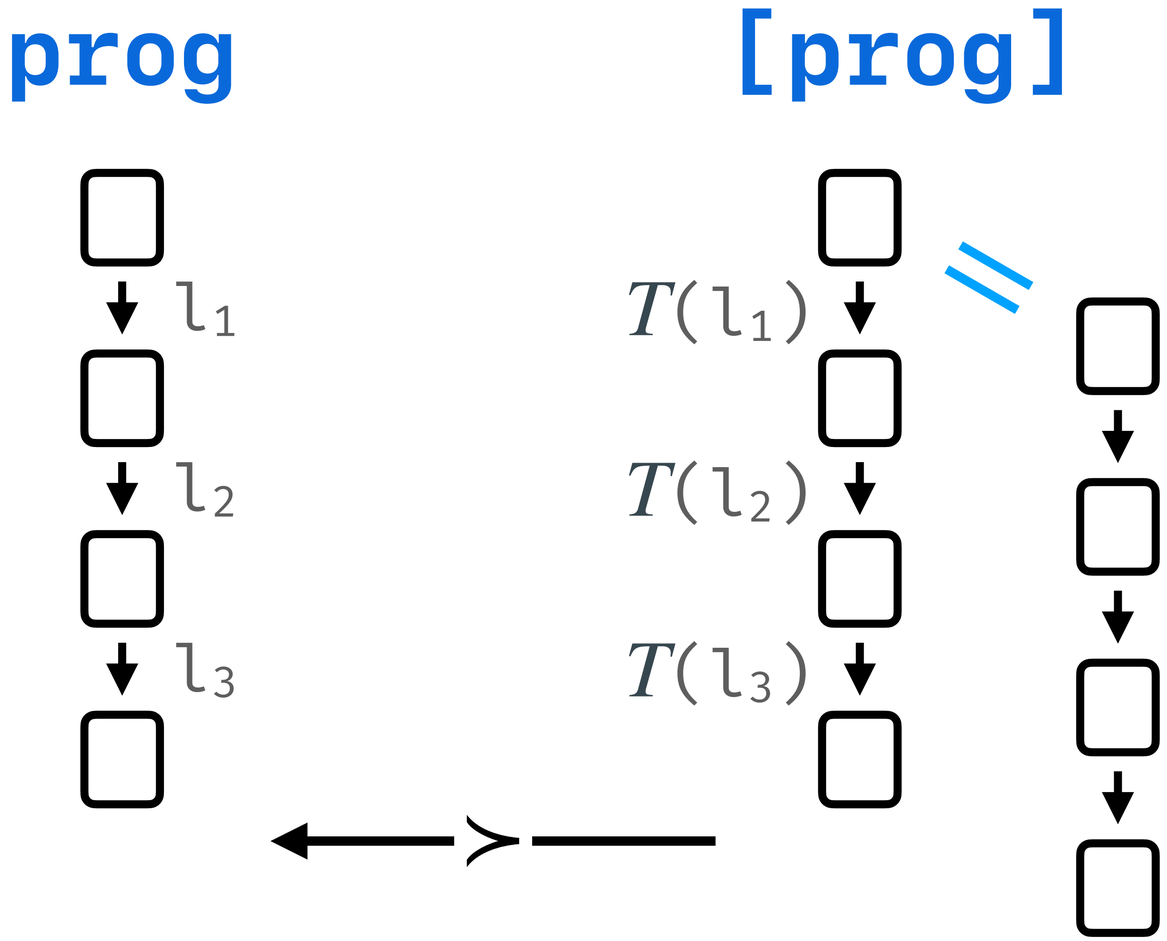


Proving "[] preserves CT"

prog CT $\Longrightarrow$ **[prog] CT**

T : leakage transformer

Cites!

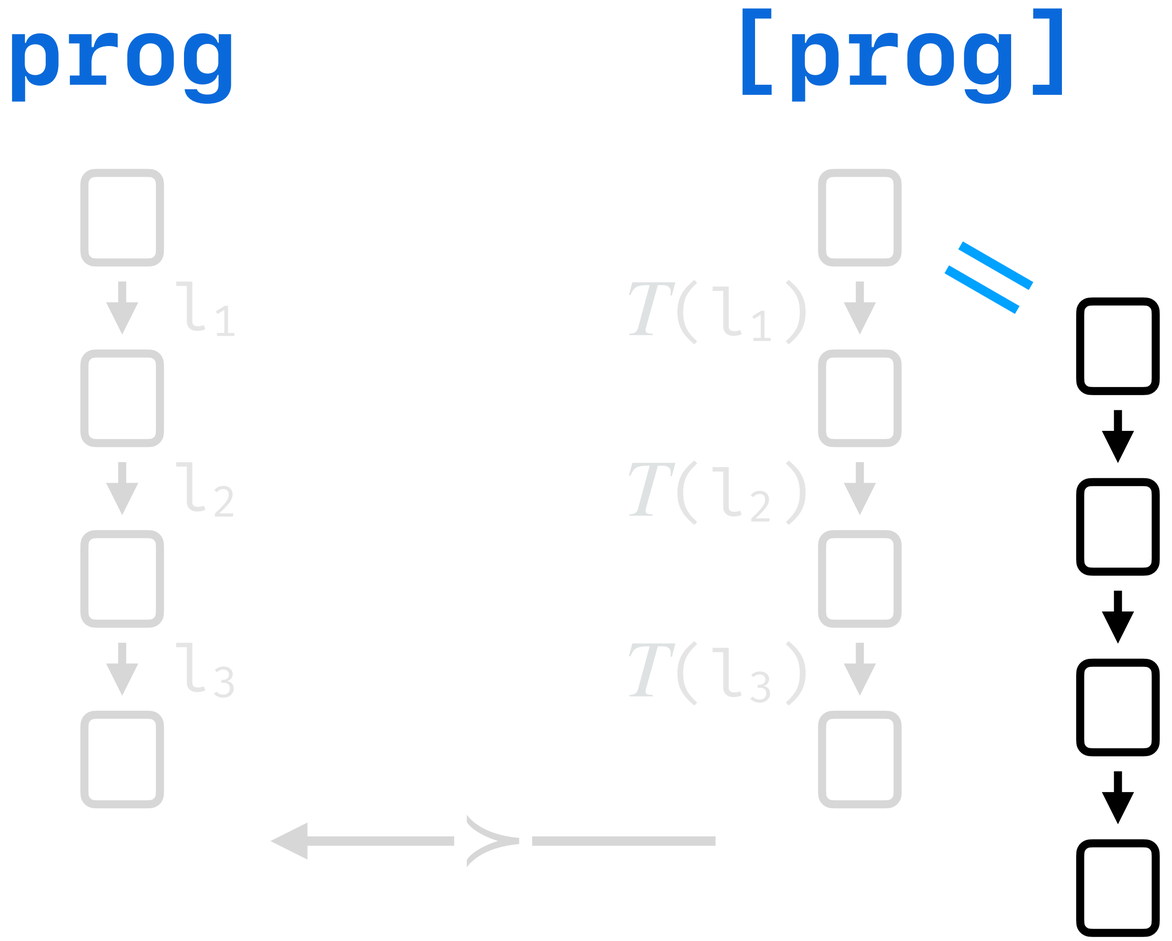


Proving "[] preserves CT"

prog CT $\Longrightarrow$ **[prog] CT**

T : leakage transformer

Cites!

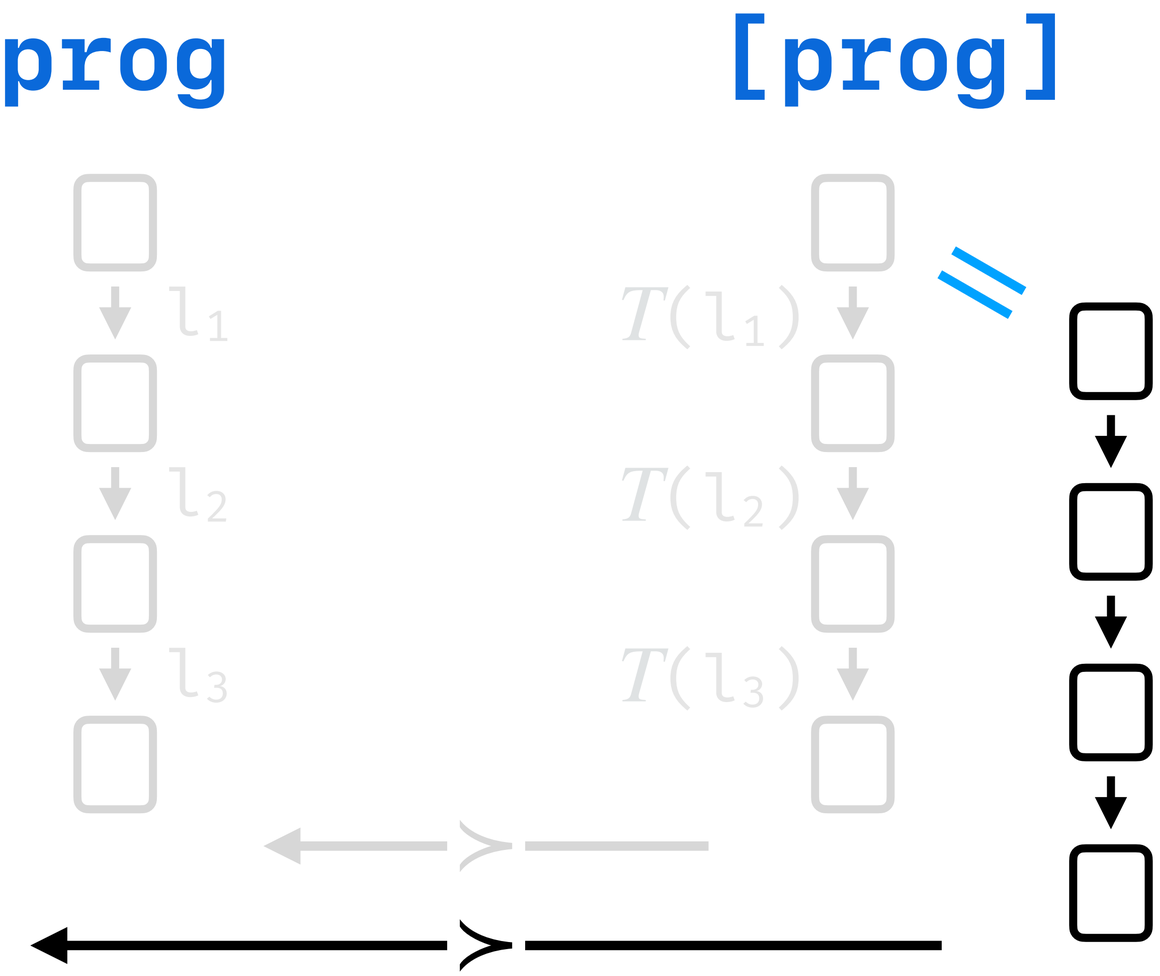


Proving "[] preserves CT"

prog CT $\Longrightarrow$ [prog] CT

T : leakage transformer

Cites!

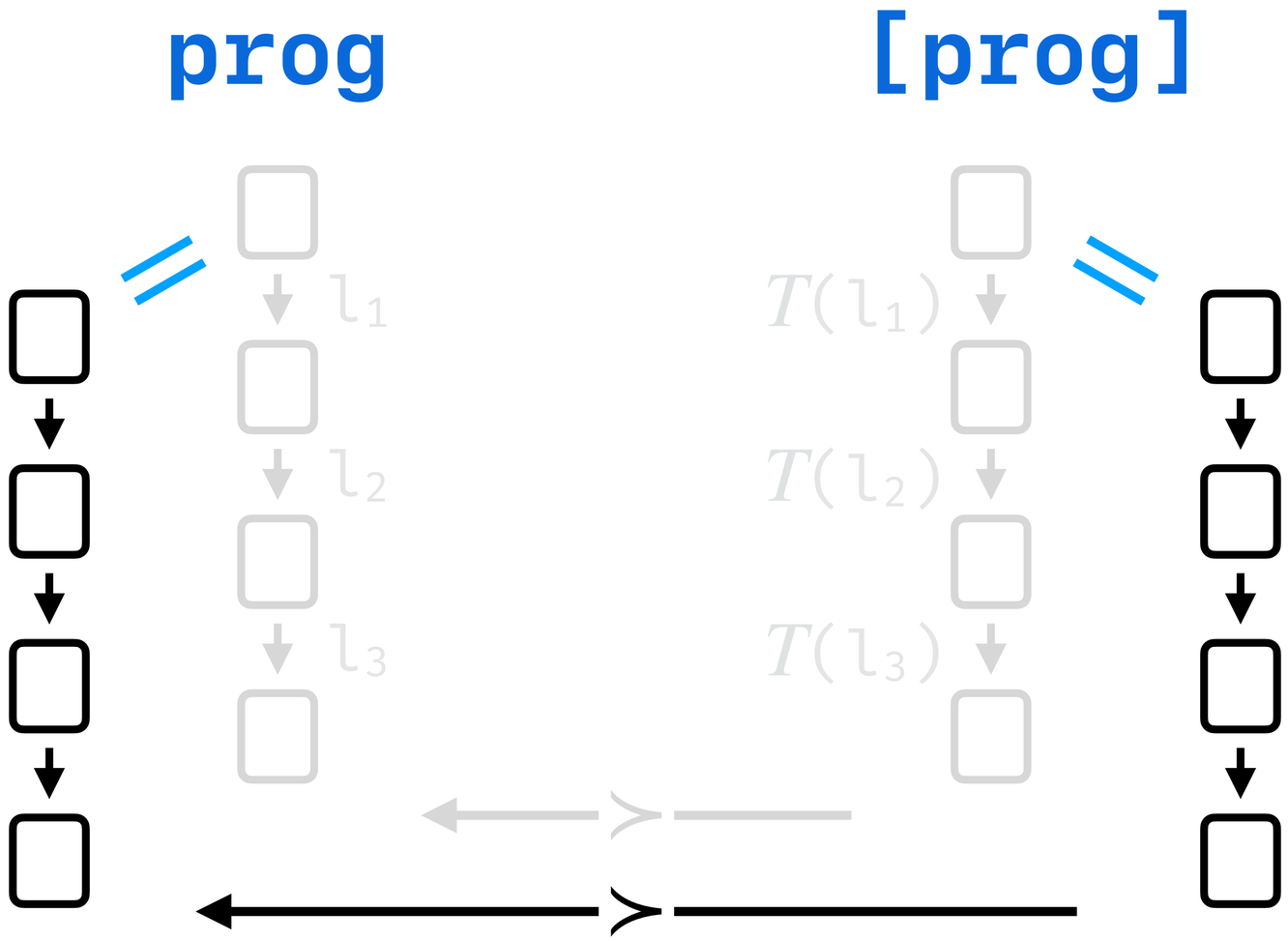


Proving "[] preserves CT"

prog CT $\Longrightarrow$ **[prog] CT**

T : leakage transformer

Cites!

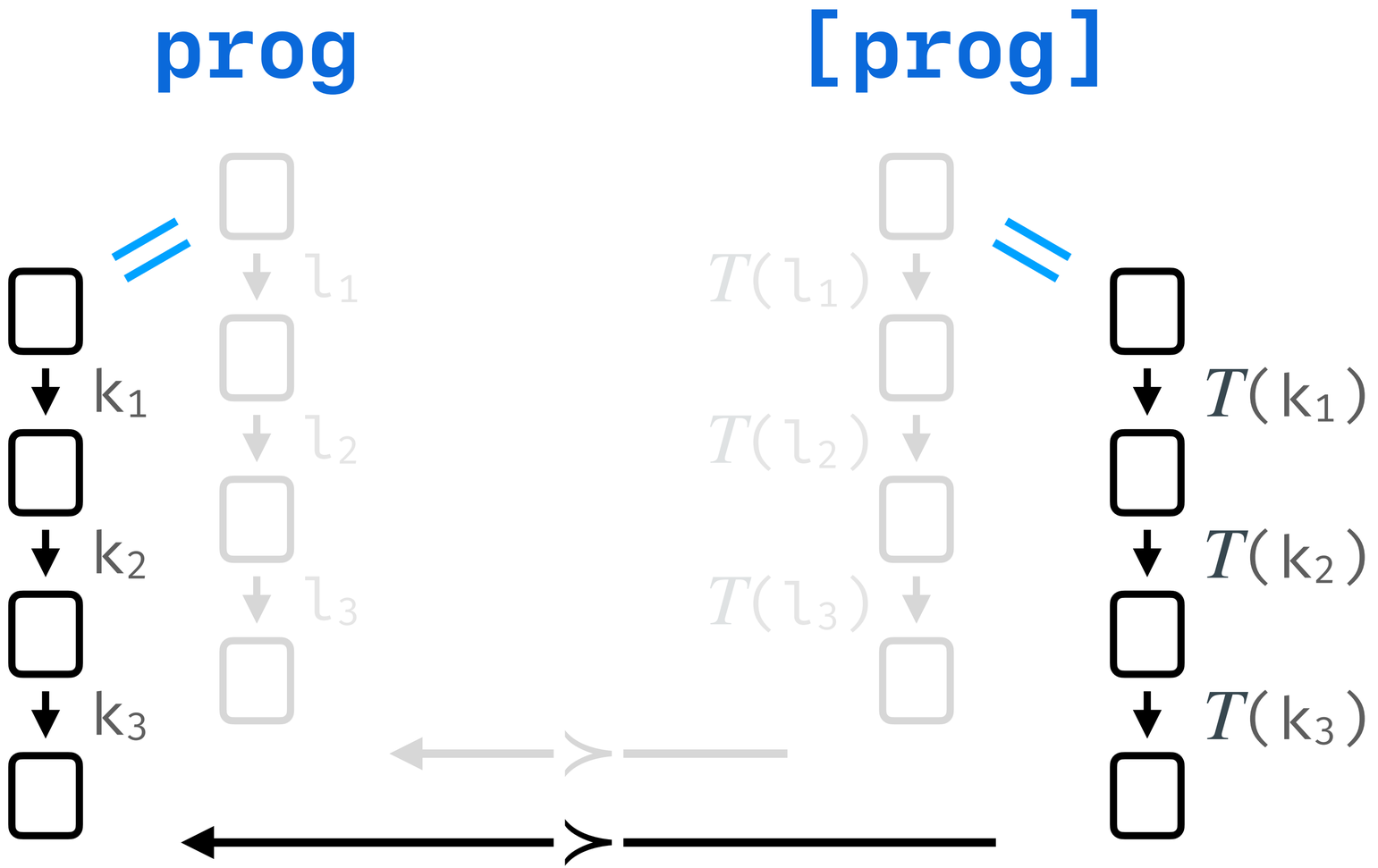


Proving "`[]` preserves CT"

`prog` CT $\Longrightarrow$ `[prog]` CT

T : leakage transformer

Cites!

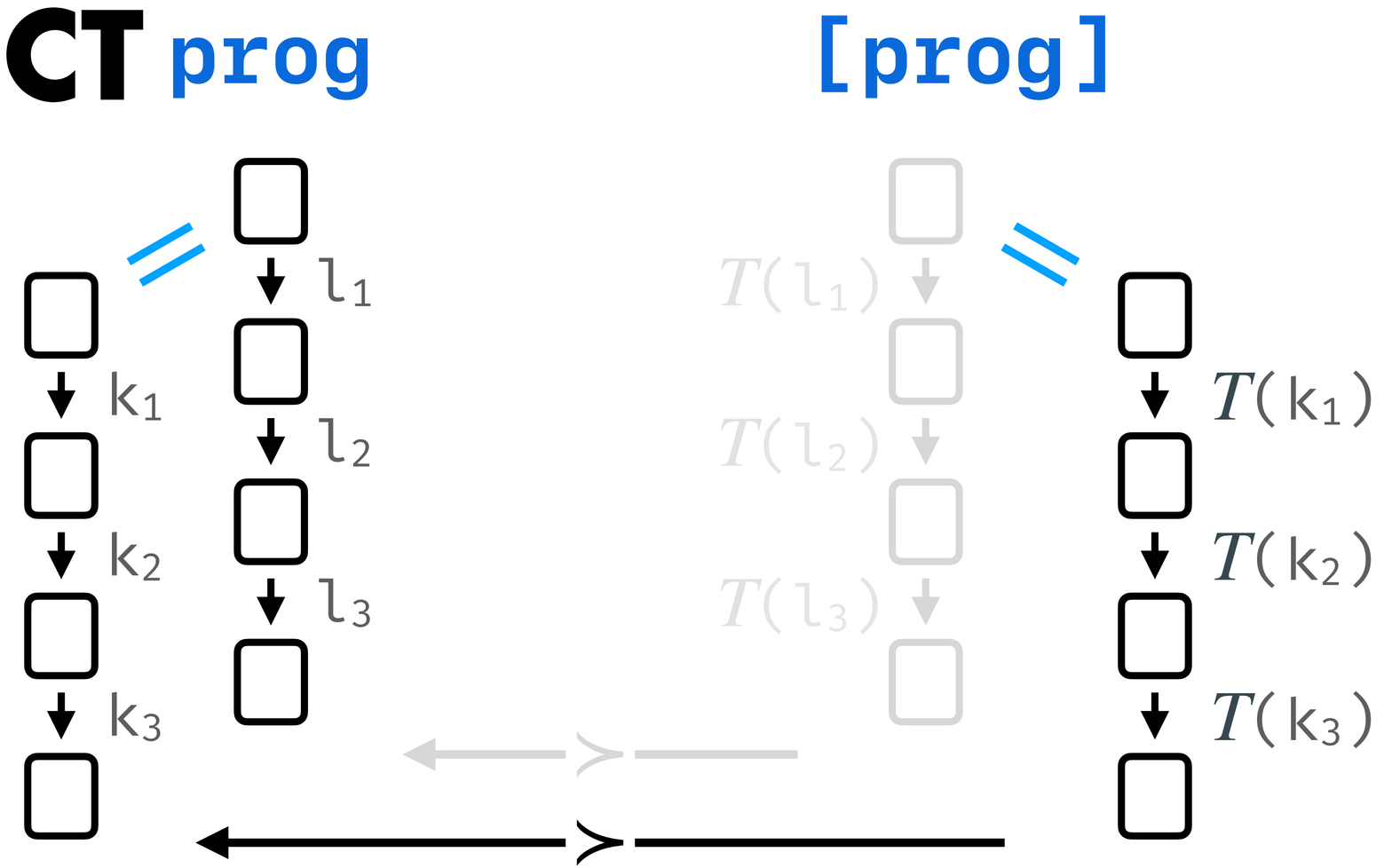


Proving "`[]` preserves CT"

`prog` CT $\Longrightarrow$ `[prog]` CT

T : leakage transformer

Cites!

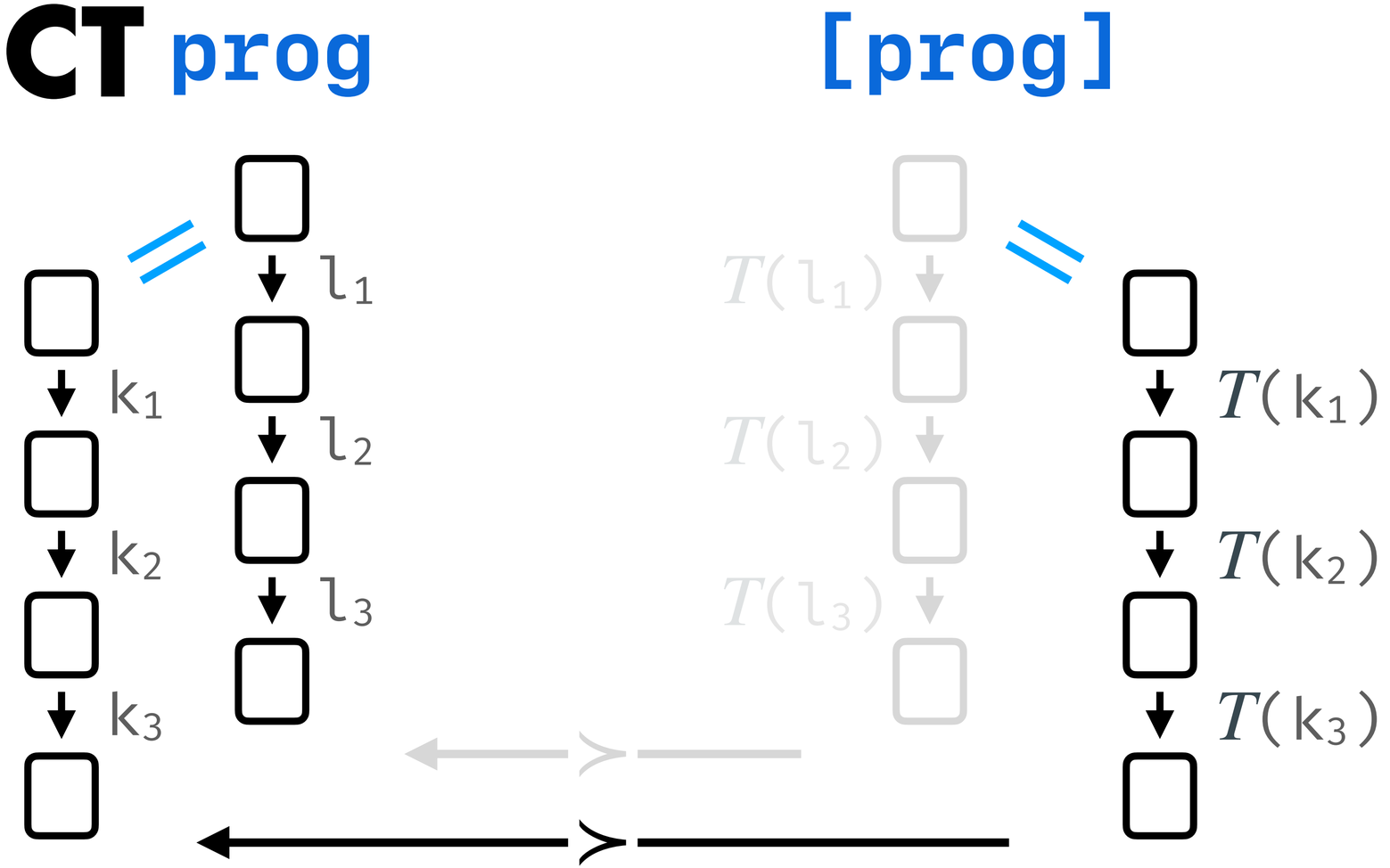


Proving "[] preserves CT"

prog CT $\Longrightarrow$ **[prog] CT**

T : leakage transformer

Cites!



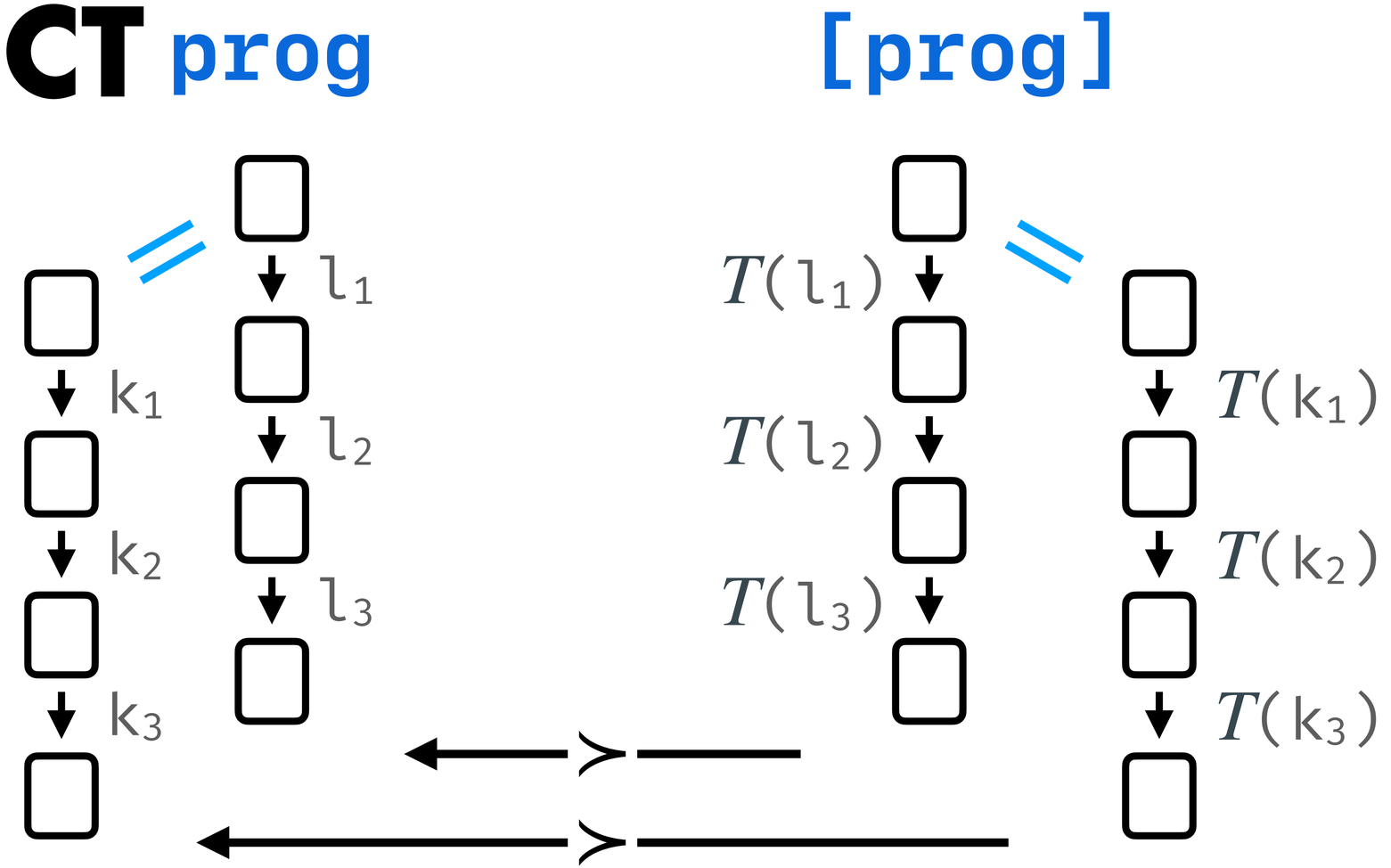
$k_1k_2k_3\ldots = l_1l_2l_3\ldots$

Proving "[] preserves CT"

`prog` **CT** $\Longrightarrow$ `[prog]` **CT**

T : leakage transformer

Cites!



$k_1k_2k_3\ldots = l_1l_2l_3\ldots$

$T(k_1k_2k_3\ldots) = T(l_1l_2l_3\ldots)$

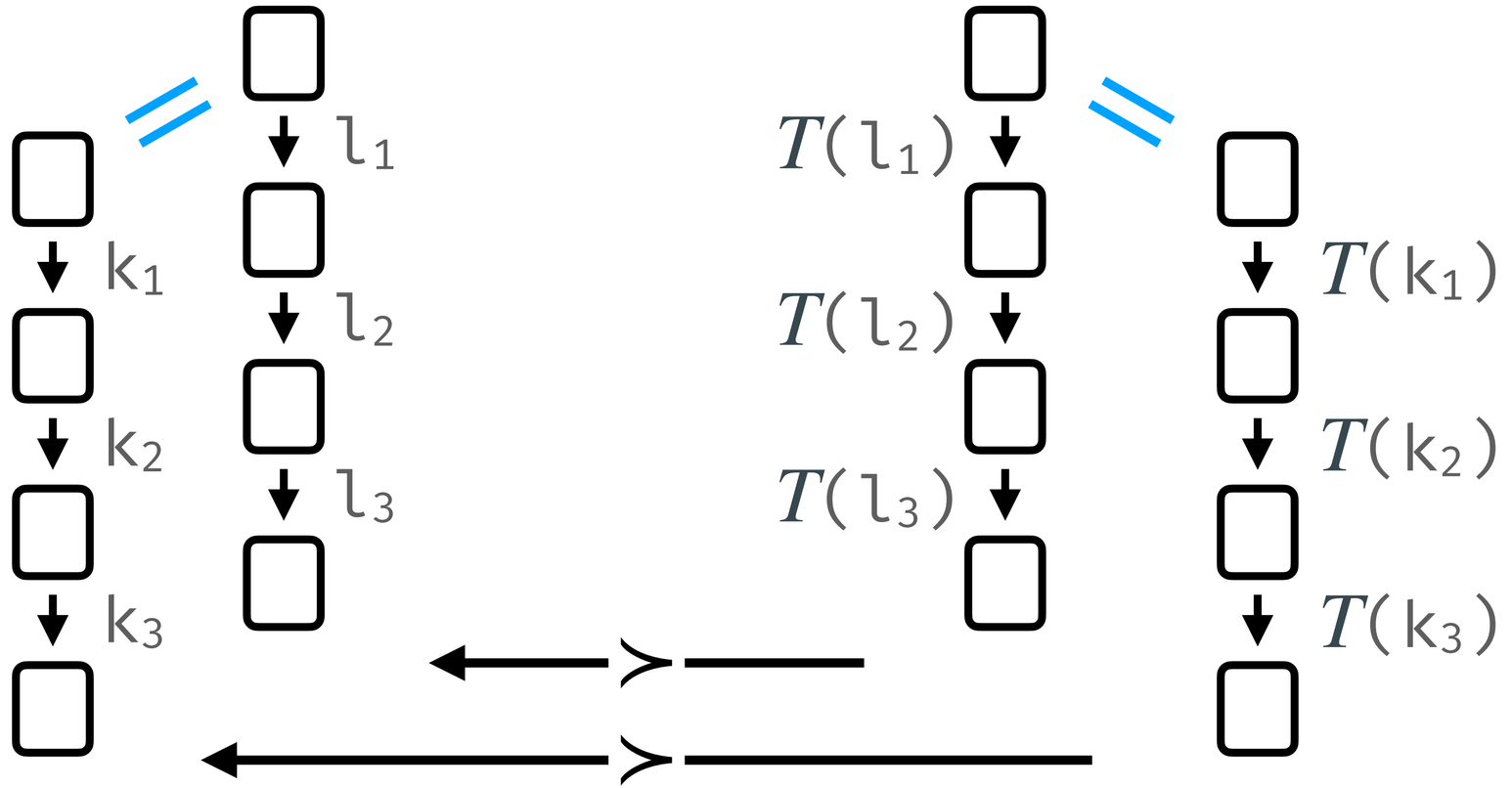
Proving "[] preserves CT"

prog CT $\Longrightarrow$ [prog] CT

T : leakage transformer

Cites!

CT prog $\Longrightarrow$ [prog] CT



$k_1k_2k_3\ldots = l_1l_2l_3\ldots$

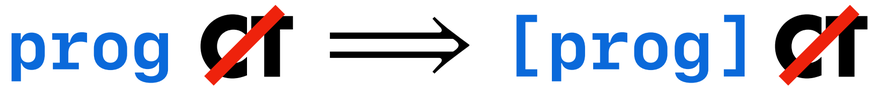
$T(k_1k_2k_3\ldots) = T(l_1l_2l_3\ldots)$

Proving "[] preserves CT"

CT prog $\Longrightarrow$ [prog] CT

THEOREM [Barthe et al. '18, '21]

$\succ$ + T $\Longrightarrow$ [] preserves CT.



Proving “[ ] reflects CT”

`prog` **CT** $\Longleftarrow$ `[prog]` **CT**

Proving “[ ] reflects CT”

Proving "[] reflects CT"

`prog` **CT** $\Longleftarrow$ `[prog]` **CT**

`prog`

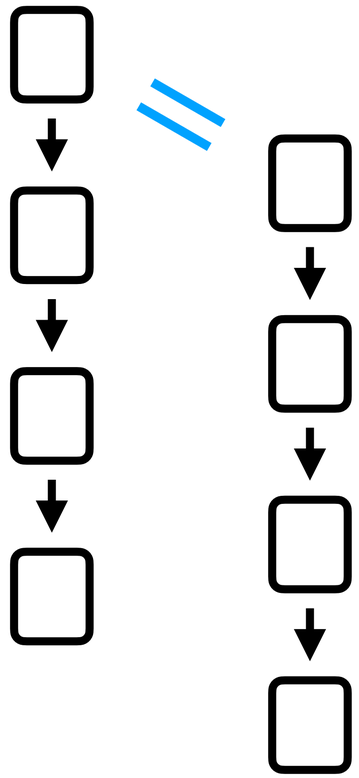
`[prog]`

Proving "[] reflects CT"

`prog` **CT** $\Longleftarrow$ `[prog]` **CT**

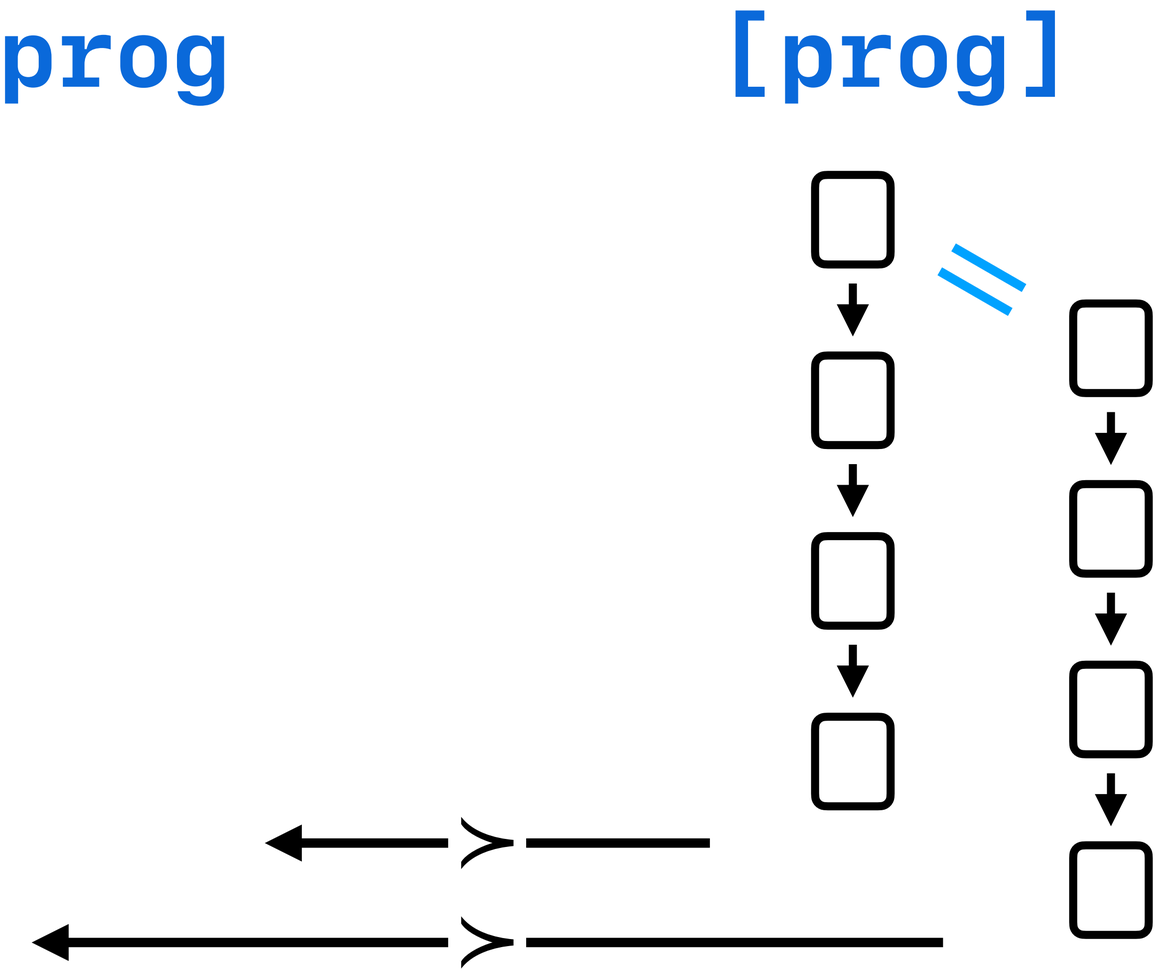
`prog`

`[prog]`



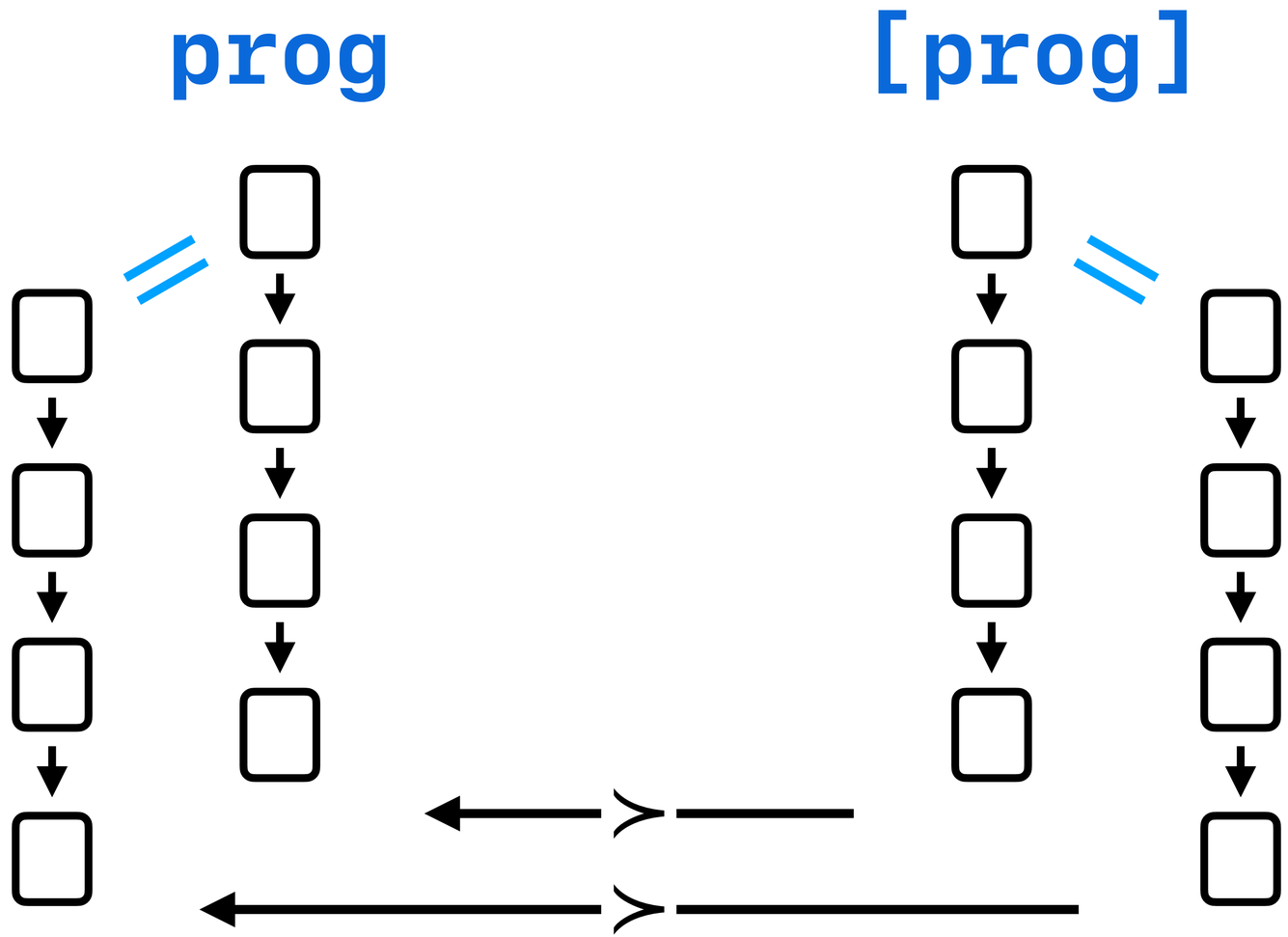
Proving "[] reflects CT"

`prog` **CT** $\Longleftarrow$ `[prog]` **CT**



Proving "[] reflects CT"

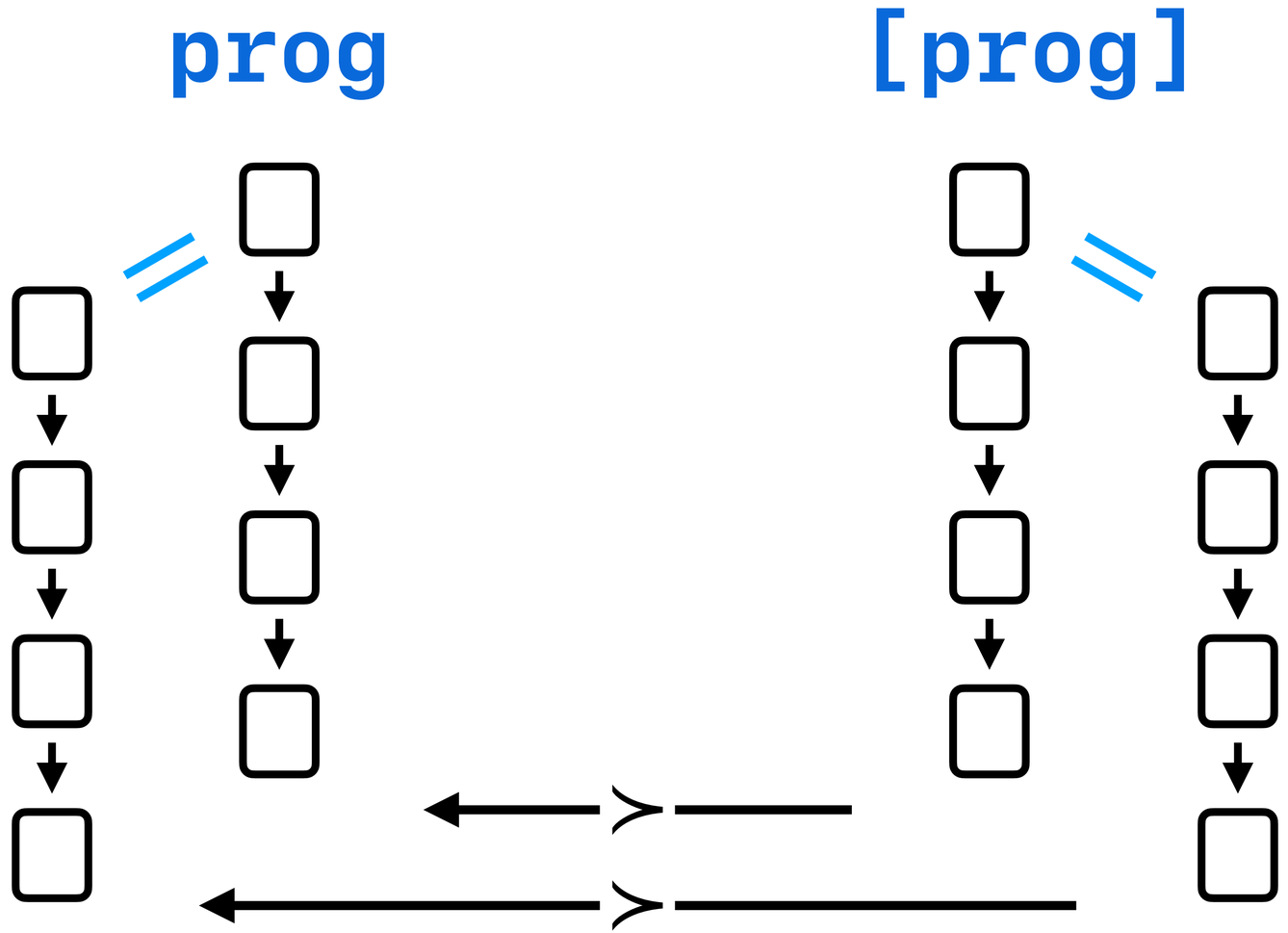
`prog` **CT** $\Longleftarrow$ `[prog]` **CT**



Proving "[] reflects CT"

prog CT $\Longleftarrow$ **[prog] CT**

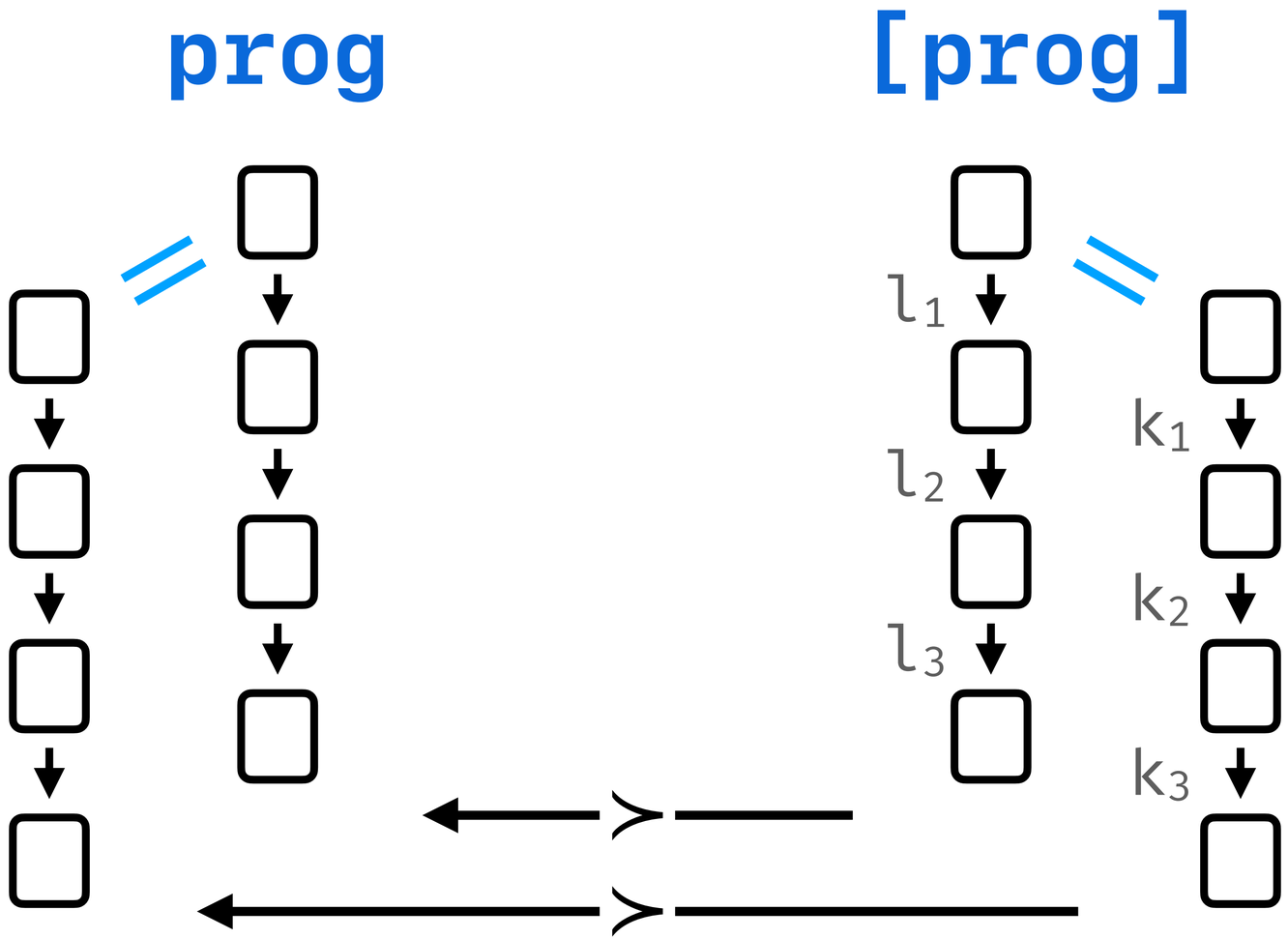
U : Reverse Transformer



Proving "[] reflects CT"

prog CT $\Longleftarrow$ **[prog] CT**

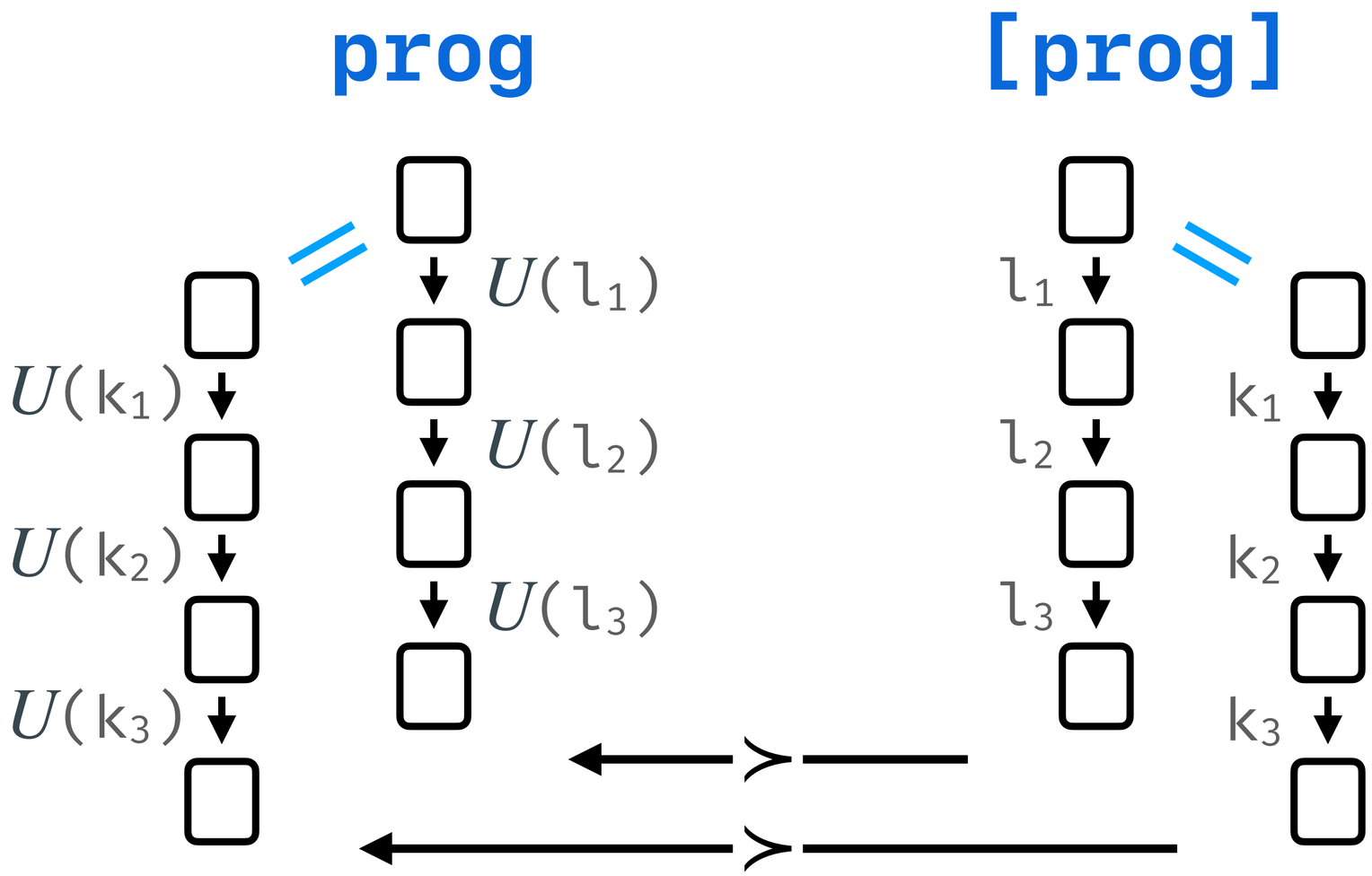
U : Reverse Transformer



Proving "[] reflects CT"

prog CT $\Longleftarrow$ **[prog] CT**

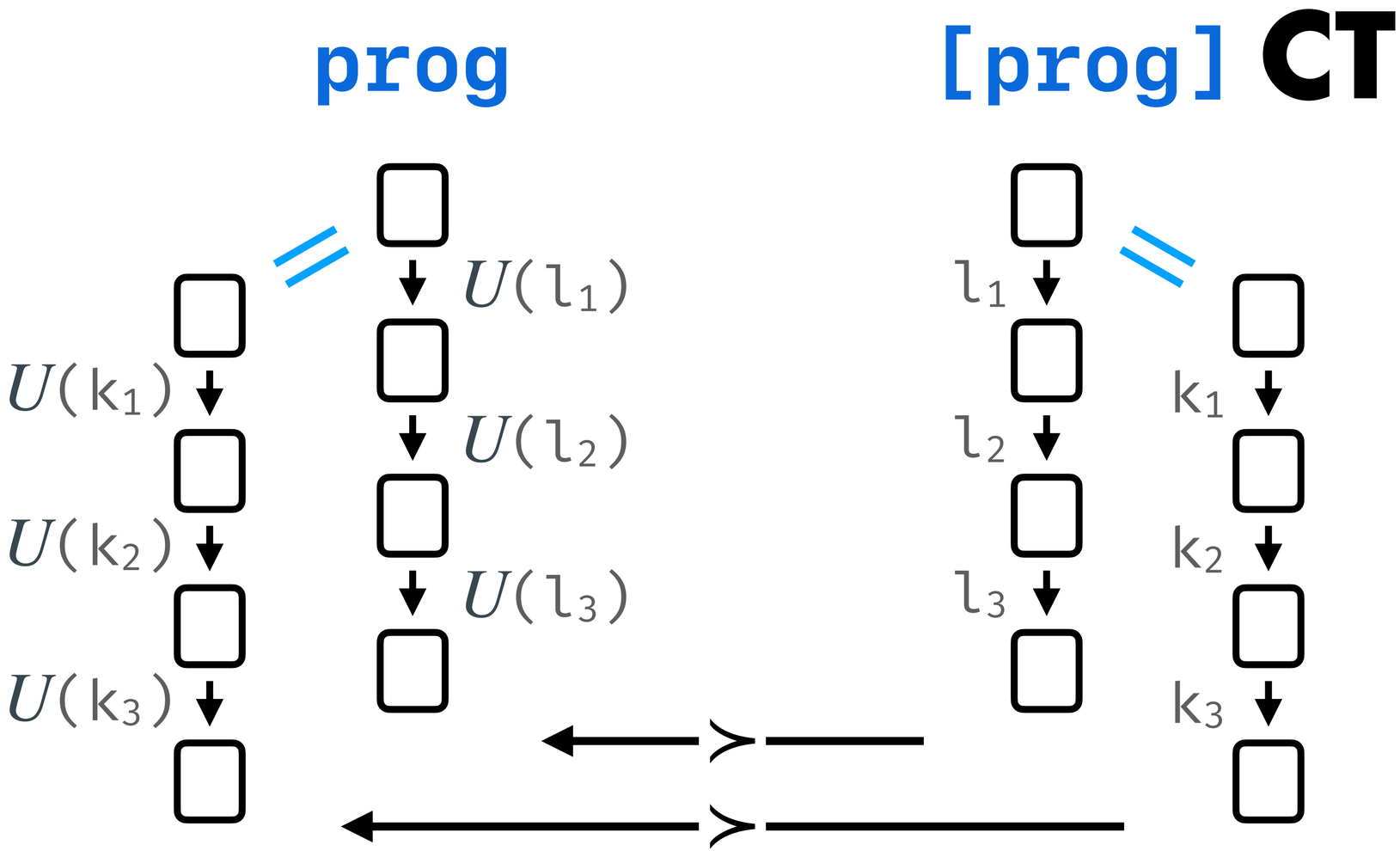
U : Reverse Transformer



Proving "[] reflects CT"

prog CT ⟸ [prog] CT

U : Reverse Transformer



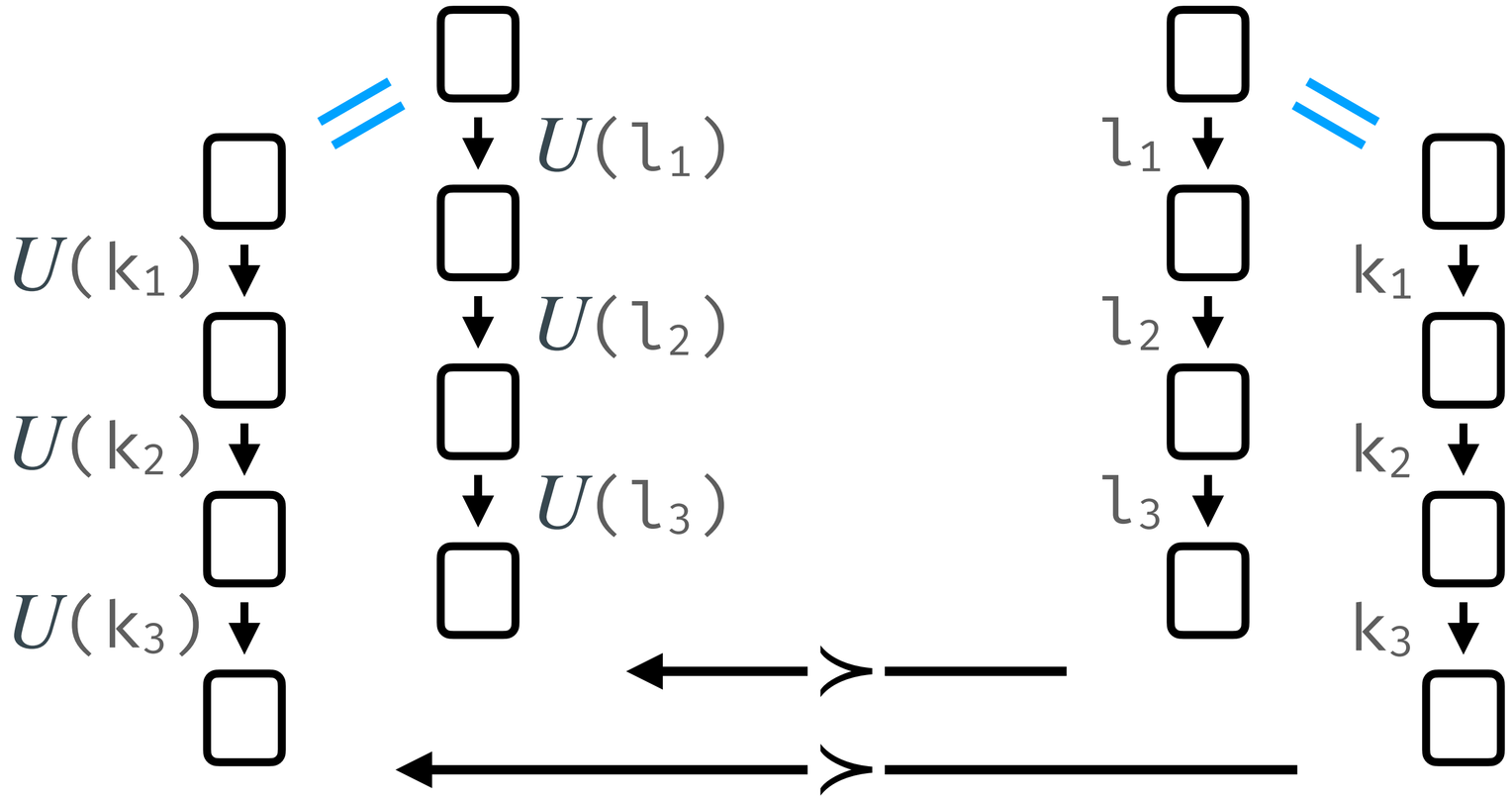
$k_1k_2k_3\ldots = l_1l_2l_3\ldots$

Proving "[] reflects CT"

prog CT ⟸ [prog] CT

U : Reverse Transformer

CT prog ⟸ [prog] CT



$U(k_1k_2k_3\ldots) = U(l_1l_2l_3\ldots)$ $\qquad$ $k_1k_2k_3\ldots = l_1l_2l_3\ldots$

Proving "[] reflects CT"

CT prog ⟸ [prog] CT

THEOREM

$\succ$ + reverse U $\Longrightarrow$ [] reflects CT.

Proving “[ ] is **CT** transparent”

Proving "`[]` is CT transparent"

`prog` **CT** $\iff$ `[prog]` **CT**

`[ ]` preserves **CT** + `[ ]` reflects **CT**

Proving "`[]` is CT transparent"

`prog` **CT** $\iff$ `[prog]` **CT**

`[ ]` preserves **CT**

transformer

T : `prog` leakage $\rightarrow$ `[prog]` leakage

+

`[ ]` reflects **CT**

reverse transformer

U : `[prog]` leakage $\rightarrow$ `prog` leakage

Proving "`[]` is CT transparent"

`prog` **CT** $\iff$ `[prog]` **CT**

`[ ]` preserves **CT**

transformer

T : `prog` leakage $\rightarrow$ `[prog]` leakage

exist for many `[ ]` already

+

`[ ]` reflects **CT**

reverse transformer

U : `[prog]` leakage $\rightarrow$ `prog` leakage

Proving "[] is CT transparent"

prog CT $\iff$ **[prog] CT**

[ ] preserves **CT**

transformer

$T : \underset{\text{leakage}}{\texttt{prog}} \rightarrow \underset{\text{leakage}}{\texttt{[prog]}}$

exist for many [ ] already

+

[ ] reflects **CT**

reverse transformer

$U : \underset{\text{leakage}}{\texttt{[prog]}} \rightarrow \underset{\text{leakage}}{\texttt{prog}}$

how?

Proving "`[]` is CT transparent"

`prog` **CT** $\iff$ `[prog]` **CT**

`[ ]` preserves **CT**

transformer

T : • `prog` leakage $\rightarrow$ `[prog]` leakage

exist for many `[ ]` already

+

`[ ]` reflects **CT**

reverse transformer

U : • `[prog]` leakage $\rightarrow$ `prog` leakage

how?

Idea: Flip T to get U

Proving "[] is CT transparent"

prog CT $\iff$ [prog] CT

[ ] preserves **CT** $+$ **[ ]** reflects **CT**

transformer

T : • **prog** leakage $\rightarrow$ **[prog]** leakage

exist for many **[ ]** already

reverse transformer

T^{-1} : • **[prog]** leakage $\rightarrow$ **prog** leakage

how?

Idea: Flip T to get U

Proving "[] is CT transparent"

`prog` **CT** $\Longleftrightarrow$ `[prog]` **CT**

[ ] preserves **CT**

transformer

T : • `prog` leakage $\rightarrow$ `[prog]` leakage

exist for many **[ ]** already

+

[ ] reflects **CT**

reverse transformer

T^{-1} : • `[prog]` leakage $\rightarrow$ `prog` leakage

how?

Idea: Flip T to get U

Sufficient Condition

T must be injective!

Proving "[] is CT transparent"

prog CT $\iff$ **[prog] CT**

[ ] preserves **CT** $+$ **[ ]** reflects **CT**

transformer

T : • **prog** leakage $\rightarrow$ **[prog]** leakage

exist for many **[ ]** already

reverse transformer

T^{-1} : • **[prog]** leakage $\rightarrow$ **prog** leakage

how?

Idea: Flip T to get U

Sufficient Condition

T must be injective! ← + details

Proving "[] is CT transparent"

`prog` **CT** $\iff$ `[prog]` **CT**

THEOREM

$\succ$ + injective T $\implies$ [ ] is CT transparent.

Proving "[] is CT transparent"

`prog` **CT** $\iff$ `[prog]` **CT**

THEOREM

$\succ$ + injective T $\implies$ [ ] is CT transparent.

[Loop Rotation] [Untiling] [Constant Folding]

[Unspilling] [Structural Analysis]

[Dead Assignment Elimination] [Dead Branch Elimination]

Making RetDec transparent

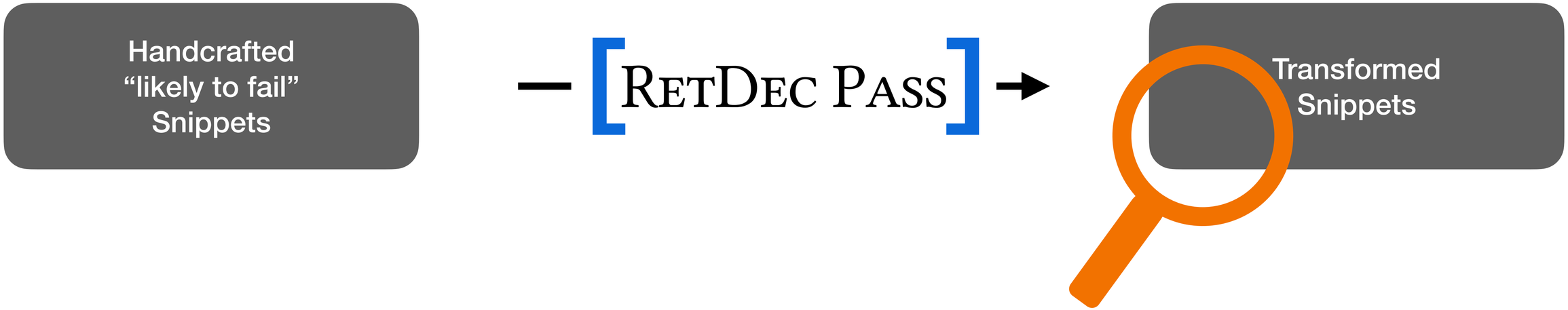
Making RetDec transparent

Making RetDec transparent

Handcrafted
“likely to fail”
Snippets

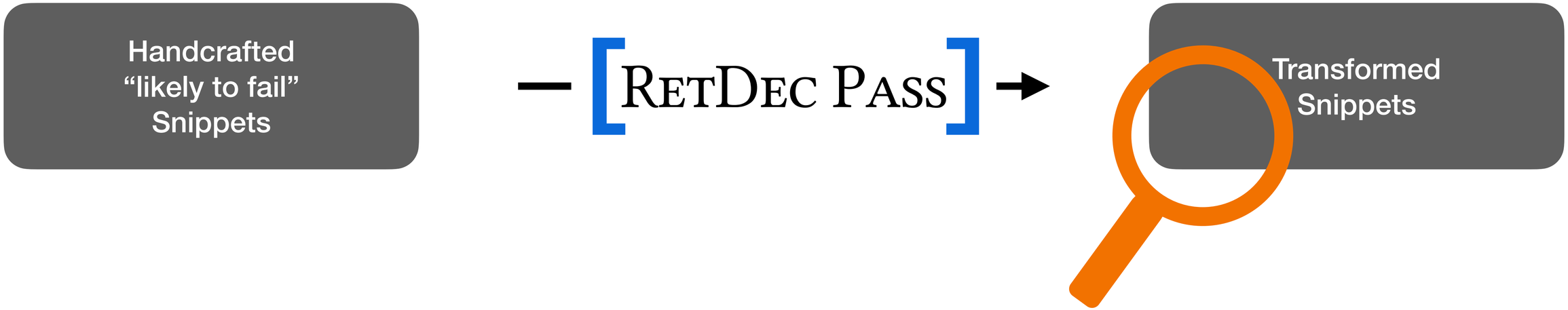
Making RetDec transparent

Making RetDec transparent



Transparent on Snippets?

Making RetDec transparent



Transparent on Snippets?

✓ Keep pass (49)

✗ Remove pass (13)

Making RetDec transparent

Handcrafted
“likely to fail”
Snippets

— [RetDec Pass] →

Transparent on Snippets?

✓ Keep pass (49)

✗ Remove pass (13)

↘ CT-RetDec

Making RetDec transparent

CT-RetDec

Making RetDec transparent

160 (10x16) Binaries
with and without
Real World Vulnerabilities

CT-RetDec

Making RetDec transparent

Making RetDec transparent

160 (10x16) Binaries
with and without
Real World Vulnerabilities

— [CT-RetDec] →

Decompiled Binaries

transparent on all

Making RetDec transparent

160 (10x16) Binaries
with and without
Real World Vulnerabilities

— [CT-RetDec] →

Decompiled Binaries

transparent on all

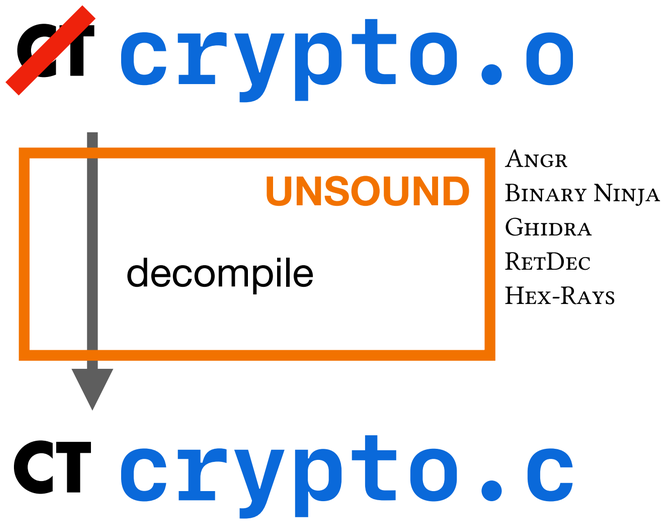
RetDec removes vulnerabilities from 45

appearing in OOPSLA 26

Decompiling for Constant Time Analysis

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter,
Youcef Bouzid, Sören van der Wall, Zhiyuan Zhang

s.van-der-wall@tu-bs.de



appearing in OOPSLA 26

Decompiling for Constant Time Analysis

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter,
Youcef Bouzid, Sören van der Wall, Zhiyuan Zhang

s.van-der-wall@tu-bs.de

~~CT~~ `crypto.o`

Angr
Binary Ninja
Ghidra
RetDec
Hex-Rays

CT `crypto.c`

[ ] CT transparent

WHEN

`prog` CT ⟺ `[prog]` CT

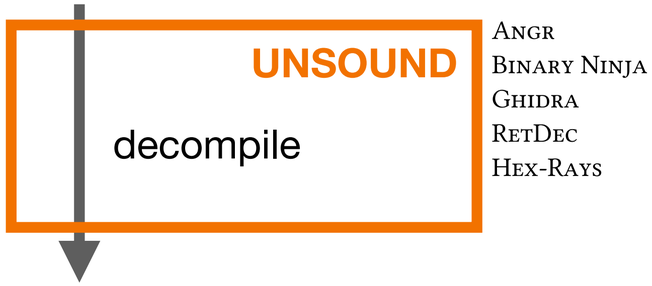
appearing in OOPSLA 26

Decompiling for Constant Time Analysis

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Youcef Bouzid, Sören van der Wall, Zhiyuan Zhang

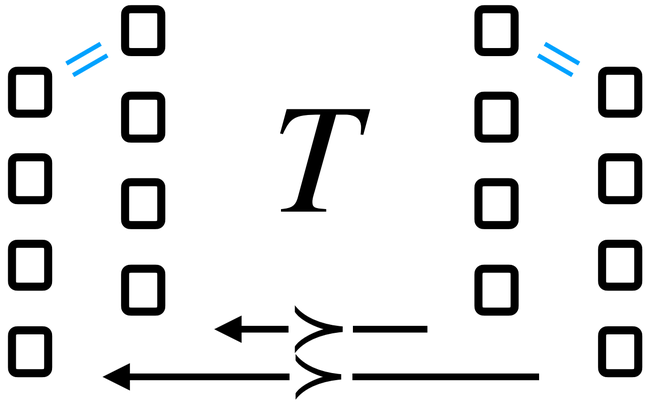
s.van-der-wall@tu-bs.de

~~CT~~ `crypto.o`



CT `crypto.c`

Proving "`[]` preserves CT"



`[ ]` CT transparent

WHEN

`prog` CT ⟺ `[prog]` CT

appearing in OOPSLA 26

Decompiling for Constant Time Analysis

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter,
Youcef Bouzid, Sören van der Wall, Zhiyuan Zhang

s.van-der-wall@tu-bs.de

~~CT~~ `crypto.o`

Angr
Binary Ninja
Ghidra
RetDec
Hex-Rays

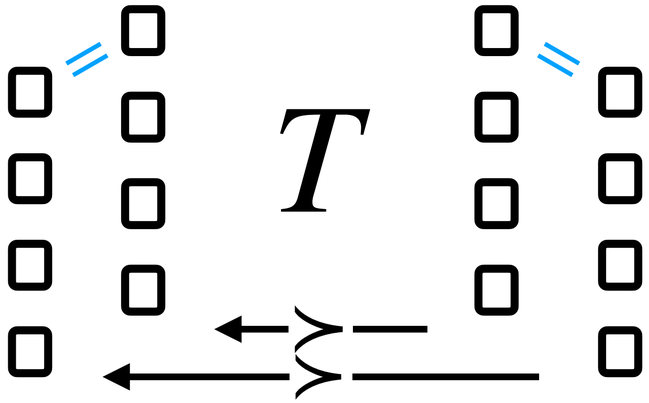
CT `crypto.c`

Proving "`[]` is CT transparent"

THEOREM

$\succ$ + injective T $\Longrightarrow$ `[]` is CT transparent.

Proving "`[]` preserves CT"



`[]` **CT** transparent

WHEN

`prog` **CT** $\Longleftrightarrow$ `[prog]` **CT**

appearing in OOPSLA 26

Decompiling for Constant Time Analysis

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter,
Youcef Bouzid, Sören van der Wall, Zhiyuan Zhang

s.van-der-wall@tu-bs.de

~~CT~~ `crypto.o`

Angr
Binary Ninja
Ghidra
RetDec
Hex-Rays

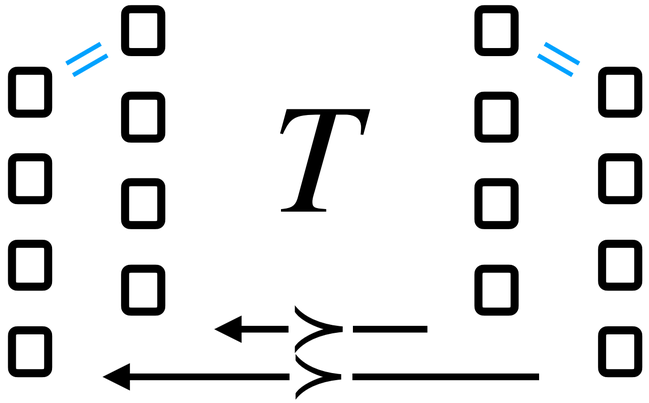
CT `crypto.c`

Proving "`[]` is CT transparent"

THEOREM

$\succ$ + injective T $\Longrightarrow$ `[]` is CT transparent.

Proving "`[]` preserves CT"



[CT-RetDec]

[ ] CT transparent

WHEN

`prog` **CT** $\Longleftrightarrow$ `[prog]` **CT**

appearing in OOPSLA 26

Decompiling for Constant Time Analysis

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter,
Youcef Bouzid, Sören van der Wall, Zhiyuan Zhang

s.van-der-wall@tu-bs.de